



Developer Guide

Amazon DynamoDB



API Version 2012-08-10

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon DynamoDB: Developer Guide

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is Amazon DynamoDB?	1
High availability and durability	1
Getting started with DynamoDB	2
DynamoDB tutorials	3
How it works	3
Cheat sheet	3
Core components	9
DynamoDB API	19
Supported data types and naming rules	22
Read consistency	29
Read/write capacity mode	30
Table classes	38
Partitions and data distribution	39
From SQL to NoSQL	43
Relational or NoSQL?	44
Characteristics of databases	47
Creating a table	50
Getting information about a table	52
Writing data to a table	54
Reading data from a table	58
Managing indexes	67
Modifying data in a table	73
Deleting data from a table	77
Removing a table	79
Additional resources for Amazon DynamoDB	80
Tools for coding and visualization	80
Prescriptive Guidance	81
Knowledge Center	82
Blog posts, repositories, and guides	83
Data modeling and design patterns	83
Training courses	84
Setting up DynamoDB	85
Setting up DynamoDB local (downloadable version)	85
Deploying	86

Usage notes	93
Release history	97
DynamoDB local telemetry	101
Setting up DynamoDB (web service)	103
Signing up for AWS	104
Granting programmatic access	104
Configuring your credentials	106
Integrating with other DynamoDB services	107
Accessing DynamoDB	108
Using the console	108
Using the AWS CLI	109
Downloading and configuring the AWS CLI	110
Using the AWS CLI with DynamoDB	110
Using the AWS CLI with DynamoDB local	111
Using the API	112
Using the NoSQL workbench	112
IP address ranges	113
Getting started with DynamoDB	115
Basic concepts	115
Prerequisites	115
Step 1: Create a table	116
Step 2: Write data	122
Step 3: Read data	128
Step 4: Update data	132
Step 5: Query data	135
Step 6: Create a global secondary index	139
Step 7: Query the global secondary index	144
Step 8: (Optional) clean up	148
Next steps	149
Getting started with DynamoDB and the AWS SDKs	150
Create a table	150
Create a DynamoDB table using an AWS SDK	150
Write an item	193
Write an item to a DynamoDB table using an AWS SDK	193
Read an item	217
Read an item from a DynamoDB table using an AWS SDK	218

Update an item	240
Update an item in a DynamoDB table using an AWS SDK	240
Delete an item	265
Delete an item in a DynamoDB table using an AWS SDK	266
Query a table	287
Query a DynamoDB table using an AWS SDK	287
Scan a table	319
Scan a DynamoDB table using an AWS SDK	287
Working with AWS SDKs	343
Programming with DynamoDB	345
Overview of AWS SDK support for DynamoDB	345
Programmatic interfaces	348
Low-level API	354
Error handling	360
Higher-level programming interfaces	367
Java 1.x: DynamoDBMapper	368
Java 2.x: DynamoDB Enhanced Client	439
.NET: Document model	439
.NET: Object persistence model	472
Running the code examples	513
Load sample data	515
Java code examples	515
.NET code examples	518
Programming with Python	521
About Boto	521
Boto documentation	522
Client and resource layers	523
Using batch_writer	526
Additional code examples	526
Sessions and thread safety	527
Config	527
Error handling	532
Logging	534
Event hooks	535
Pagination and the Paginator	536
Waiters	538

Programming with JavaScript	539
About AWS SDK for JavaScript	539
AWS SDK for JavaScript V3	540
JavaScript documentation	540
Abstraction layers	541
Marshall utility function	543
Reading items	544
Conditional writes	545
Pagination	546
Config	548
Waiters	551
Error handling	552
Logging	553
Considerations	554
Programming with Java 2.x	555
About AWS SDK for Java 2.x	556
Getting started	557
SDK for Java 2.x documentation	566
Supported interfaces	567
Additional code examples	581
Sync and async programming	581
HTTP clients	582
Config	584
Error handling	590
AWS request ID	591
Logging	592
Pagination	594
Data class annotations	596
Working with DynamoDB	597
Working with tables	597
Basic operations on tables	598
Considerations when changing read/write Capacity Mode	607
Considerations when choosing a table class	608
Provisioned capacity tables	609
Item sizes and formats	618
Managing throughput capacity with auto scaling	619

Tagging resources	642
Working with tables: Java	648
Working with tables: .NET	655
Working with global tables	665
Replicate data seamlessly across Regions with global tables	666
Provide security and access for your global tables with AWS KMS	667
How it works	668
Best practices and requirements	673
Tutorial: Creating a global table	676
Monitoring global tables	682
Using IAM with global tables	682
Determining the version	686
Upgrading global tables	688
Working with read and write operations	698
DynamoDB API	698
PartiQL query language	895
Working with indexes	941
Global secondary indexes	945
Local Secondary Indexes	1006
Working with transactions	1060
How it works	1061
Using IAM with transactions	1070
Example code	1073
Working with streams	1077
Options	1077
Working with Kinesis Data Streams	1079
Working with DynamoDB Streams	1097
Working with On-Demand backup and restore	1157
Using AWS Backup	1159
Using DynamoDB backups	1168
Working with point-in-time recovery	1188
How it works	1189
Before you begin	1191
Restoring a table to a point in time	1192
In-memory acceleration with DAX	1198
Use cases for DAX	1199

DAX usage notes	1200
How it works	1201
How DAX processes requests	1203
Item cache	1204
Query cache	1205
DAX cluster components	1206
Nodes	1207
Clusters	1207
Regions and availability zones	1208
Parameter groups	1209
Security groups	1209
Cluster ARN	1209
Cluster endpoint	1210
Node endpoints	1210
Subnet groups	1210
Events	1211
Maintenance window	1211
Creating a DAX cluster	1212
Creating an IAM service role for DAX to access DynamoDB	1213
Using the AWS CLI	1214
Using the console	1220
Consistency models	1225
Consistency among DAX cluster nodes	1226
DAX item cache behavior	1226
DAX query cache behavior	1229
Strongly consistent and transactional reads	1230
Negative caching	1231
Strategies for writes	1231
Developing with the DAX client	1235
Tutorial: Running a sample application	1235
Modifying an existing application to use DAX	1283
Managing DAX clusters	1284
IAM permissions for managing a DAX cluster	1285
Scaling a DAX cluster	1287
Customizing DAX cluster settings	1289
Configuring TTL settings	1290

Tagging support for DAX	1291
AWS CloudTrail integration	1292
Deleting a DAX cluster	1293
Monitoring DAX	1293
Monitoring tools	1293
Monitoring with CloudWatch	1295
Logging DAX operations using AWS CloudTrail	1319
DAX T3/T2 burstable instances	1319
DAX T2 instance family	1320
DAX T3 instance family	1320
DAX access control	1321
IAM service role for DAX	1322
IAM policy to allow DAX cluster access	1323
Case study: Accessing DynamoDB and DAX	1324
Access to DynamoDB, but no access with DAX	1326
Access to DynamoDB and to DAX	1328
Access to DynamoDB via DAX, but no direct access to DynamoDB	1333
DAX encryption at rest	1335
Enabling encryption at rest using the AWS Management Console	1337
DAX encryption in transit	1338
Using service-linked roles for DAX	1339
Service-linked role permissions for DAX	1340
Creating a service-linked role for DAX	1341
Editing a service-linked role for DAX	1342
Deleting a service-linked role for DAX	1342
Accessing DAX across AWS accounts	1343
Set up IAM	1344
Set up a VPC	1347
Modify the DAX client to allow cross-account access	1349
DAX cluster sizing guide	1353
Overview	1354
Estimating traffic	1354
Load testing	1355
API reference	1356
Data modeling	1357
Data modeling foundations	1358

Single table design	1359
Multiple table design	1361
Data modeling building blocks	1362
Composite sort key	1363
Multi-tenancy	1364
Sparse index	1366
Time to live	1367
Time to live archival	1368
Vertical partitioning	1369
Write sharding	1372
Data modeling schema design packages	1373
Prerequisites	1374
Social network	1375
Gaming profile	1384
Complaint management system	1393
Recurring payments	1410
Device status updates	1415
Online shop	1430
Migrating to DynamoDB	1455
Reasons to migrate	1455
Considerations when migrating	1456
How it works	1458
Migration tools	1459
Choosing a migration strategy	1460
Offline migration	1463
Hybrid migration	1465
Online - migrating each table 1:1	1466
Online - migrating with a custom staging table	1467
NoSQL Workbench	1471
Download	1472
Install	1473
Data modeler	1477
Creating a new model	1477
Importing an existing model	1485
Exporting a model	1487
Editing an existing model	1489

Data visualizer	1493
Adding sample data	1493
Importing from CSV	1496
Facets	1497
Aggregate view	1500
Committing a data model	1501
Operation builder	1504
Connecting to datasets	1505
Building operations	1510
Cloning tables	1535
Exporting to CSV	1538
Sample data models	1539
Employee data model	1539
Discussion forum data model	1540
Music library data model	1540
Ski resort data model	1541
Credit card offers data model	1541
Bookmarks data model	1542
Release history	1542
Code examples	1548
Actions	1556
Create a table	1557
Delete a table	1600
Delete an item from a table	1615
Get a batch of items	1637
Get an item from a table	1659
Get information about a table	1681
List tables	1696
Put an item in a table	1713
Query a table	1737
Run a PartiQL statement	1769
Run batches of PartiQL statements	1791
Scan a table	1817
Update an item in a table	1841
Write a batch of items	1867
Scenarios	1895

Accelerate reads with DAX	1896
Get started with tables, items, and queries	1904
Query a table by using batches of PartiQL statements	2054
Query a table using PartiQL	2113
Use a document model	2166
Use a high-level object persistence model	2182
Cross-service examples	2192
Build an app to submit data to a DynamoDB table	2192
Create a REST API to track COVID-19 data	2194
Create a messenger application	2195
Create a serverless application to manage photos	2196
Create a web application to track DynamoDB data	2200
Create a websocket chat application	2202
Detect PPE in images	2203
Invoke a Lambda function from a browser	2204
Save EXIF and other image information	2205
Use API Gateway to invoke a Lambda function	2206
Use Step Functions to invoke Lambda functions	2207
Use scheduled events to invoke a Lambda function	2208
Security	2210
AWS managed policies	2211
AWS managed policies	2211
AmazonDynamoDBReadOnlyAccess	2211
DynamoDB updates to AWS managed policies	2212
Resource-based policies	2214
Create table	2215
Attach resource-based policy	2221
Attach policy to a stream	2225
Remove resource-based policy	2228
Cross-account access	2229
Blocking public access	2230
IAM actions	2233
IAM authorization	2237
Examples	2238
Considerations	2244
Best practices	2245

Data protection	2247
Encryption at rest	2247
Data protection in DAX	2273
Internetwork traffic privacy	2274
IAM	2275
Identity and Access Management	2275
Using conditions	2309
Identity and access management in DAX	2332
Compliance validation	2332
Resilience	2334
Infrastructure security	2335
Using VPC endpoints	2335
AWS PrivateLink for DynamoDB	2344
Types of Amazon VPC endpoints	2345
Considerations when using AWS PrivateLink for Amazon DynamoDB	2346
Creating an Amazon VPC endpoint	2346
Accessing Amazon DynamoDB interface endpoints	2347
Accessing DynamoDB tables and control API operations from DynamoDB interface endpoints	2347
Updating an on-premises DNS configuration	2349
Creating an Amazon VPC endpoint policy	2351
Configuration and vulnerability analysis	2352
Security best practices	2353
Preventative security best practices	2353
Detective security best practices	2355
Monitoring	2359
Logging and monitoring in DynamoDB	2359
Monitoring tools	2360
Monitoring with Amazon CloudWatch	2361
Logging DynamoDB operations by using AWS CloudTrail	2393
Logging and monitoring in DAX	2416
Contributor Insights	2417
How it works	2417
Getting started	2423
Using IAM	2429
Using AWS User Notifications User Notifications with Amazon DynamoDB	2434

Best practices	2435
NoSQL design	2435
NoSQL vs. RDBMS	2436
Two key concepts	2436
General approach	2437
NoSQL Workbench	2438
Deletion protection	2438
The DynamoDB Well-Architected Lens	2439
Cost optimization	2439
Conducting the Amazon DynamoDB Well-Architected Lens review	2488
The pillars of the Amazon DynamoDB Well-Architected Lens	2488
Partition key design	2491
Burst capacity	2491
Adaptive capacity	2492
Distributing workloads	2493
Write sharding	2495
Uploading data efficiently	2496
Sort key design	2498
Version control	2498
Secondary indexes	2499
General guidelines	2500
Sparse indexes	2503
Aggregation	2505
GSI overloading	2506
GSI sharding	2508
Creating a replica	2509
Large items	2510
Compression	2510
Vertical partitioning	2511
Using Amazon S3	2511
Time series data	2512
Design pattern for time series data	2512
Time series table examples	2512
Many-to-many relationships	2513
Adjacency lists	2514
Materialized graphs	2515

Hybrid DynamoDB–RDBMS	2520
Not migrating	2520
Hybrid system implementation	2521
Relational modeling	2522
Traditional relational database models	2522
How DynamoDB eliminates the need for JOIN operations	2524
How DynamoDB transactions eliminate overhead to the write process	2525
First steps	2526
Example	2528
Querying and scanning	2532
Scan performance	2532
Avoid spikes	2532
Parallel scans	2535
Table design	2537
Global table design	2537
Global table design	2538
Key facts	2538
Use cases	2540
Write modes	2540
Request routing	2549
Evacuating a Region	2558
Throughput capacity with global tables	2561
Checklist and FAQ for global tables	2562
Control plane	2569
Billing and Usage Reports	2570
Throughput Capacity	2573
Streams	2577
Storage	2577
Backup and Restore	2578
Data Transfer	2582
CloudWatch	2582
DAX	2583
Using DynamoDB with other AWS services	2586
Integrating with Amazon Cognito	2586
Integrating with Amazon Redshift	2588
Integrating with Amazon EMR	2590

Overview	2590
Tutorial: Working with Amazon DynamoDB and Apache Hive	2591
Creating an external table in Hive	2600
Processing HiveQL statements	2603
Querying data in DynamoDB	2605
Copying data to and from Amazon DynamoDB	2607
Performance tuning	2621
Integrating with S3	2626
Import from Amazon S3	2627
Export to Amazon S3	2648
Integrating with Amazon OpenSearch Service	2672
How it works	2673
Creating an integration	2674
Next steps	2674
Handling breaking changes	2674
Integration best practices	2678
Creating a snapshot	2678
Change data capture	2679
Zero-ETL integration with OpenSearch Service	2679
Quotas and limits	2683
Read/write capacity mode and throughput	2683
Capacity unit sizes (for provisioned tables)	2684
Request unit sizes (for on-demand tables)	2684
Throughput default quotas	2684
Increasing or decreasing throughput (for provisioned tables)	2686
Reserved Capacity	2687
Import quotas	2687
Contributor Insights	2688
Tables	2688
Table size	2688
Maximum number of tables per account per region	2688
Global tables	2688
Secondary indexes	2690
Secondary indexes per table	2690
Projected Secondary Index attributes per table	2690
Partition keys and sort keys	2690

Partition key length	2690
Partition key values	2690
Sort key length	2690
Sort key values	2691
Naming rules	2691
Table names and Secondary Index names	2691
Attribute names	2691
Data types	2692
String	2692
Number	2692
Binary	2692
Items	2693
Item size	2693
Item size for tables with Local Secondary Indexes	2693
Attributes	2693
Attribute name-value pairs per item	2693
Number of values in list, map, or set	2693
Attribute values	2693
Nested attribute depth	2694
Expression parameters	2694
Lengths	2694
Operators and operands	2694
Reserved words	2694
DynamoDB transactions	2695
DynamoDB Streams	2695
Simultaneous readers of a shard in DynamoDB Streams	2695
Maximum write capacity for a table with DynamoDB Streams enabled	2695
DynamoDB Accelerator (DAX)	2696
AWS region availability	2696
Nodes	2696
Parameter groups	2696
Subnet groups	2696
API-specific limits	2697
DynamoDB encryption at rest	2699
Table export to Amazon S3	2700
Backup and restore	2700

API reference	2701
Troubleshooting	2702
Latency	2702
Throttling	2704
Make sure provisioned mode tables have adequate capacity	2704
Consider switching to on-demand mode	2705
Distribute read and write operations evenly between partition keys	2705
Increase your table-level read or write throughput quota	2706
Appendix	2707
Troubleshooting SSL/TLS connection establishment issues	2707
Testing your application or service	2707
Testing your client browser	2708
Updating your software application client	2708
Updating your client browser	2709
Manually updating your certificate bundle	2709
Example tables and data	2710
Sample data files	2711
Creating example tables and uploading data	2724
Creating example tables and uploading data - Java	2724
Creating example tables and uploading data - .NET	2734
Example application using AWS SDK for Python (Boto3)	2746
Step 1: Deploy and test locally	2748
Step 2: Examine the data model and implementation details	2752
Step 3: Deploy in production	2762
Step 4: Clean up resources	2771
Integrating with AWS Data Pipeline	2772
Prerequisites to export and import data	2775
Exporting data from DynamoDB to Amazon S3	2784
Importing data from Amazon S3 to DynamoDB	2785
Troubleshooting	2787
Predefined templates for AWS Data Pipeline and DynamoDB	2788
Amazon DynamoDB Storage Backend for Titan	2789
Reserved words in DynamoDB	2789
Legacy conditional parameters	2802
AttributesToGet	2804
AttributeUpdates	2805

ConditionalOperator	2808
Expected	2808
KeyConditions	2814
QueryFilter	2817
ScanFilter	2819
Writing conditions with legacy parameters	2821
Previous low-level API version (2011-12-05)	2830
BatchGetItem	2831
BatchWriteItem	2838
CreateTable	2846
DeleteItem	2853
DeleteTable	2860
DescribeTables	2864
GetItem	2868
ListTables	2872
PutItem	2875
Query	2882
Scan	2896
UpdateItem	2914
UpdateTable	2924
AWS SDK for Java 1.x examples	2929
DAX and Java SDK v1	2929
Modifying an existing SDK for Java 1.x application to use DAX	2941
Querying global secondary indexes with SDK for Java 1.x	2946
Document history	2951
Earlier updates	2970
Legacy features	3004
Global tables version 2017.11.29 (Legacy)	3004
How it works	3004
Best Practices and Requirements	3010
Creating a global table	3013
Monitoring global tables	3018
Using IAM with global tables	3019

What is Amazon DynamoDB?

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data. For more information, see [DynamoDB encryption at rest](#).

With DynamoDB, you can create database tables that can store and retrieve any amount of data and serve any level of request traffic. You can scale up or scale down your tables' throughput capacity without downtime or performance degradation. You can use the AWS Management Console to monitor resource utilization and performance metrics.

DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival for regulatory compliance needs. For more information, see [Using On-Demand backup and restore for DynamoDB](#).

You can create on-demand backups and enable point-in-time recovery for your Amazon DynamoDB tables. Point-in-time recovery helps protect your tables from accidental write or delete operations. With point-in-time recovery, you can restore a table to any point in time during the last 35 days. For more information, see [Point-in-time recovery: How it works](#).

DynamoDB allows you to delete expired items from tables automatically to help you reduce storage usage and the cost of storing data that is no longer relevant. For more information, see [Time to Live \(TTL\)](#).

High availability and durability

DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid-state disks (SSDs) and is automatically replicated across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables in sync across AWS Regions. For more information, see [Global tables - multi-Region replication for DynamoDB](#).

Getting started with DynamoDB

We recommend that you begin by reading the following sections:

- [Amazon DynamoDB: How it works](#)—To learn essential DynamoDB concepts.
- [Setting up DynamoDB](#)—To learn how to set up DynamoDB (the downloadable version or the web service).
- [Accessing DynamoDB](#)—To learn how to access DynamoDB using the console, AWS CLI, or API.

From there, you have two options to quickly get started with DynamoDB:

- [Getting started with DynamoDB](#)
- [Getting started with DynamoDB and the AWS SDKs](#)

To learn more about application development, see the following:

- [Programming with DynamoDB and the AWS SDKs](#)
- [Working with tables, items, queries, scans, and indexes](#)

To quickly find recommendations for maximizing performance and minimizing throughput costs, see the following: [Best practices for designing and architecting with DynamoDB](#). To learn how to tag DynamoDB resources, see [Adding tags and labels to resources](#).

For best practices, how-to guides, and tools, see [Amazon DynamoDB resources](#).

You can use AWS Database Migration Service (AWS DMS) to migrate data from a relational database or MongoDB to a DynamoDB table. For more information, see the [AWS Database Migration Service User Guide](#).

To learn how to use MongoDB as a migration source, see [Using MongoDB as a source for AWS Database Migration Service](#). To learn how to use DynamoDB as a migration target, see [Using an Amazon DynamoDB database as a target for AWS Database Migration Service](#).

DynamoDB tutorials

The following tutorials present complete end-to-end procedures to familiarize yourself with DynamoDB. These tutorials can be completed with the free tier of AWS and will give you practical experience using DynamoDB.

- [Build an Application Using a NoSQL Key-Value Data Store](#)
- [Create and Query a NoSQL Table with Amazon DynamoDB](#)

Amazon DynamoDB: How it works

The following sections provide an overview of Amazon DynamoDB service components and how they interact.

After you read this introduction, try working through the [Creating tables and loading data for code examples in DynamoDB](#) section, which walks you through the process of creating sample tables, uploading data, and performing some basic database operations.

For language-specific tutorials with sample code, see [Getting started with DynamoDB and the AWS SDKs](#).

Topics

- [Cheat sheet for DynamoDB](#)
- [Core components of Amazon DynamoDB](#)
- [DynamoDB API](#)
- [Supported data types and naming rules in Amazon DynamoDB](#)
- [Read consistency](#)
- [Read/write capacity mode](#)
- [Table classes](#)
- [Partitions and data distribution](#)

Cheat sheet for DynamoDB

This cheat sheet provides a quick reference for working with Amazon DynamoDB and its various AWS SDKs.

Initial setup

1. [Sign up for AWS.](#)
2. [Get an AWS access key](#) to access DynamoDB programmatically.
3. [Configure your DynamoDB credentials.](#)

See also:

- [Setting up DynamoDB \(web service\)](#)
- [Getting started with DynamoDB](#)
- [Basic overview of core components](#)

SDK or CLI

Choose your preferred [SDK](#), or set up the [AWS CLI](#).

 **Note**

When you use the AWS CLI on Windows, a backslash (\) that is not inside a quote is treated as a carriage return. Also, you must escape any quotes and braces inside other quotes. For an example, see the **Windows** tab in "Create a table" in the next section.

See also:

- [AWS CLI with DynamoDB](#)
- [Getting started with DynamoDB - step 2](#)

Basic actions

This section provides code for basic DynamoDB tasks. For more information about these tasks, see [Getting started with DynamoDB and the AWS SDKs](#).

Create a table

Default

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
   AttributeName=Artist,AttributeType=S \
   AttributeName=SongTitle,AttributeType=S \
--key-schema \
   AttributeName=Artist,KeyType=HASH \
   AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5
```

Windows

```
aws dynamodb create-table ^
--table-name Music ^
--attribute-definitions ^
   AttributeName=Artist,AttributeType=S ^
   AttributeName=SongTitle,AttributeType=S ^
--key-schema ^
   AttributeName=Artist,KeyType=HASH ^
   AttributeName=SongTitle,KeyType=RANGE ^
--provisioned-throughput ^
    ReadCapacityUnits=10,WriteCapacityUnits=5
```

Write item to a table

```
aws dynamodb put-item \ --table-name Music \ --item file://item.json
```

Read item from a table

```
aws dynamodb get-item \ --table-name Music \ --item file://item.json
```

Delete item from a table

```
aws dynamodb delete-item --table-name Music --key file://key.json
```

Query a table

```
aws dynamodb query --table-name Music  
--key-condition-expression "ArtistName=:Artist and SongName=:Songtitle"
```

Delete a table

```
aws dynamodb delete-table --table-name Music
```

List table names

```
aws dynamodb list-tables
```

Naming rules

- All names must be encoded using UTF-8 and are case sensitive.
- Table names and index names must be between 3 and 255 characters long, and can contain only the following characters:
 - a-z
 - A-Z
 - 0-9
 - _(underscore)
 - -(dash)
 - .(dot)
- Attribute names must be at least one character long, and less than 64 KB in size.

For more information, see [Naming rules](#).

Service quota basics

Read and write units

- **Read capacity unit (RCU)** – One strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.
- **Write capacity unit (WCU)** – One write per second, for items up to 1 KB in size.

Table limits

- **Table size** – There is no practical limit on table size. Tables are unconstrained in terms of the number of items or the number of bytes.
- **Number of tables** – For any AWS account, there is an initial quota of 2,500 tables per AWS Region.
- **Page size limit for query and scan** – There is a limit of 1 MB per page, per query or scan. If your query parameters or scan operation on a table result in more than 1 MB of data, DynamoDB returns the initial matching items. It also returns a `LastEvaluatedKey` property that you can use in a new request to read the next page.

Indexes

- **Local secondary indexes (LSIs)** – You can define a maximum of five local secondary indexes. LSIs are primarily useful when an index must have strong consistency with the base table.
- **Global secondary indexes (GSIs)** – There is a default quota of 20 global secondary indexes per table.
- **Projected secondary index attributes per table** – You can project a total of up to 100 attributes into all of a table's local and global secondary indexes. This only applies to user-specified projected attributes.

Partition keys

- The minimum length of a partition key value is 1 byte. The maximum length is 2048 bytes.
- There is no practical limit on the number of distinct partition key values, for tables or for secondary indexes.
- The minimum length of a sort key value is 1 byte. The maximum length is 1024 bytes.
- In general, there is no practical limit on the number of distinct sort key values per partition key value. The exception is for tables with secondary indexes.

For more information on secondary indexes, partition key design, and sort key design, see [Best practices](#).

Limits for commonly used data types

- **String** – The length of a string is constrained by the maximum item size of 400 KB. Strings are Unicode with UTF-8 binary encoding.
- **Number** – A number can have up to 38 digits of precision, and can be positive, negative, or zero.
- **Binary** – The length of a binary is constrained by the maximum item size of 400 KB. Applications that work with binary attributes must encode the data in base64 encoding before sending it to DynamoDB.

For a full list of supported data types, see [Data types](#). For more information, also see [Service quotas](#).

Items, attributes, and expression parameters

The maximum item size in DynamoDB is 400 KB, which includes both attribute name binary length (UTF-8 length) and attribute value binary lengths (UTF-8 length). The attribute name counts towards the size limit.

There is no limit on the number of values in a list, map, or set, as long as the item that contains the values fits within the 400-KB item size limit.

For expression parameters, the maximum length of any expression string is 4 KB.

For more information about item size, attributes, and expression parameters, see [Service quotas](#).

More information

- [Security](#)
- [Monitoring and logging](#)
- [Working with streams](#)
- [Backups and Point-in-time recovery](#)
- [Integrating with other AWS services](#)
- [API reference](#)
- [Architecture Center: Database Best Practices](#)
- [Video tutorials](#)
- [DynamoDB forum](#)

Core components of Amazon DynamoDB

In DynamoDB, tables, items, and attributes are the core components that you work with. A *table* is a collection of *items*, and each item is a collection of *attributes*. DynamoDB uses primary keys to uniquely identify each item in a table and secondary indexes to provide more querying flexibility. You can use DynamoDB Streams to capture data modification events in DynamoDB tables.

There are limits in DynamoDB. For more information, see [Service, account, and table quotas in Amazon DynamoDB](#).

The following video will give you an introductory look at tables, items, and attributes.

Tables, items, and attributes

Tables, items, and attributes

The following are the basic DynamoDB components:

- **Tables** – Similar to other database systems, DynamoDB stores data in tables. A *table* is a collection of data. For example, see the example table called *People* that you could use to store personal contact information about friends, family, or anyone else of interest. You could also have a *Cars* table to store information about vehicles that people drive.
- **Items** – Each table contains zero or more items. An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a *People* table, each item represents a person. For a *Cars* table, each item represents one vehicle. Items in DynamoDB are similar in many ways to rows, records, or tuples in other database systems. In DynamoDB, there is no limit to the number of items you can store in a table.
- **Attributes** – Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called *PersonID*, *LastName*, *FirstName*, and so on. For a *Department* table, an item might have attributes such as *DepartmentID*, *Name*, *Manager*, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.

The following diagram shows a table named *People* with some example items and attributes.

People

```
{  
    "PersonID": 101,  
    "LastName": "Smith",  
    "FirstName": "Fred",  
    "Phone": "555-4321"  
}  
  
{  
    "PersonID": 102,  
    "LastName": "Jones",  
    "FirstName": "Mary",  
    "Address": {  
        "Street": "123 Main",  
        "City": "Anytown",  
        "State": "OH",  
        "ZIPCode": 12345  
    }  
}  
  
{  
    "PersonID": 103,  
    "LastName": "Stephens",  
    "FirstName": "Howard",  
    "Address": {  
        "Street": "123 Main",  
        "City": "London",  
        "PostalCode": "ER3 5K8"  
    },  
    "FavoriteColor": "Blue"  
}
```

Note the following about the *People* table:

- Each item in the table has a unique identifier, or primary key, that distinguishes the item from all of the others in the table. In the *People* table, the primary key consists of one attribute (*PersonID*).
- Other than the primary key, the *People* table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand. Each item can have its own distinct attributes.
- Most of the attributes are *scalar*, which means that they can have only one value. Strings and numbers are common examples of scalars.

- Some of the items have a nested attribute (*Address*). DynamoDB supports nested attributes up to 32 levels deep.

The following is another example table named *Music* that you could use to keep track of your music collection.

```
Music

{
    "Artist": "No One You Know",
    "SongTitle": "My Dog Spot",
    "AlbumTitle": "Hey Now",
    "Price": 1.98,
    "Genre": "Country",
    "CriticRating": 8.4
}

{
    "Artist": "No One You Know",
    "SongTitle": "Somewhere Down The Road",
    "AlbumTitle": "Somewhat Famous",
    "Genre": "Country",
    "CriticRating": 8.4,
    "Year": 1984
}

{
    "Artist": "The Acme Band",
    "SongTitle": "Still in Love",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 2.47,
    "Genre": "Rock",
    "PromotionInfo": {
        "RadioStationsPlaying": [
            "KHCR",
            "KQBX",
            "WTNR",
            "WJJH"
        ],
        "TourDates": {
            "Seattle": "20150622",
            "London": "20150703"
        }
    }
}
```

```
        "Cleveland": "20150630"
    },
    "Rotation": "Heavy"
}
}

{
    "Artist": "The Acme Band",
    "SongTitle": "Look Out, World",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 0.99,
    "Genre": "Rock"
}
```

Note the following about the *Music* table:

- The primary key for *Music* consists of two attributes (*Artist* and *SongTitle*). Each item in the table must have these two attributes. The combination of *Artist* and *SongTitle* distinguishes each item in the table from all of the others.
- Other than the primary key, the *Music* table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand. Each item can have its own distinct attributes.
- One of the items has a nested attribute (*PromotionInfo*), which contains other nested attributes. DynamoDB supports nested attributes up to 32 levels deep.

For more information, see [Working with tables and data in DynamoDB](#).

Primary key

When you create a table, in addition to the table name, you must specify the primary key of the table. The primary key uniquely identifies each item in the table, so that no two items can have the same key.

DynamoDB supports two different kinds of primary keys:

- **Partition key** – A simple primary key, composed of one attribute known as the *partition key*.

DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.

In a table that has only a partition key, no two items can have the same partition key value.

The *People* table described in [Tables, items, and attributes](#) is an example of a table with a simple primary key (*PersonID*). You can access any item in the *People* table directly by providing the *PersonId* value for that item.

- **Partition key and sort key** – Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*.

DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored. All items with the same partition key value are stored together, in sorted order by sort key value.

In a table that has a partition key and a sort key, it's possible for multiple items to have the same partition key value. However, those items must have different sort key values.

The *Music* table described in [Tables, items, and attributes](#) is an example of a table with a composite primary key (*Artist* and *SongTitle*). You can access any item in the *Music* table directly, if you provide the *Artist* and *SongTitle* values for that item.

A composite primary key gives you additional flexibility when querying data. For example, if you provide only the value for *Artist*, DynamoDB retrieves all of the songs by that artist. To retrieve only a subset of songs by a particular artist, you can provide a value for *Artist* along with a range of values for *SongTitle*.

Note

The partition key of an item is also known as its *hash attribute*. The term *hash attribute* derives from the use of an internal hash function in DynamoDB that evenly distributes data items across partitions, based on their partition key values.

The sort key of an item is also known as its *range attribute*. The term *range attribute* derives from the way DynamoDB stores items with the same partition key physically close together, in sorted order by the sort key value.

Each primary key attribute must be a scalar (meaning that it can hold only a single value). The only data types allowed for primary key attributes are string, number, or binary. There are no such restrictions for other, non-key attributes.

Secondary indexes

You can create one or more secondary indexes on a table. A *secondary index* lets you query the data in the table using an alternate key, in addition to queries against the primary key. DynamoDB doesn't require that you use indexes, but they give your applications more flexibility when querying your data. After you create a secondary index on a table, you can read data from the index in much the same way as you do from the table.

DynamoDB supports two kinds of indexes:

- Global secondary index – An index with a partition key and sort key that can be different from those on the table.
- Local secondary index – An index that has the same partition key as the table, but a different sort key.

In DynamoDB, global secondary indexes (GSIs) are indexes that span the entire table, allowing you to query across all partition keys. Local secondary indexes (LSIs) are indexes that have the same partition key as the base table but a different sort key.

Each table in DynamoDB has a quota of 20 global secondary indexes (default quota) and 5 local secondary indexes.

In the example *Music* table shown previously, you can query data items by *Artist* (partition key) or by *Artist* and *SongTitle* (partition key and sort key). What if you also wanted to query the data by *Genre* and *AlbumTitle*? To do this, you could create an index on *Genre* and *AlbumTitle*, and then query the index in much the same way as you'd query the *Music* table.

The following diagram shows the example *Music* table, with a new index called *GenreAlbumTitle*. In the index, *Genre* is the partition key and *AlbumTitle* is the sort key.



Music Table	GenreAlbumTitle
<pre>"SongTitle": "My Dog Spot", "AlbumTitle": "Hey Now", "Price": 1.98, "Genre": "Country", "CriticRating": 8.4 }</pre>	<pre>"AlbumTitle": "Hey Now", "Artist": "No One You Know", "SongTitle": "My Dog Spot" }</pre>
<pre>{ "Artist": "No One You Know", "SongTitle": "Somewhere Down The Road", "AlbumTitle": "Somewhat Famous", "Genre": "Country", "CriticRating": 8.4, "Year": 1984 }</pre>	<pre>{ "Genre": "Country", "AlbumTitle": "Somewhat Famous", "Artist": "No One You Know", "SongTitle": "Somewhere Down The Road" }</pre>

Music Table	<i>GenreAlbumTitle</i>
<pre>{ "Artist": "The Acme Band", "SongTitle": "Still in Love", "AlbumTitle": "The Buck Starts Here", "Price": 2.47, "Genre": "Rock", "PromotionInfo": { "RadioStationsPlaying": ["KHCR", "KQBX", "WTNR", "WJJH"], "TourDates": { "Seattle": "20150622", "Cleveland": "20150630" }, "Rotation": "Heavy" } }</pre>	<pre>{ "Genre": "Rock", "AlbumTitle": "The Buck Starts Here", "Artist": "The Acme Band", "SongTitle": "Still In Love" }</pre>
<pre>{ "Artist": "The Acme Band", "SongTitle": "Look Out, World", "AlbumTitle": "The Buck Starts Here", "Price": 0.99, "Genre": "Rock" }</pre>	<pre>{ "Genre": "Rock", "AlbumTitle": "The Buck Starts Here", "Artist": "The Acme Band", "SongTitle": "Look Out, World" }</pre>

Note the following about the *GenreAlbumTitle* index:

- Every index belongs to a table, which is called the *base table* for the index. In the preceding example, *Music* is the base table for the *GenreAlbumTitle* index.
- DynamoDB maintains indexes automatically. When you add, update, or delete an item in the base table, DynamoDB adds, updates, or deletes the corresponding item in any indexes that belong to that table.
- When you create an index, you specify which attributes will be copied, or *projected*, from the base table to the index. At a minimum, DynamoDB projects the key attributes from the base table into the index. This is the case with *GenreAlbumTitle*, where only the key attributes from the *Music* table are projected into the index.

You can query the *GenreAlbumTitle* index to find all albums of a particular genre (for example, all *Rock* albums). You can also query the index to find all albums within a particular genre that have certain album titles (for example, all *Country* albums with titles that start with the letter H).

For more information, see [Improving data access with secondary indexes](#).

DynamoDB Streams

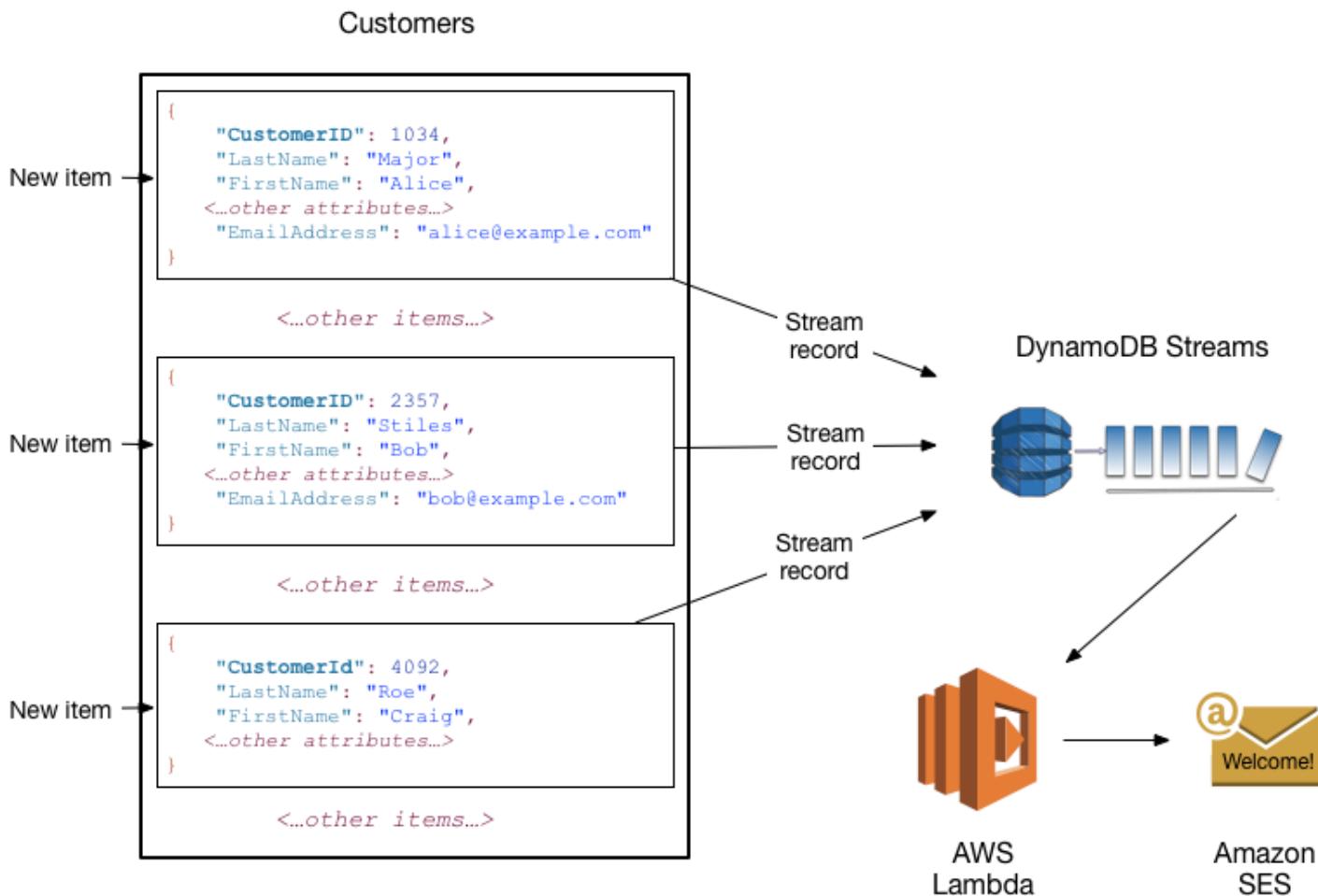
DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near-real time, and in the order that the events occurred.

Each event is represented by a *stream record*. If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

- A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
- An item is updated: The stream captures the "before" and "after" image of any attributes that were modified in the item.
- An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

You can use DynamoDB Streams together with AWS Lambda to create a *trigger*—code that runs automatically whenever an event of interest appears in a stream. For example, consider a *Customers* table that contains customer information for a company. Suppose that you want to send a "welcome" email to each new customer. You could enable a stream on that table, and then associate the stream with a Lambda function. The Lambda function would run whenever a new stream record appears, but only process new items added to the *Customers* table. For any item that has an *EmailAddress* attribute, the Lambda function would invoke Amazon Simple Email Service (Amazon SES) to send an email to that address.



Note

In this example, the last customer, Craig Roe, will not receive an email because he doesn't have an *EmailAddress*.

In addition to triggers, DynamoDB Streams enables powerful solutions such as data replication within and across AWS Regions, materialized views of data in DynamoDB tables, data analysis using Kinesis materialized views, and much more.

For more information, see [Change data capture for DynamoDB Streams](#).

DynamoDB API

To work with Amazon DynamoDB, your application must use a few simple API operations. The following is a summary of these operations, organized by category.

Note

For a full list of the API operations, see the [Amazon DynamoDB API Reference](#).

Topics

- [Control plane](#)
- [Data plane](#)
- [DynamoDB Streams](#)
- [Transactions](#)

Control plane

Control plane operations let you create and manage DynamoDB tables. They also let you work with indexes, streams, and other objects that are dependent on tables.

- [CreateTable](#) – Creates a new table. Optionally, you can create one or more secondary indexes, and enable DynamoDB Streams for the table.
- [DescribeTable](#) – Returns information about a table, such as its primary key schema, throughput settings, and index information.
- [ListTables](#) – Returns the names of all of your tables in a list.
- [UpdateTable](#) – Modifies the settings of a table or its indexes, creates or removes new indexes on a table, or modifies DynamoDB Streams settings for a table.
- [DeleteTable](#) – Removes a table and all of its dependent objects from DynamoDB.

Data plane

Data plane operations let you perform create, read, update, and delete (also called *CRUD*) actions on data in a table. Some of the data plane operations also let you read data from a secondary index.

You can use [PartiQL - a SQL-compatible query language for Amazon DynamoDB](#), to perform these CRUD operations or you can use DynamoDB's classic CRUD APIs that separates each operation into a distinct API call.

PartiQL - A SQL-compatible query language

- `ExecuteStatement` – Reads multiple items from a table. You can also write or update a single item from a table. When writing or updating a single item, you must specify the primary key attributes.
- `BatchExecuteStatement` – Writes, updates or reads multiple items from a table. This is more efficient than `ExecuteStatement` because your application only needs a single network round trip to write or read the items.

Classic APIs

Creating data

- `PutItem` – Writes a single item to a table. You must specify the primary key attributes, but you don't have to specify other attributes.
- `BatchWriteItem` – Writes up to 25 items to a table. This is more efficient than calling `PutItem` multiple times because your application only needs a single network round trip to write the items.

Reading data

- `GetItem` – Retrieves a single item from a table. You must specify the primary key for the item that you want. You can retrieve the entire item, or just a subset of its attributes.
- `BatchGetItem` – Retrieves up to 100 items from one or more tables. This is more efficient than calling `GetItem` multiple times because your application only needs a single network round trip to read the items.
- `Query` – Retrieves all items that have a specific partition key. You must specify the partition key value. You can retrieve entire items, or just a subset of their attributes. Optionally, you can apply

a condition to the sort key values so that you only retrieve a subset of the data that has the same partition key. You can use this operation on a table, provided that the table has both a partition key and a sort key. You can also use this operation on an index, provided that the index has both a partition key and a sort key.

- **Scan** – Retrieves all items in the specified table or index. You can retrieve entire items, or just a subset of their attributes. Optionally, you can apply a filtering condition to return only the values that you are interested in and discard the rest.

Updating data

- **UpdateItem** – Modifies one or more attributes in an item. You must specify the primary key for the item that you want to modify. You can add new attributes and modify or remove existing attributes. You can also perform conditional updates, so that the update is only successful when a user-defined condition is met. Optionally, you can implement an atomic counter, which increments or decrements a numeric attribute without interfering with other write requests.

Deleting data

- **DeleteItem** – Deletes a single item from a table. You must specify the primary key for the item that you want to delete.
- **BatchWriteItem** – Deletes up to 25 items from one or more tables. This is more efficient than calling `DeleteItem` multiple times because your application only needs a single network round trip to delete the items.

 **Note**

You can use `BatchWriteItem` to both create data and delete data.

DynamoDB Streams

DynamoDB Streams operations let you enable or disable a stream on a table, and allow access to the data modification records contained in a stream.

- **ListStreams** – Returns a list of all your streams, or just the stream for a specific table.
- **DescribeStream** – Returns information about a stream, such as its Amazon Resource Name (ARN) and where your application can begin reading the first few stream records.

- `GetShardIterator` – Returns a *shard iterator*, which is a data structure that your application uses to retrieve the records from the stream.
- `GetRecords` – Retrieves one or more stream records, using a given shard iterator.

Transactions

Transactions provide atomicity, consistency, isolation, and durability (ACID) enabling you to maintain data correctness in your applications more easily.

You can use [PartiQL - a SQL-compatible query language for Amazon DynamoDB](#), to perform transactional operations or you can use DynamoDB's classic CRUD APIs that separates each operation into a distinct API call.

PartiQL - A SQL-compatible query language

- `ExecuteTransaction` – A batch operation that allows CRUD operations to multiple items both within and across tables with a guaranteed all-or-nothing result.

Classic APIs

- `TransactWriteItems` – A batch operation that allows Put, Update, and Delete operations to multiple items both within and across tables with a guaranteed all-or-nothing result.
- `TransactGetItems` – A batch operation that allows Get operations to retrieve multiple items from one or more tables.

Supported data types and naming rules in Amazon DynamoDB

This section describes the Amazon DynamoDB naming rules and the various data types that DynamoDB supports. There are limits that apply to data types. For more information, see [Data types](#).

Topics

- [Naming rules](#)
- [Data types](#)
- [Data type descriptors](#)

Naming rules

Tables, attributes, and other objects in DynamoDB must have names. Names should be meaningful and concise—for example, names such as *Products*, *Books*, and *Authors* are self-explanatory.

The following are the naming rules for DynamoDB:

- All names must be encoded using UTF-8, and are case-sensitive.
- Table names and index names must be between 3 and 255 characters long, and can contain only the following characters:
 - a-z
 - A-Z
 - 0-9
 - _ (underscore)
 - - (dash)
 - . (dot)
- Attribute names must be at least one character long and less than 64 KB in size. It is considered best practice to keep your attribute names as short as possible. This helps reduce read request units consumed, as attribute names are included in metering of storage and throughput usage.

The following are the exceptions. These attribute names must be no greater than 255 characters long:

- Secondary index partition key names
- Secondary index sort key names
- The names of any user-specified projected attributes (applicable only to local secondary indexes)

Reserved words and special characters

DynamoDB has a list of reserved words and special characters. For a complete list, see [Reserved words in DynamoDB](#). Also, the following characters have special meaning in DynamoDB: # (hash) and : (colon).

Although DynamoDB allows you to use these reserved words and special characters for names, we recommend that you avoid doing so because you have to define placeholder variables whenever

you use these names in an expression. For more information, see [Expression attribute names in DynamoDB](#).

Data types

DynamoDB supports many different data types for attributes within a table. They can be categorized as follows:

- **Scalar Types** – A scalar type can represent exactly one value. The scalar types are number, string, binary, Boolean, and null.
 - **Document Types** – A document type can represent a complex structure with nested attributes, such as what you would find in a JSON document. The document types are list and map.
 - **Set Types** – A set type can represent multiple scalar values. The set types are string set, number set, and binary set.

When you create a table or a secondary index, you must specify the names and data types of each primary key attribute (partition key and sort key). Furthermore, each primary key attribute must be defined as type string, number, or binary.

DynamoDB is a NoSQL database and is *schemaless*. This means that other than the primary key attributes, you don't have to define any attributes or data types when you create tables. By comparison, relational databases require you to define the names and data types of each column when you create a table.

The following are descriptions of each data type, along with examples in JSON format.

Scalar types

The scalar types are number, string, binary, Boolean, and null.

Number

Numbers can be positive, negative, or zero. Numbers can have up to 38 digits of precision. Exceeding this results in an exception. If you need greater precision than 38 digits, you can use strings.

In DynamoDB, numbers are represented as variable length. Leading and trailing zeroes are trimmed.

All numbers are sent across the network to DynamoDB as strings to maximize compatibility across languages and libraries. However, DynamoDB treats them as number type attributes for mathematical operations.

You can use the number data type to represent a date or a timestamp. One way to do this is by using epoch time—the number of seconds since 00:00:00 UTC on 1 January 1970. For example, the epoch time 1437136300 represents 12:31:40 PM UTC on 17 July 2015.

For more information, see http://en.wikipedia.org/wiki/Unix_time.

String

Strings are Unicode with UTF-8 binary encoding. The minimum length of a string can be zero, if the attribute is not used as a key for an index or table, and is constrained by the maximum DynamoDB item size limit of 400 KB.

The following additional constraints apply to primary key attributes that are defined as type string:

- For a simple primary key, the maximum length of the first attribute value (the partition key) is 2048 bytes.
- For a composite primary key, the maximum length of the second attribute value (the sort key) is 1024 bytes.

DynamoDB collates and compares strings using the bytes of the underlying UTF-8 string encoding. For example, "a" (0x61) is greater than "A" (0x41), and "ż" (0xC2BF) is greater than "z" (0x7A).

You can use the string data type to represent a date or a timestamp. One way to do this is by using ISO 8601 strings, as shown in these examples:

- 2016-02-15
- 2015-12-21T17:42:34Z
- 20150311T122706Z

For more information, see http://en.wikipedia.org/wiki/ISO_8601.

Note

Unlike conventional relational databases, DynamoDB does not natively support a date and time data type. It can be useful instead to store date and time data as a number data type, using Unix epoch time.

Binary

Binary type attributes can store any binary data, such as compressed text, encrypted data, or images. Whenever DynamoDB compares binary values, it treats each byte of the binary data as unsigned.

The length of a binary attribute can be zero, if the attribute is not used as a key for an index or table, and is constrained by the maximum DynamoDB item size limit of 400 KB.

If you define a primary key attribute as a binary type attribute, the following additional constraints apply:

- For a simple primary key, the maximum length of the first attribute value (the partition key) is 2048 bytes.
- For a composite primary key, the maximum length of the second attribute value (the sort key) is 1024 bytes.

Your applications must encode binary values in base64-encoded format before sending them to DynamoDB. Upon receipt of these values, DynamoDB decodes the data into an unsigned byte array and uses that as the length of the binary attribute.

The following example is a binary attribute, using base64-encoded text.

```
dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbmNvZGVk
```

Boolean

A Boolean type attribute can store either `true` or `false`.

Null

Null represents an attribute with an unknown or undefined state.

Document types

The document types are list and map. These data types can be nested within each other, to represent complex data structures up to 32 levels deep.

There is no limit on the number of values in a list or a map, as long as the item containing the values fits within the DynamoDB item size limit (400 KB).

An attribute value can be an empty string or empty binary value if the attribute is not used for a table or index key. An attribute value cannot be an empty set (string set, number set, or binary set), however, empty lists and maps are allowed. Empty string and binary values are allowed within lists and maps. For more information, see [Attributes](#).

List

A list type attribute can store an ordered collection of values. Lists are enclosed in square brackets:

[...]

A list is similar to a JSON array. There are no restrictions on the data types that can be stored in a list element, and the elements in a list element do not have to be of the same type.

The following example shows a list that contains two strings and a number.

```
FavoriteThings: ["Cookies", "Coffee", 3.14159]
```

Note

DynamoDB lets you work with individual elements within lists, even if those elements are deeply nested. For more information, see [Using expressions in DynamoDB](#).

Map

A map type attribute can store an unordered collection of name-value pairs. Maps are enclosed in curly braces: { ... }

A map is similar to a JSON object. There are no restrictions on the data types that can be stored in a map element, and the elements in a map do not have to be of the same type.

Maps are ideal for storing JSON documents in DynamoDB. The following example shows a map that contains a string, a number, and a nested list that contains another map.

```
{  
    Day: "Monday",  
    UnreadEmails: 42,  
    ItemsOnMyDesk: [  
        "Coffee Cup",  
        "Telephone",  
        {  
            Pens: { Quantity : 3},  
            Pencils: { Quantity : 2},  
            Erasers: { Quantity : 1}  
        }  
    ]  
}
```

Note

DynamoDB lets you work with individual elements within maps, even if those elements are deeply nested. For more information, see [Using expressions in DynamoDB](#).

Sets

DynamoDB supports types that represent sets of number, string, or binary values. All the elements within a set must be of the same type. For example, a Number Set can only contain numbers and a String Set can only contain strings.

There is no limit on the number of values in a set, as long as the item containing the values fits within the DynamoDB item size limit (400 KB).

Each value within a set must be unique. The order of the values within a set is not preserved. Therefore, your applications must not rely on any particular order of elements within the set. DynamoDB does not support empty sets, however, empty string and binary values are allowed within a set.

The following example shows a string set, a number set, and a binary set:

```
["Black", "Green", "Red"]  
  
[42.2, -19, 7.5, 3.14]  
  
["U3Vubnk=", "UmFpbnk=", "U25vd3k="]
```

Data type descriptors

The low-level DynamoDB API protocol uses *Data type descriptors* as tokens that tell DynamoDB how to interpret each attribute.

The following is a complete list of DynamoDB data type descriptors:

- **S** – String
- **N** – Number
- **B** – Binary
- **BOOL** – Boolean
- **NULL** – Null
- **M** – Map
- **L** – List
- **SS** – String Set
- **NS** – Number Set
- **BS** – Binary Set

Read consistency

Amazon DynamoDB reads data from tables, local secondary indexes (LSIs), global secondary indexes (GSIs), and streams. For more information, see [Core components of Amazon DynamoDB](#). Both tables and LSIs provide two read consistency options: *eventually consistent* (default) and *strongly consistent* reads. All reads from GSIs and streams are eventually consistent.

When your application writes data to a DynamoDB table and receives an HTTP 200 response (OK), that means the write completed successfully and has been durably persisted. DynamoDB provides *read-committed* isolation and ensures that read operations always return committed values for an item. The read will never present a view to the item from a write which did not ultimately succeed. Read-committed isolation does not prevent modifications of the item immediately after the read operation.

Eventually Consistent Reads

Eventually consistent is the default read consistent model for all read operations. When issuing eventually consistent reads to a DynamoDB table or an index, the responses may not reflect the

results of a recently completed write operation. If you repeat your read request after a short time, the response should eventually return the more recent item. Eventually consistent reads are supported on tables, local secondary indexes, and global secondary indexes. Also note that all reads from a DynamoDB stream are also eventually consistent.

Eventually consistent reads are half the cost of strongly consistent reads. For more information, see [Amazon DynamoDB pricing](#).

Strongly Consistent Reads

Read operations such as `GetItem`, `Query`, and `Scan` provide an optional `ConsistentRead` parameter. If you set `ConsistentRead` to true, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. Strongly consistent reads are only supported on tables and local secondary indexes. Strongly consistent reads from a global secondary index or a DynamoDB stream are not supported.

Global tables read consistency

DynamoDB also supports [global tables](#) for multi-active and multi-Region replication. A global table is composed of multiple replica tables in different AWS Regions. Any change made to any item in any replica table is replicated to all the other replicas within the same global table, typically within a second, and are eventually consistent. For more information, see [Consistency and conflict resolution](#).

Read/write capacity mode

Amazon DynamoDB has two read/write capacity modes for processing reads and writes on your tables:

- On-demand
- Provisioned (default, free-tier eligible)

The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity. You can set the read/write capacity mode when creating a table or you can change it later.

Secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations when changing read/write Capacity Mode](#).

The following video will give you an introductory look at table capacity modes.

[Table capacity modes](#)

For more information on best practices for optimizing costs of your DynamoDB tables, see [Optimizing costs on DynamoDB tables](#).

Topics

- [On-demand mode](#)
- [Provisioned mode](#)

On-demand mode

Amazon DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use.

When you choose on-demand mode, DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level hits a new peak, DynamoDB adapts rapidly to accommodate the workload. Tables that use on-demand mode deliver the same single-digit millisecond latency, service-level agreement (SLA) commitment, and security that DynamoDB already offers. You can choose on-demand for both new and existing tables and you can continue using the existing DynamoDB APIs without changing code.

On-demand mode is a good option if any of the following are true:

- You create new tables with unknown workloads.
- You have unpredictable application traffic.
- You prefer the ease of paying for only what you use.

The request rate is only limited by the DynamoDB throughput default table quotas, but it can be raised upon request. For more information, see [Throughput default quotas](#).

To get started with on-demand, you can create or update a table to use on-demand mode. For more information, see [Basic operations on DynamoDB tables](#).

Tables can be switched to on-demand mode once every 24 hours. Creating a table as on-demand also starts this 24-hour period. Tables can be returned to provisioned capacity mode at any time. For issues to consider when switching read/write capacity modes, see [Considerations when changing read/write Capacity Mode](#).

Topics

- [Read request units and write request units](#)
- [Peak traffic and scaling properties](#)
- [Initial throughput for on-demand capacity mode](#)
- [Table behavior while switching read/write capacity mode](#)
- [Pre-warming a table for on-demand capacity mode](#)

Read request units and write request units

For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform. DynamoDB charges you for the reads and writes that your application performs on your tables in terms of read request units and write request units.

DynamoDB read requests can be either strongly consistent, eventually consistent, or transactional.

- A *strongly consistent* read request of an item up to 4 KB requires one read request unit.
- An *eventually consistent* read request of an item up to 4 KB requires one-half read request unit.
- A *transactional* read request of an item up to 4 KB requires two read request units.

If you need to read an item that is larger than 4 KB, DynamoDB needs additional read request units. The total number of read request units required depends on the item size, and whether you want an eventually consistent or strongly consistent read. For example, if your item size is 8 KB, you require 2 read request units to sustain one strongly consistent read, 1 read request unit if you choose eventually consistent reads, or 4 read request units for a transactional read request.

To learn more about DynamoDB read consistency models, see [Read consistency](#).

Important

If you perform a read operation on an item that *does not exist*, DynamoDB will still consume read throughput as outlined above.

One *write request unit* represents one write for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB needs to consume additional write request units. Transactional write requests require 2 write request units to perform one write for items up to 1

KB. The total number of write request units required depends on the item size. For example, if your item size is 2 KB, you require 2 write request units to sustain one write request or 4 write request units for a transactional write request.

For detailed pricing examples and to estimate costs using the pricing calculator, see [Amazon DynamoDB Pricing](#).

Peak traffic and scaling properties

DynamoDB tables using on-demand capacity mode automatically adapt to your application's traffic volume. On-demand capacity mode instantly accommodates up to double the previous peak traffic on a table. For example, if your application's traffic pattern varies between 25,000 and 50,000 strongly consistent reads per second where 50,000 reads per second is the previous traffic peak, on-demand capacity mode instantly accommodates sustained traffic of up to 100,000 reads per second. If your application sustains traffic of 100,000 reads per second, that peak becomes your new previous peak, enabling subsequent traffic to reach up to 200,000 reads per second.

If you need more than double your previous peak on table, DynamoDB automatically allocates more capacity as your traffic volume increases to help ensure that your workload does not experience throttling. However, throttling can occur if you exceed double your previous peak within 30 minutes. For example, if your application's traffic pattern varies between 25,000 and 50,000 strongly consistent reads per second where 50,000 reads per second is the previously reached traffic peak, DynamoDB recommends spacing your traffic growth over at least 30 minutes before driving more than 100,000 reads per second.

Initial throughput for on-demand capacity mode

If you recently switched an existing table to on-demand capacity mode for the first time, or if you created a new table with on-demand capacity mode enabled, the table has the following previous peak settings, even though the table has not served traffic previously using on-demand capacity mode:

Following are examples of possible scenarios.

- **A provisioned table configured as 100 WCU and 100 RCU.** When this table is switched to on-demand for the first time, DynamoDB will ensure it is scaled out to instantly sustain at least 4,000 write units/sec and 12,000 read units/sec.
- **A provisioned table configured as 8,000 WCU and 24,000 RCU.** When this table is switched to on-demand, it will continue to be able to sustain at least 8,000 write units/sec and 24,000 read units/sec at any time.

- **A provisioned table configured with 8,000 WCU and 24,000 RCU, that consumed 6,000 write units/sec and 18,000 read units/sec for a sustained period.** When this table is switched to on-demand, it will continue to be able to sustain at least 8,000 write units/sec and 24,000 read units/sec. The previous traffic may further allow the table to sustain much higher levels of traffic without throttling.
- **A table previously provisioned with 10,000 WCU and 10,000 RCU, but currently provisioned with 10 RCU and 10 WCU.** When this table is switched to on-demand, it will be able to sustain at least 10,000 write units/sec and 10,000 read units/sec.

Table behavior while switching read/write capacity mode

When you switch a table from provisioned capacity mode to on-demand capacity mode, DynamoDB makes several changes to the structure of your table and partitions. This process can take several minutes. During the switching period, your table delivers throughput that is consistent with the previously provisioned write capacity unit and read capacity unit amounts. When switching from on-demand capacity mode back to provisioned capacity mode, your table delivers throughput consistent with the previous peak reached when the table was set to on-demand capacity mode.

Pre-warming a table for on-demand capacity mode

With on-demand capacity mode, the requests can burst up to double the previous peak on the table. Note that throttling can occur if the requests spikes to more than double the default capacity or the previously achieved peak request rate within 30 minutes. One solution is to pre-warm the tables to the anticipated peak capacity of the spike.

To pre-warm the table, follow these steps:

1. Make sure to check your account limits and confirm that you can reach the desired capacity in provisioned mode.
2. If you're pre-warming a table that already exists, or a new table in on-demand mode, start this process at least 24 hours before the anticipated peak. You can only switch between on-demand and provisioned mode once per 24 hours.
3. To pre-warm a table that's currently in on-demand mode, switch it to provisioned mode and wait until the table is active. Then go to the next step.

If you want to pre-warm a new table that's in provisioned mode, or has already been in provisioned mode for 24 hours, you can proceed to the next step without waiting.

4. Set the table's write throughput to the desired peak value, and keep it there for several minutes. You will incur cost from this high volume of throughput until you switch back to on-demand.
5. Switch to On-Demand capacity mode. This should sustain the provisioned throughput capacity values.

Provisioned mode

If you choose provisioned mode, you specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes. This helps you govern your DynamoDB use to stay at or below a defined request rate in order to obtain cost predictability.

Provisioned mode is a good option if any of the following are true:

- You have predictable application traffic.
- You run applications whose traffic is consistent or ramps gradually.
- You can forecast capacity requirements to control costs.

Read capacity units and write capacity units

For provisioned mode tables, you specify throughput capacity in terms of read capacity units (RCUs) and write capacity units (WCUs):

- One *read capacity unit* represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. Transactional read requests require two read capacity units to perform one read per second for items up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB must consume additional read capacity units. The total number of read capacity units required depends on the item size, and whether you want an eventually consistent or strongly consistent read. For example, if your item size is 8 KB, you require 2 read capacity units to sustain one strongly consistent read per second, 1 read capacity unit if you choose eventually consistent reads, or 4 read capacity units for a transactional read request. For more information, see [Capacity unit consumption for reads](#).

 **Note**

To learn more about DynamoDB read consistency models, see [Read consistency](#).

- One *write capacity unit* represents one write per second for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB must consume additional write capacity units. Transactional write requests require 2 write capacity units to perform one write per second for items up to 1 KB. The total number of write capacity units required depends on the item size. For example, if your item size is 2 KB, you require 2 write capacity units to sustain one write request per second or 4 write capacity units for a transactional write request. For more information, see [Capacity unit consumption for writes](#).

 **Important**

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

If your application reads or writes larger items (up to the DynamoDB maximum item size of 400 KB), it will consume more capacity units.

For example, suppose that you create a provisioned table with 6 read capacity units and 6 write capacity units. With these settings, your application could do the following:

- Perform strongly consistent reads of up to 24 KB per second ($4 \text{ KB} \times 6 \text{ read capacity units}$).
- Perform eventually consistent reads of up to 48 KB per second (twice as much read throughput).
- Perform transactional read requests of up to 12 KB per second.
- Write up to 6 KB per second ($1 \text{ KB} \times 6 \text{ write capacity units}$).
- Perform transactional write requests of up to 3 KB per second.

For more information, see [Managing settings on DynamoDB provisioned capacity tables](#).

Provisioned throughput is the maximum amount of capacity that an application can consume from a table or index. If your application exceeds your provisioned throughput capacity on a table or index, it is subject to request throttling.

Throttling prevents your application from consuming too many capacity units.

When a request is throttled, it fails with an HTTP 400 code (Bad Request) and a `ProvisionedThroughputExceededException`. The AWS SDKs have built-in support for retrying throttled requests (see [Error retries and exponential backoff](#)), so you do not need to write

this logic yourself. For more information on how to resolve throttling issues, see [Why is my Amazon DynamoDB table being throttled?](#)

You can use the AWS Management Console to monitor your provisioned and actual throughput, and to modify your throughput settings if necessary.

For detailed pricing examples and to estimate costs using the pricing calculator, see [Amazon DynamoDB Pricing](#).

DynamoDB auto scaling

DynamoDB auto scaling actively manages throughput capacity for tables and global secondary indexes. With auto scaling, you define a range (upper and lower limits) for read and write capacity units. You also define a target utilization percentage within that range. DynamoDB auto scaling seeks to maintain your target utilization, even as your application workload increases or decreases.

With DynamoDB auto scaling, a table or a global secondary index can increase its provisioned read and write capacity to handle sudden increases in traffic, without request throttling. When the workload decreases, DynamoDB auto scaling can decrease the throughput so that you don't pay for unused provisioned capacity.

Note

If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default.

You can manage auto scaling settings at any time by using the console, the AWS CLI, or one of the AWS SDKs.

For more information, see [Managing throughput capacity automatically with DynamoDB auto scaling](#).

Reserved capacity

As a DynamoDB customer, you can purchase *reserved capacity* in advance for tables that use the DynamoDB Standard table class, as described at [Amazon DynamoDB Pricing](#). With reserved capacity, you pay a one-time upfront fee and commit to a minimum provisioned usage level over a period of time. Your reserved capacity is billed at the hourly reserved capacity rate. By reserving your read and write capacity units ahead of time, you realize significant cost savings on your

provisioned capacity costs. Any capacity that you provision in excess of your reserved capacity is billed at standard provisioned capacity rates.

Reserved capacity discounts are applied first to the account that purchased the reserved capacity. Any unused reserved capacity discount is then applied to other accounts in the same AWS organization as the purchasing account. You can turn off Reserved Instance discount sharing on the **Preferences** page on the Billing and Cost Management console. For more information, see [Turning off reserved instances and Savings Plans discount sharing](#).

Note

Reserved capacity is not available for replicated write capacity units. Reserved Capacity is applied only to the region in which it was purchased. Reserved capacity is also not available for tables using the DynamoDB Standard-IA table class or on-demand capacity mode.

To manage reserved capacity, go to the [DynamoDB console](#) and choose **Reserved Capacity**.

Note

You can prevent users from viewing or purchasing reserved capacity, while still allowing them to access the rest of the console. For more information, see "Grant Permissions to Prevent Purchasing of Reserved Capacity Offerings" in [Identity and Access Management for Amazon DynamoDB](#).

For more information about specific pricing see [Amazon DynamoDB Pricing](#).

Table classes

DynamoDB offers two table classes designed to help you optimize for cost. The DynamoDB Standard table class is the default, and is recommended for the vast majority of workloads. The DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) table class is optimized for tables where storage is the dominant cost. For example, tables that store infrequently accessed data, such as application logs, old social media posts, e-commerce order history, and past gaming achievements, are good candidates for the Standard-IA table class. See [Amazon DynamoDB Pricing](#) for pricing details.

Every DynamoDB table is associated with a table class (DynamoDB Standard by default). All secondary indexes associated with the table use the same table class. Each table class offers different pricing for data storage as well as for read and write requests. You can select the most cost-effective table class for your table based on its storage and throughput usage patterns.

The choice of a table class is not permanent—you can change this setting using the AWS Management Console, AWS CLI, or AWS SDK. DynamoDB also supports managing your table class using AWS CloudFormation for single-Region tables and global tables. To learn more about selecting your table class, see [Considerations when choosing a table class](#).

Partitions and data distribution

Amazon DynamoDB stores data in partitions. A *partition* is an allocation of storage for a table, backed by solid state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region. Partition management is handled entirely by DynamoDB—you never have to manage partitions yourself.

When you create a table, the initial status of the table is CREATING. During this phase, DynamoDB allocates sufficient partitions to the table so that it can handle your provisioned throughput requirements. You can begin writing and reading table data after the table status changes to ACTIVE.

DynamoDB allocates additional partitions to a table in the following situations:

- If you increase the table's provisioned throughput settings beyond what the existing partitions can support.
- If an existing partition fills to capacity and more storage space is required.

Partition management occurs automatically in the background and is transparent to your applications. Your table remains available throughout and fully supports your provisioned throughput requirements.

For more details, see [Partition key design](#).

Global secondary indexes in DynamoDB are also composed of partitions. The data in a global secondary index is stored separately from the data in its base table, but index partitions behave in much the same way as table partitions.

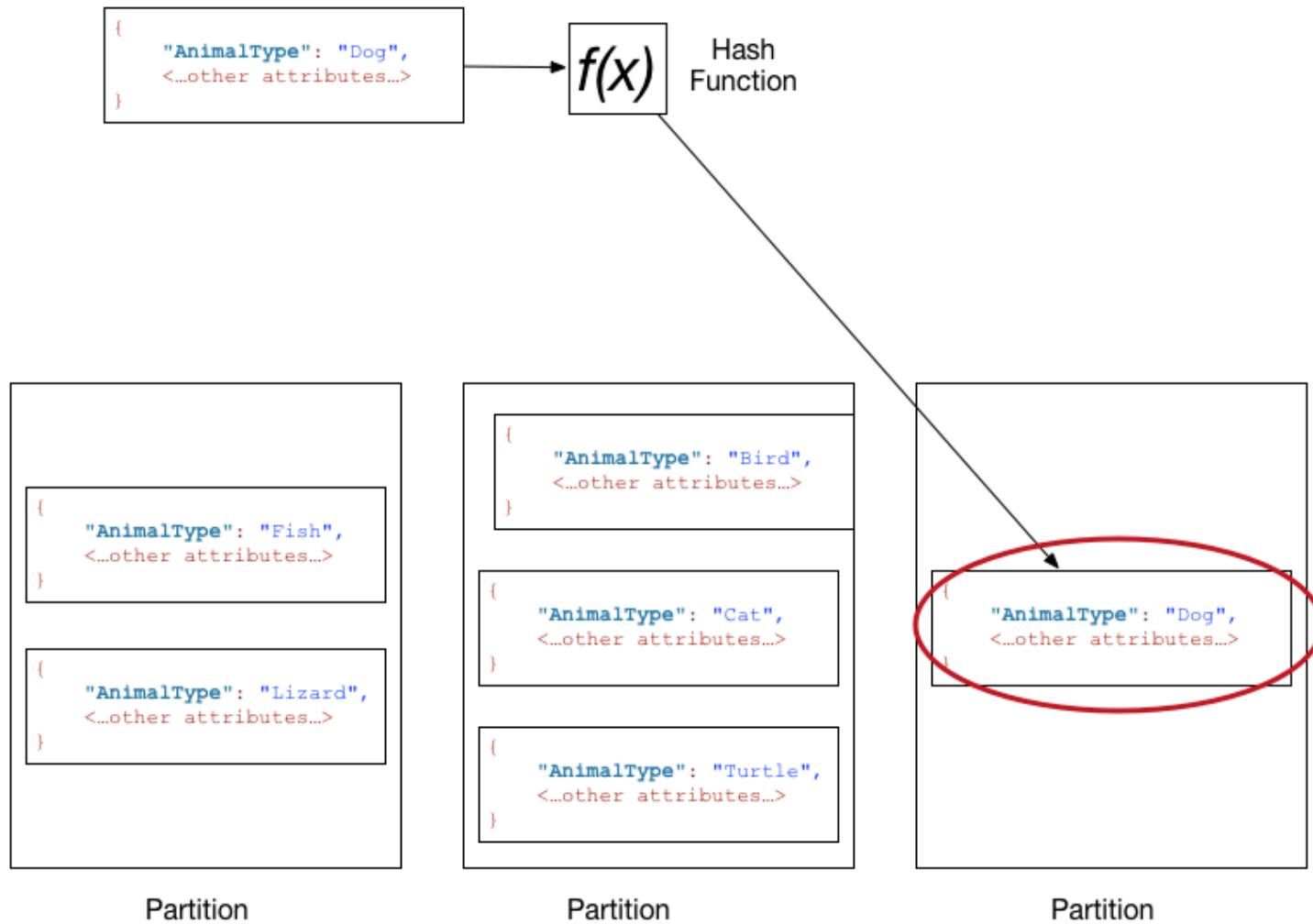
Data distribution: Partition key

If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value.

To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored.

To read an item from the table, you must specify the partition key value for the item. DynamoDB uses this value as input to its hash function, yielding the partition in which the item can be found.

The following diagram shows a table named *Pets*, which spans multiple partitions. The table's primary key is *AnimalType* (only this key attribute is shown). DynamoDB uses its hash function to determine where to store a new item, in this case based on the hash value of the string *Dog*. Note that the items are not stored in sorted order. Each item's location is determined by the hash value of its partition key.



Note

DynamoDB is optimized for uniform distribution of items across a table's partitions, no matter how many partitions there may be. We recommend that you choose a partition key that can have a large number of distinct values relative to the number of items in the table.

Data distribution: Partition key and sort key

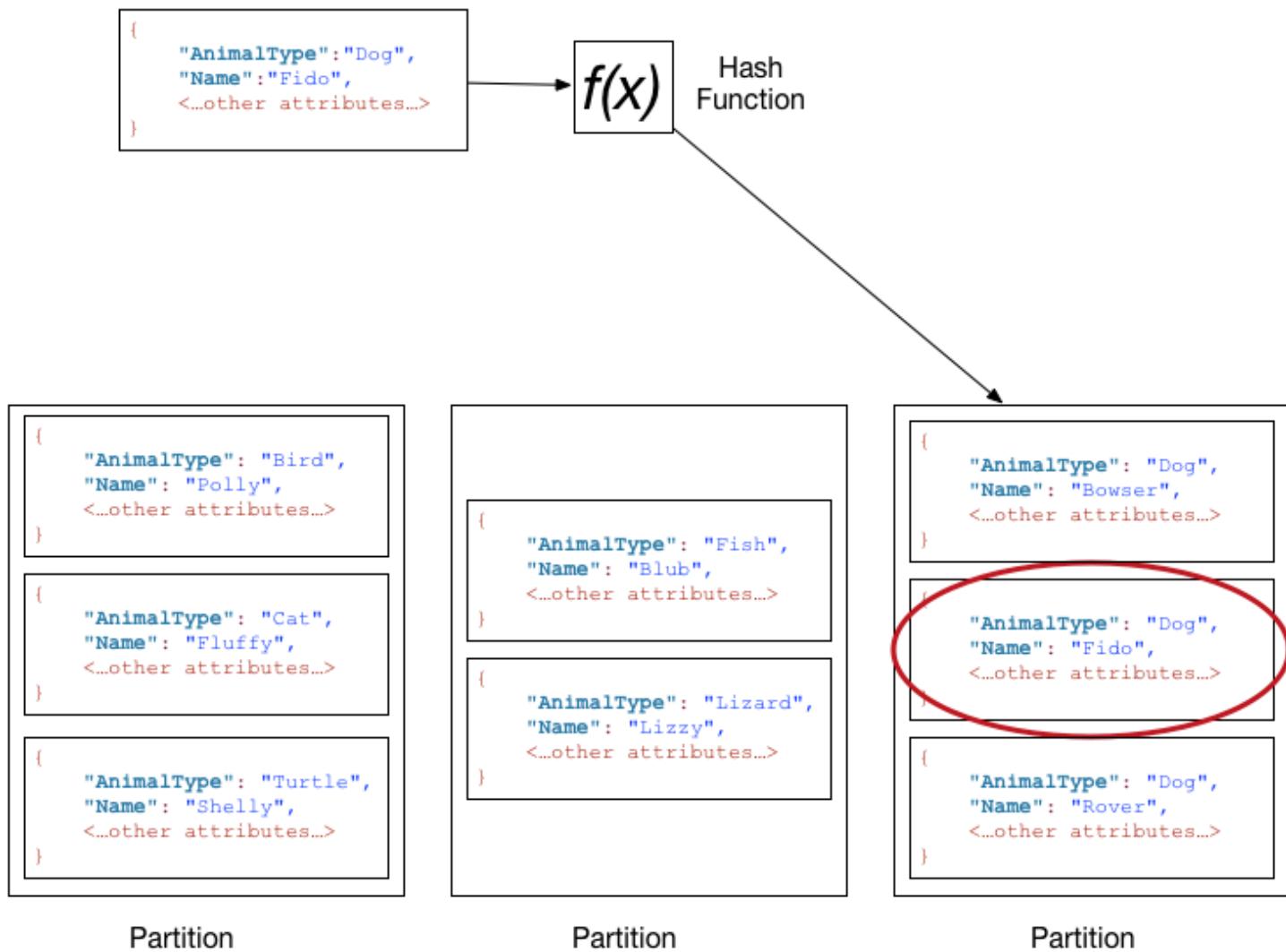
If the table has a composite primary key (partition key and sort key), DynamoDB calculates the hash value of the partition key in the same way as described in [Data distribution: Partition key](#). However, it tends to keep items which have the same value of partition key close together and in sorted order by the sort key attribute's value. The set of items which have the same value of partition key is called an item collection. Item collections are optimized for efficient retrieval of ranges of the items within the collection. If your table doesn't have local secondary indexes, DynamoDB will automatically split your item collection over as many partitions as required to store the data and to serve read and write throughput.

To write an item to the table, DynamoDB calculates the hash value of the partition key to determine which partition should contain the item. In that partition, several items could have the same partition key value. So DynamoDB stores the item among the others with the same partition key, in ascending order by sort key.

To read an item from the table, you must specify its partition key value and sort key value. DynamoDB calculates the partition key's hash value, yielding the partition in which the item can be found.

You can read multiple items from the table in a single operation (Query) if the items you want have the same partition key value. DynamoDB returns all of the items with that partition key value. Optionally, you can apply a condition to the sort key so that it returns only the items within a certain range of values.

Suppose that the *Pets* table has a composite primary key consisting of *AnimalType* (partition key) and *Name* (sort key). The following diagram shows DynamoDB writing an item with a partition key value of *Dog* and a sort key value of *Fido*.



To read that same item from the *Pets* table, DynamoDB calculates the hash value of *Dog*, yielding the partition in which these items are stored. DynamoDB then scans the sort key attribute values until it finds *Fido*.

To read all of the items with an *AnimalType* of *Dog*, you can issue a Query operation without specifying a sort key condition. By default, the items are returned in the order that they are stored (that is, in ascending order by sort key). Optionally, you can request descending order instead.

To query only some of the *Dog* items, you can apply a condition to the sort key (for example, only the *Dog* items where *Name* begins with a letter that is within the range A through K).

Note

In a DynamoDB table, there is no upper limit on the number of distinct sort key values per partition key value. If you needed to store many billions of *Dog* items in the *Pets* table, DynamoDB would allocate enough storage to handle this requirement automatically.

From SQL to NoSQL

If you are an application developer, you might have some experience using a relational database management system (RDBMS) and Structured Query Language (SQL). As you begin working with Amazon DynamoDB, you will encounter many similarities, but also many things that are different. NoSQL is a term used to describe nonrelational database systems that are highly available, scalable, and optimized for high performance. Instead of the relational model, NoSQL databases (like DynamoDB) use alternate models for data management, such as key-value pairs or document storage. For more information, see [What is NoSQL?](#).

Amazon DynamoDB supports [PartiQL](#), an open-source, SQL-compatible query language that makes it easy for you to efficiently query data, regardless of where or in what format it is stored. With PartiQL, you can easily process structured data from relational databases, semi-structured and nested data in open data formats, and even schema-less data in NoSQL or document databases that allow different attributes for different rows. For more information, see [PartiQL query language](#).

The following sections describe common database tasks, comparing and contrasting SQL statements with their equivalent DynamoDB operations.

Note

The SQL examples in this section are compatible with the MySQL RDBMS.

The DynamoDB examples in this section show the name of the DynamoDB operation, along with the parameters for that operation in JSON format. For code examples that use these operations, see [Getting started with DynamoDB and the AWS SDKs](#).

Topics

- [Relational \(SQL\) or NoSQL?](#)
- [Characteristics of databases](#)

- [Creating a table](#)
- [Getting information about a table](#)
- [Writing data to a table](#)
- [Key differences between SQL and DynamoDB when reading data from a table](#)
- [Managing indexes](#)
- [Modifying data in a table](#)
- [Deleting data from a table](#)
- [Removing a table](#)

Relational (SQL) or NoSQL?

Today's applications have more demanding requirements than ever before. For example, an online game might start out with just a few users and a very small amount of data. However, if the game becomes successful, it can easily outstrip the resources of the underlying database management system. It is common for web-based applications to have hundreds, thousands, or millions of concurrent users, with terabytes or more of new data generated per day. Databases for such applications must handle tens (or hundreds) of thousands of reads and writes per second.

Amazon DynamoDB is well-suited for these kinds of workloads. As a developer, you can start small and gradually increase your utilization as your application becomes more popular. DynamoDB scales seamlessly to handle very large amounts of data and very large numbers of users.

For more information on traditional relational database modeling and how to adapt it for DynamoDB, see [Best practices for modeling relational data in DynamoDB](#).

The following table shows some high-level differences between a relational database management system (RDBMS) and DynamoDB.

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Optimal Workloads	Ad hoc queries; data warehousing; OLAP (online analytical processing).	Web-scale applications, including social networks, gaming, media sharing, and Internet of Things (IoT).

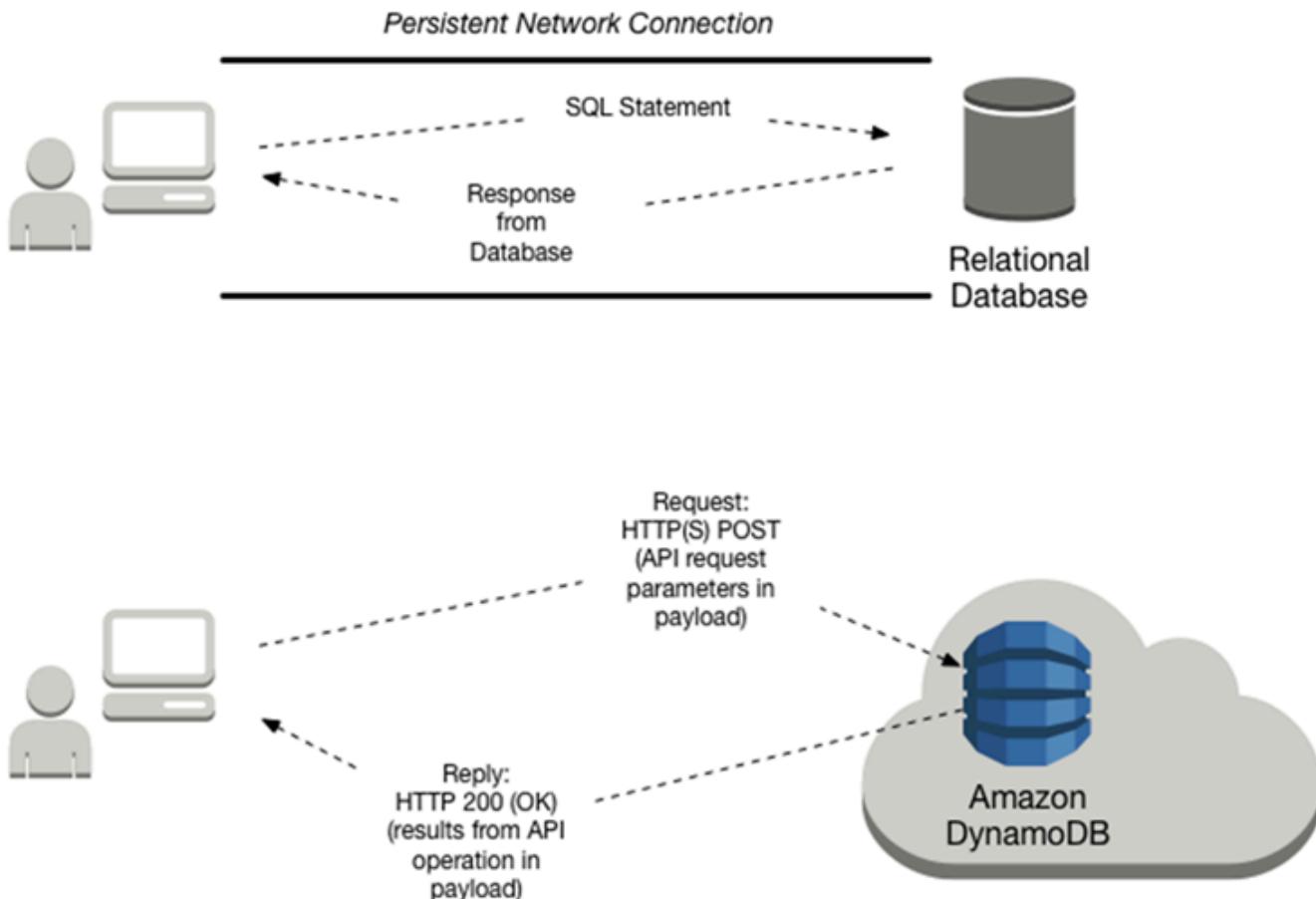
Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Data Model	<p>The relational model requires a well-defined schema, where data is normalized into tables, rows, and columns. In addition, all of the relationships are defined among tables, columns, indexes, and other database elements.</p>	<p>DynamoDB is schemaless. Every table must have a primary key to uniquely identify each data item, but there are no similar constraints on other non-key attributes. DynamoDB can manage structured or semistructured data, including JSON documents.</p>
Data Access	<p>SQL is the standard for storing and retrieving data. Relational databases offer a rich set of tools for simplifying the development of database-driven applications, but all of these tools use SQL.</p>	<p>You can use the AWS Management Console, the AWS CLI, or NoSQL WorkBench to work with DynamoDB and perform ad hoc tasks. PartiQL, a SQL-compatible query language, lets you select, insert, update, and delete data in DynamoDB. Applications can use the AWS software development kits (SDKs) to work with DynamoDB using object-based, document-centric, or low-level interfaces.</p>

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Performance	<p>Relational databases are optimized for storage, so performance generally depends on the disk subsystem. Developers and database administrators must optimize queries, indexes, and table structures in order to achieve peak performance.</p>	<p>DynamoDB is optimized for compute, so performance is mainly a function of the underlying hardware and network latency. As a managed service, DynamoDB insulates you and your applications from these implementation details, so that you can focus on designing and building robust, high-performance applications.</p>
Scaling	<p>It is easiest to scale up with faster hardware. It is also possible for database tables to span across multiple hosts in a distributed system, but this requires additional investment. Relational databases have maximum sizes for the number and size of files, which imposes upper limits on scalability.</p>	<p>DynamoDB is designed to scale out using distributed clusters of hardware. This design allows increased throughput without increased latency. Customers specify their throughput requirements, and DynamoDB allocates sufficient resources to meet those requirements. There are no upper limits on the number of items per table, nor the total size of that table.</p>

Characteristics of databases

Before your application can access a database, it must be *authenticated* to ensure that the application is allowed to use the database. It must be *authorized* so that the application can perform only the actions for which it has permissions.

The following diagram shows a client's interaction with a relational database and with Amazon DynamoDB.



The following table has more details about client interaction tasks.

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Tools for Accessing the Database	Most relational databases provide a command line	In most cases, you write application code. You can also

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
	<p>interface (CLI) so that you can enter ad hoc SQL statements and see the results immediately.</p>	<p>use the AWS Management Console, the AWS Command Line Interface (AWS CLI), or NoSQL Workbench to send ad hoc requests to DynamoDB and view the results. PartiQL, a SQL-compatible query language, lets you select, insert, update, and delete data in DynamoDB.</p>
Connecting to the Database	<p>An application program establishes and maintains a network connection with the database. When the application is finished, it terminates the connection.</p>	<p>DynamoDB is a web service, and interactions with it are stateless. Applications do not need to maintain persistent network connections. Instead, interaction with DynamoDB occurs using HTTP(S) requests and responses.</p>
Authentication	<p>An application cannot connect to the database until it is authenticated. The RDBMS can perform the authentication itself, or it can offload this task to the host operating system or a directory service.</p>	<p>Every request to DynamoDB must be accompanied by a cryptographic signature, which authenticates that particular request. The AWS SDKs provide all of the logic necessary for creating signatures and signing requests. For more information, see Signing AWS API requests in the <i>AWS General Reference</i>.</p>

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Authorization	<p>Applications can perform only the actions for which they have been authorized. Database administrators or application owners can use the SQL GRANT and REVOKE statements to control access to database objects (such as tables), data (such as rows within a table), or the ability to issue certain SQL statements.</p>	<p>In DynamoDB, authorization is handled by AWS Identity and Access Management (IAM). You can write an IAM policy to grant permissions on a DynamoDB resource (such as a table), and then allow users and roles to use that policy. IAM also features fine-grained access control for individual data items in DynamoDB tables. For more information, see Identity and Access Management for Amazon DynamoDB.</p>
Sending a Request	<p>The application issues a SQL statement for every database operation that it wants to perform. Upon receipt of the SQL statement, the RDBMS checks its syntax, creates a plan for performing the operation, and then runs the plan.</p>	<p>The application sends HTTP(S) requests to DynamoDB. The requests contain the name of the DynamoDB operation to perform, along with parameters. DynamoDB runs the request immediately.</p>
Receiving a Response	<p>The RDBMS returns the results from the SQL statement. If there is an error, the RDBMS returns an error status and message.</p>	<p>DynamoDB returns an HTTP(S) response containing the results of the operation. If there is an error, DynamoDB returns an HTTP error status and messages.</p>

Creating a table

Tables are the fundamental data structures in relational databases and in Amazon DynamoDB. A relational database management system (RDBMS) requires you to define the table's schema when you create it. In contrast, DynamoDB tables are schemaless—other than the primary key, you do not need to define any extra attributes or data types when you create a table.

The following section compares how you would create a table with SQL to how you would create it with DynamoDB.

Topics

- [Creating a table with SQL](#)
- [Creating a table with DynamoDB](#)

Creating a table with SQL

With SQL you would use the CREATE TABLE statement to create a table, as shown in the following example.

```
CREATE TABLE Music (
    Artist VARCHAR(20) NOT NULL,
    SongTitle VARCHAR(30) NOT NULL,
    AlbumTitle VARCHAR(25),
    Year INT,
    Price FLOAT,
    Genre VARCHAR(10),
    Tags TEXT,
    PRIMARY KEY(Artist, SongTitle)
);
```

The primary key for this table consists of *Artist* and *SongTitle*.

You must define all of the table's columns and data types, and the table's primary key. (You can use the ALTER TABLE statement to change these definitions later, if necessary.)

Many SQL implementations let you define storage specifications for your table, as part of the CREATE TABLE statement. Unless you indicate otherwise, the table is created with default storage settings. In a production environment, a database administrator can help determine the optimal storage parameters.

Creating a table with DynamoDB

Use the `CreateTable` operation to create a provisioned mode table, specifying parameters as shown following:

```
{  
    TableName : "Music",  
    KeySchema: [  
        {  
            AttributeName: "Artist",  
            KeyType: "HASH" //Partition key  
        },  
        {  
            AttributeName: "SongTitle",  
            KeyType: "RANGE" //Sort key  
        }  
    ],  
    AttributeDefinitions: [  
        {  
            AttributeName: "Artist",  
            AttributeType: "S"  
        },  
        {  
            AttributeName: "SongTitle",  
            AttributeType: "S"  
        }  
    ],  
    ProvisionedThroughput: {           // Only specified if using provisioned mode  
        ReadCapacityUnits: 1,  
        WriteCapacityUnits: 1  
    }  
}
```

The primary key for this table consists of *Artist* (partition key) and *SongTitle* (sort key).

You must provide the following parameters to `CreateTable`:

- `TableName` – Name of the table.
- `KeySchema` – Attributes that are used for the primary key. For more information, see [Tables, items, and attributes](#) and [Primary key](#).
- `AttributeDefinitions` – Data types for the key schema attributes.

- **ProvisionedThroughput** (for provisioned tables) – Number of reads and writes per second that you need for this table. DynamoDB reserves sufficient storage and system resources so that your throughput requirements are always met. You can use the `UpdateTable` operation to change these later, if necessary. You do not need to specify a table's storage requirements because storage allocation is managed entirely by DynamoDB.

 **Note**

For code examples that use `CreateTable`, see [Getting started with DynamoDB and the AWS SDKs](#).

Getting information about a table

You can verify that a table has been created according to your specifications. In a relational database, all of the table's schema is shown. Amazon DynamoDB tables are schemaless, so only the primary key attributes are shown.

Topics

- [Getting information about a table with SQL](#)
- [Getting information about a table in DynamoDB](#)

Getting information about a table with SQL

Most relational database management systems (RDBMS) allow you to describe a table's structure—columns, data types, primary key definition, and so on. There is no standard way to do this in SQL. However, many database systems provide a `DESCRIBE` command. The following is an example from MySQL.

```
DESCRIBE Music;
```

This returns the structure of your table, with all of the column names, data types, and sizes.

Field	Type	Null	Key	Default	Extra
Artist	varchar(20)	NO	PRI	NULL	

SongTitle varchar(30) NO	PRI	NULL		
AlbumTitle varchar(25) YES		NULL		
Year int(11) YES		NULL		
Price float YES		NULL		
Genre varchar(10) YES		NULL		
Tags text YES		NULL		
+-----+-----+-----+-----+				

The primary key for this table consists of *Artist* and *SongTitle*.

Getting information about a table in DynamoDB

DynamoDB has a `DescribeTable` operation, which is similar. The only parameter is the table name.

```
{  
    TableName : "Music"  
}
```

The reply from `DescribeTable` looks like the following.

```
{  
    "Table": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "Artist",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "AttributeType": "S"  
            }  
        ],  
        "TableName": "Music",  
        "KeySchema": [  
            {  
                "AttributeName": "Artist",  
                "KeyType": "HASH" //Partition key  
            },  
            {  
                "AttributeName": "SongTitle",  
                "KeyType": "RANGE" //Sort key  
            }  
        ]  
    }  
}
```

```
 }  
 ],  
  
 ...
```

DescribeTable also returns information about indexes on the table, provisioned throughput settings, an approximate item count, and other metadata.

Writing data to a table

Relational database tables contain *rows* of data. Rows are composed of *columns*. Amazon DynamoDB tables contain *items*. Items are composed of *attributes*.

This section describes how to write one row (or item) to a table.

Topics

- [Writing data to a table with SQL](#)
- [Writing data to a table in DynamoDB](#)

Writing data to a table with SQL

A table in a relational database is a two-dimensional data structure composed of rows and columns. Some database management systems also provide support for semistructured data, usually with native JSON or XML data types. However, the implementation details vary among vendors.

In SQL, you would use the INSERT statement to add a row to a table.

```
INSERT INTO Music  
  (Artist, SongTitle, AlbumTitle,  
   Year, Price, Genre,  
   Tags)  
VALUES(  
  'No One You Know', 'Call Me Today', 'Somewhat Famous',  
  2015, 2.14, 'Country',  
  '{"Composers": ["Smith", "Jones", "Davis"], "LengthInSeconds": 214}'  
)
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these columns.

Note

This example uses the *Tags* column to store semistructured data about the songs in the *Music* table. The *Tags* column is defined as type TEXT, which can store up to 65,535 characters in MySQL.

Writing data to a table in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to add an item to a table.

DynamoDB API

With the DynamoDB API, you use the PutItem operation to add an item to a table.

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "No One You Know",  
        "SongTitle": "Call Me Today",  
        "AlbumTitle": "Somewhat Famous",  
        "Year": 2015,  
        "Price": 2.14,  
        "Genre": "Country",  
        "Tags": {  
            "Composers": [  
                "Smith",  
                "Jones",  
                "Davis"  
            ],  
            "LengthInSeconds": 214  
        }  
    }  
}
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

Here are some key things to know about this PutItem example:

- DynamoDB provides native support for documents, using JSON. This makes DynamoDB ideal for storing semistructured data, such as *Tags*. You can also retrieve and manipulate data from within JSON documents.
- The *Music* table does not have any predefined attributes, other than the primary key (*Artist* and *SongTitle*).
- Most SQL databases are transaction oriented. When you issue an INSERT statement, the data modifications are not permanent until you issue a COMMIT statement. With Amazon DynamoDB, the effects of a PutItem operation are permanent when DynamoDB replies with an HTTP 200 status code (OK).

 **Note**

For code examples using PutItem, see [Getting started with DynamoDB and the AWS SDKs](#).

The following are some other PutItem examples.

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "No One You Know",  
        "SongTitle": "My Dog Spot",  
        "AlbumTitle": "Hey Now",  
        "Price": 1.98,  
        "Genre": "Country",  
        "CriticRating": 8.4  
    }  
}
```

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "No One You Know",  
        "SongTitle": "Somewhere Down The Road",  
        "AlbumTitle": "Somewhat Famous",  
        "Genre": "Country",  
        "CriticRating": 8.4,  
        "Year": 1984  
    }  
}
```

```
    }  
}
```

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "The Acme Band",  
        "SongTitle": "Still In Love",  
        "AlbumTitle": "The Buck Starts Here",  
        "Price": 2.47,  
        "Genre": "Rock",  
        "PromotionInfo": {  
            "RadioStationsPlaying": [  
                "KHCR", "KBQX", "WTNR", "WJJH"  
            ],  
            "TourDates": {  
                "Seattle": "20150625",  
                "Cleveland": "20150630"  
            },  
            "Rotation": "Heavy"  
        }  
    }  
}
```

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "The Acme Band",  
        "SongTitle": "Look Out, World",  
        "AlbumTitle": "The Buck Starts Here",  
        "Price": 0.99,  
        "Genre": "Rock"  
    }  
}
```

Note

In addition to PutItem, DynamoDB supports a BatchWriteItem operation for writing multiple items at the same time.

PartiQL for DynamoDB

With PartiQL, you use the `ExecuteStatement` operation to add an item to a table, using the PartiQL Insert statement.

```
INSERT into Music value {  
    'Artist': 'No One You Know',  
    'SongTitle': 'Call Me Today',  
    'AlbumTitle': 'Somewhat Famous',  
    'Year' : '2015',  
    'Genre' : 'Acme'  
}
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

 **Note**

For code examples using Insert and `ExecuteStatement`, see [PartiQL insert statements for DynamoDB](#).

Key differences between SQL and DynamoDB when reading data from a table

With SQL, you use the `SELECT` statement to retrieve one or more rows from a table. You use the `WHERE` clause to determine the data that is returned to you.

This is different than using Amazon DynamoDB which provides the following operations for reading data:

- `ExecuteStatement` retrieves a single or multiple items from a table.
`BatchExecuteStatement` retrieves multiple items from different tables in a single operation.
Both of these operations use [PartiQL](#), a SQL-compatible query language.
- `GetItem` – Retrieves a single item from a table. This is the most efficient way to read a single item because it provides direct access to the physical location of the item. (DynamoDB also provides the `BatchGetItem` operation, allowing you to perform up to 100 `GetItem` calls in a single operation.)

- **Query** – Retrieves all of the items that have a specific partition key. Within those items, you can apply a condition to the sort key and retrieve only a subset of the data. Query provides quick, efficient access to the partitions where the data is stored. (For more information, see [Partitions and data distribution](#).)
- **Scan** – Retrieves all of the items in the specified table. (This operation should not be used with large tables because it can consume large amounts of system resources.)

Note

With a relational database, you can use the SELECT statement to join data from multiple tables and return the results. Joins are fundamental to the relational model. To ensure that joins run efficiently, the database and its applications should be performance-tuned on an ongoing basis. DynamoDB is a non-relational NoSQL database that does not support table joins. Instead, applications read data from one table at a time.

The following sections describe different use cases for reading data, and how to perform these tasks with a relational database and with DynamoDB.

Topics

- [Reading an item using its primary key](#)
- [Querying a table](#)
- [Scanning a table](#)

Reading an item using its primary key

One common access pattern for databases is to read a single item from a table. You have to specify the primary key of the item you want.

Topics

- [Reading an item using its primary key with SQL](#)
- [Reading an item using its primary key in DynamoDB](#)

Reading an item using its primary key with SQL

In SQL, you would use the SELECT statement to retrieve data from a table. You can request one or more columns in the result (or all of them, if you use the * operator). The WHERE clause determines which rows to return.

The following is a SELECT statement to retrieve a single row from the *Music* table. The WHERE clause specifies the primary key values.

```
SELECT *
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

You can modify this query to retrieve only a subset of the columns.

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

Reading an item using its primary key in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to read an item from a table.

DynamoDB API

With the DynamoDB API, you use the PutItem operation to add an item to a table.

DynamoDB provides the GetItem operation for retrieving an item by its primary key. GetItem is highly efficient because it provides direct access to the physical location of the item. (For more information, see [Partitions and data distribution](#).)

By default, GetItem returns the entire item with all of its attributes.

```
{
  TableName: "Music",
  Key: {
    "Artist": "No One You Know",
```

```
        "SongTitle": "Call Me Today"
    }
}
```

You can add a `ProjectionExpression` parameter to return only some of the attributes.

```
{
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    "ProjectionExpression": "AlbumTitle, Year, Price"
}
```

Note that the primary key for this table consists of `Artist` and `SongTitle`.

The DynamoDB `GetItem` operation is very efficient. It uses the primary key values to determine the exact storage location of the item in question, and retrieves it directly from there. The SQL `SELECT` statement is similarly efficient, in the case of retrieving items by primary key values.

The SQL `SELECT` statement supports many kinds of queries and table scans. DynamoDB provides similar functionality with its `Query` and `Scan` operations, which are described in [Querying a table](#) and [Scanning a table](#).

The SQL `SELECT` statement can perform table joins, allowing you to retrieve data from multiple tables at the same time. Joins are most effective where the database tables are normalized and the relationships among the tables are clear. However, if you join too many tables in one `SELECT` statement application performance can be affected. You can work around such issues by using database replication, materialized views, or query rewrites.

DynamoDB is a nonrelational database and doesn't support table joins. If you are migrating an existing application from a relational database to DynamoDB, you need to denormalize your data model to eliminate the need for joins.

 **Note**

For code examples that use `GetItem`, see [Getting started with DynamoDB and the AWS SDKs](#).

PartiQL for DynamoDB

With PartiQL, you use the `ExecuteStatement` operation to read an item from a table, using the PartiQL Select statement.

```
SELECT AlbumTitle, Year, Price  
FROM Music  
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

Note that the primary key for this table consists of Artist and SongTitle.

 **Note**

The select PartiQL statement can also be used to Query or Scan a DynamoDB table

For code examples using Select and `ExecuteStatement`, see [PartiQL select statements for DynamoDB](#).

Querying a table

Another common access pattern is reading multiple items from a table, based on your query criteria.

Topics

- [Querying a table with SQL](#)
- [Querying a table in DynamoDB](#)

Querying a table with SQL

When using SQL the `SELECT` statement lets you query on key columns, non-key columns, or any combination. The `WHERE` clause determines which rows are returned, as shown in the following examples.

```
/* Return a single song, by primary key */  
  
SELECT * FROM Music  
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today';
```

```
/* Return all of the songs by an artist */  
  
SELECT * FROM Music  
WHERE Artist='No One You Know';
```

```
/* Return all of the songs by an artist, matching first part of title */  
  
SELECT * FROM Music  
WHERE Artist='No One You Know' AND SongTitle LIKE 'Call%';
```

```
/* Return all of the songs by an artist, with a particular word in the title...  
...but only if the price is less than 1.00 */
```

```
SELECT * FROM Music  
WHERE Artist='No One You Know' AND SongTitle LIKE '%Today%'  
AND Price < 1.00;
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

Querying a table in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to query an item from a table.

DynamoDB API

With Amazon DynamoDB, you can use the `Query` operation to retrieve data in a similar fashion. The `Query` operation provides quick, efficient access to the physical locations where the data is stored. For more information, see [Partitions and data distribution](#).

You can use `Query` with any table or secondary index. You must specify an equality condition for the partition key's value, and you can optionally provide another condition for the sort key attribute if it is defined.

The `KeyConditionExpression` parameter specifies the key values that you want to query. You can use an optional `FilterExpression` to remove certain items from the results before they are returned to you.

In DynamoDB, you must use `ExpressionAttributeValues` as placeholders in expression parameters (such as `KeyConditionExpression` and `FilterExpression`). This is analogous

to the use of *bind variables* in relational databases, where you substitute the actual values into the SELECT statement at runtime.

Note that the primary key for this table consists of *Artist* and *SongTitle*.

The following are some DynamoDB Query examples.

```
// Return a single song, by primary key

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and SongTitle = :t",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call Me Today"
    }
}
```

```
// Return all of the songs by an artist

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a",
    ExpressionAttributeValues: {
        ":a": "No One You Know"
    }
}
```

```
// Return all of the songs by an artist, matching first part of title

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and begins_with(SongTitle, :t)",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call"
    }
}
```

Note

For code examples that use Query, see [Getting started with DynamoDB and the AWS SDKs](#).

PartiQL for DynamoDB

With PartiQL, you can perform a query by using the ExecuteStatement operation and the Select statement on the partition key.

```
SELECT AlbumTitle, Year, Price  
FROM Music  
WHERE Artist='No One You Know'
```

Using the SELECT statement in this way returns all the songs associated with this particular Artist.

For code examples using Select and ExecuteStatement, see [PartiQL select statements for DynamoDB](#).

Scanning a table

In SQL, a SELECT statement without a WHERE clause will return every row in a table. In Amazon DynamoDB, the Scan operation does the same thing. In both cases, you can retrieve all of the items or just some of them.

Whether you are using a SQL or a NoSQL database, scans should be used sparingly because they can consume large amounts of system resources. Sometimes a scan is appropriate (such as scanning a small table) or unavoidable (such as performing a bulk export of data). However, as a general rule, you should design your applications to avoid performing scans. For more information, see [Query operations in DynamoDB](#).

Note

Doing a bulk export also creates at least 1 file per partition. All of the items in each file are from that particular partition's hashed keyspace.

Topics

- [Scanning a table with SQL](#)
- [Scanning a table in DynamoDB](#)

Scanning a table with SQL

When using SQL you can scan a table and retrieve all of its data by using a SELECT statement without specifying a WHERE clause. You can request one or more columns in the result. Or you can request all of them if you use the wildcard character (*).

The following are examples of using a SELECT statement.

```
/* Return all of the data in the table */
SELECT * FROM Music;
```

```
/* Return all of the values for Artist and Title */
SELECT Artist, Title FROM Music;
```

Scanning a table in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to perform a scan on a table.

DynamoDB API

With the DynamoDB API, you use the Scan operation to return one or more items and item attributes by accessing every item in a table or a secondary index.

```
// Return all of the data in the table
{
    TableName: "Music"
}
```

```
// Return all of the values for Artist and Title
{
    TableName: "Music",
    ProjectionExpression: "Artist, Title"
}
```

The Scan operation also provides a `FilterExpression` parameter, which you can use to discard items that you do not want to appear in the results. A `FilterExpression` is applied after the scan is performed, but before the results are returned to you. (This is not recommended with large tables. You are still charged for the entire Scan, even if only a few matching items are returned.)

 **Note**

For code examples that use Scan, see [Getting started with DynamoDB and the AWS SDKs](#).

PartiQL for DynamoDB

With PartiQL, you perform a scan by using the `ExecuteStatement` operation to return all the contents for a table using the `Select` statement.

```
SELECT AlbumTitle, Year, Price  
FROM Music
```

Note that this statement will return all items for in the Music table.

For code examples using Select and `ExecuteStatement`, see [PartiQL select statements for DynamoDB](#).

Managing indexes

Indexes give you access to alternate query patterns, and can speed up queries. This section compares and contrasts index creation and usage in SQL and Amazon DynamoDB.

Whether you are using a relational database or DynamoDB, you should be judicious with index creation. Whenever a write occurs on a table, all of the table's indexes must be updated. In a write-heavy environment with large tables, this can consume large amounts of system resources. In a read-only or read-mostly environment, this is not as much of a concern. However, you should ensure that the indexes are actually being used by your application, and not simply taking up space.

Topics

- [Creating an index](#)

- [Querying and scanning an index](#)

Creating an index

Compare the CREATE INDEX statement in SQL with the UpdateTable operation in Amazon DynamoDB.

Topics

- [Creating an index with SQL](#)
- [Creating an index in DynamoDB](#)

Creating an index with SQL

In a relational database, an index is a data structure that lets you perform fast queries on different columns in a table. You can use the CREATE INDEX SQL statement to add an index to an existing table, specifying the columns to be indexed. After the index has been created, you can query the data in the table as usual, but now the database can use the index to quickly find the specified rows in the table instead of scanning the entire table.

After you create an index, the database maintains it for you. Whenever you modify data in the table, the index is automatically modified to reflect changes in the table.

In MySQL, you would create an index like the following.

```
CREATE INDEX GenreAndPriceIndex  
ON Music (genre, price);
```

Creating an index in DynamoDB

In DynamoDB, you can create and use a *secondary index* for similar purposes.

Indexes in DynamoDB are different from their relational counterparts. When you create a secondary index, you must specify its key attributes—a partition key and a sort key. After you create the secondary index, you can Query it or Scan it just as you would with a table. DynamoDB does not have a query optimizer, so a secondary index is only used when you Query it or Scan it.

DynamoDB supports two different kinds of indexes:

- Global secondary indexes – The primary key of the index can be any two attributes from its table.

- Local secondary indexes – The partition key of the index must be the same as the partition key of its table. However, the sort key can be any other attribute.

DynamoDB ensures that the data in a secondary index is eventually consistent with its table. You can request strongly consistent Query or Scan operations on a table or a local secondary index. However, global secondary indexes support only eventual consistency.

You can add a global secondary index to an existing table, using the UpdateTable operation and specifying GlobalSecondaryIndexUpdates.

```
{  
    TableName: "Music",  
    AttributeDefinitions:[  
        {AttributeName: "Genre", AttributeType: "S"},  
        {AttributeName: "Price", AttributeType: "N"}  
    ],  
    GlobalSecondaryIndexUpdates: [  
        {  
            Create: {  
                IndexName: "GenreAndPriceIndex",  
                KeySchema: [  
                    {AttributeName: "Genre", KeyType: "HASH"}, //Partition key  
                    {AttributeName: "Price", KeyType: "RANGE"} //Sort key  
                ],  
                Projection: {  
                    "ProjectionType": "ALL"  
                },  
                ProvisionedThroughput: {  
                    "ReadCapacityUnits": 1,"WriteCapacityUnits": 1  
                }  
            }  
        }  
    ]  
}
```

You must provide the following parameters to UpdateTable:

- TableName – The table that the index will be associated with.
- AttributeDefinitions – The data types for the key schema attributes of the index.
- GlobalSecondaryIndexUpdates – Details about the index you want to create:

- `IndexName` – A name for the index.
- `KeySchema` – The attributes that are used for the index's primary key.
- `Projection` – Attributes from the table that are copied to the index. In this case, `ALL` means that all of the attributes are copied.
- `ProvisionedThroughput` (for provisioned tables) – The number of reads and writes per second that you need for this index. (This is separate from the provisioned throughput settings of the table.)

Part of this operation involves backfilling data from the table into the new index. During backfilling, the table remains available. However, the index is not ready until its `Backfilling` attribute changes from true to false. You can use the `DescribeTable` operation to view this attribute.

 **Note**

For code examples that use `UpdateTable`, see [Getting started with DynamoDB and the AWS SDKs](#).

Querying and scanning an index

Compare querying and scanning an index using the `SELECT` statement in SQL with the `Query` and `Scan` operations in Amazon DynamoDB.

Topics

- [Querying and scanning an index with SQL](#)
- [Querying and scanning an index in DynamoDB](#)

Querying and scanning an index with SQL

In a relational database, you do not work directly with indexes. Instead, you query tables by issuing `SELECT` statements, and the query optimizer can make use of any indexes.

A *query optimizer* is a relational database management system (RDBMS) component that evaluates the available indexes and determines whether they can be used to speed up a query. If the indexes can be used to speed up a query, the RDBMS accesses the index first and then uses it to locate the data in the table.

Here are some SQL statements that can use *GenreAndPriceIndex* to improve performance. We assume that the *Music* table has enough data in it that the query optimizer decides to use this index, rather than simply scanning the entire table.

```
/* All of the rock songs */
```

```
SELECT * FROM Music  
WHERE Genre = 'Rock';
```

```
/* All of the cheap country songs */
```

```
SELECT Artist, SongTitle, Price FROM Music  
WHERE Genre = 'Country' AND Price < 0.50;
```

Querying and scanning an index in DynamoDB

In DynamoDB, you perform Query and Scan operations directly on the index, in the same way that you would on a table. You can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to query or scan the index. You must specify both TableName and IndexName.

The following are some queries on *GenreAndPriceIndex* in DynamoDB. (The key schema for this index consists of *Genre* and *Price*.)

DynamoDB API

```
// All of the rock songs  
  
{  
    TableName: "Music",  
    IndexName: "GenreAndPriceIndex",  
    KeyConditionExpression: "Genre = :genre",  
    ExpressionAttributeValues: {  
        ":genre": "Rock"  
    },  
};
```

This example uses a `ProjectionExpression` to indicate that you only want some of the attributes, rather than all of them, to appear in the results.

```
// All of the cheap country songs
```

```
{  
    TableName: "Music",  
    IndexName: "GenreAndPriceIndex",  
    KeyConditionExpression: "Genre = :genre and Price < :price",  
    ExpressionAttributeValues: {  
        ":genre": "Country",  
        ":price": 0.50  
    },  
    ProjectionExpression: "Artist, SongTitle, Price"  
};
```

The following is a scan on *GenreAndPriceIndex*.

```
// Return all of the data in the index  
  
{  
    TableName: "Music",  
    IndexName: "GenreAndPriceIndex"  
}
```

PartiQL for DynamoDB

With PartiQL, you use the PartiQL Select statement to perform queries and scans on the index.

```
// All of the rock songs  
  
SELECT *  
FROM Music.GenreAndPriceIndex  
WHERE Genre = 'Rock'
```

```
// All of the cheap country songs  
  
SELECT *  
FROM Music.GenreAndPriceIndex  
WHERE Genre = 'Rock' AND Price < 0.50
```

The following is a scan on *GenreAndPriceIndex*.

```
// Return all of the data in the index  
  
SELECT *
```

```
FROM Music.GenreAndPriceIndex
```

Note

For code examples using Select, see [PartiQL select statements for DynamoDB](#).

Modifying data in a table

The SQL language provides the UPDATE statement for modifying data. Amazon DynamoDB uses the UpdateItem operation to accomplish similar tasks.

Topics

- [Modifying data in a table with SQL](#)
- [Modifying data in a table in DynamoDB](#)

Modifying data in a table with SQL

In SQL, you would use the UPDATE statement to modify one or more rows. The SET clause specifies new values for one or more columns, and the WHERE clause determines which rows are modified. The following is an example.

```
UPDATE Music
SET RecordLabel = 'Global Records'
WHERE Artist = 'No One You Know' AND SongTitle = 'Call Me Today';
```

If no rows match the WHERE clause, the UPDATE statement has no effect.

Modifying data in a table in DynamoDB

In DynamoDB, you can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to modify a single item. If you want to modify multiple items, you must use multiple operations.

DynamoDB API

With the DynamoDB API, you use the UpdateItem operation to modify a single item.

```
{
```

```
TableName: "Music",
Key: {
    "Artist":"No One You Know",
    "SongTitle":"Call Me Today"
},
UpdateExpression: "SET RecordLabel = :label",
ExpressionAttributeValues: {
    ":label": "Global Records"
}
}
```

You must specify the Key attributes of the item to be modified and an UpdateExpression to specify attribute values. UpdateItem behaves like an "upsert" operation. The item is updated if it exists in the table, but if not, a new item is added (inserted).

UpdateItem supports *conditional writes*, where the operation succeeds only if a specific ConditionExpression evaluates to true. For example, the following UpdateItem operation does not perform the update unless the price of the song is greater than or equal to 2.00.

```
{
  TableName: "Music",
  Key: {
    "Artist":"No One You Know",
    "SongTitle":"Call Me Today"
  },
  UpdateExpression: "SET RecordLabel = :label",
  ConditionExpression: "Price >= :p",
  ExpressionAttributeValues: {
    ":label": "Global Records",
    ":p": 2.00
  }
}
```

UpdateItem also supports *atomic counters*, or attributes of type Number that can be incremented or decremented. Atomic counters are similar in many ways to sequence generators, identity columns, or autoincrement fields in SQL databases.

The following is an example of an UpdateItem operation to initialize a new attribute (*Plays*) to keep track of the number of times a song has been played.

```
{
```

```
TableName: "Music",
Key: {
    "Artist":"No One You Know",
    "SongTitle":"Call Me Today"
},
UpdateExpression: "SET Plays = :val",
ExpressionAttributeValues: {
    ":val": 0
},
ReturnValues: "UPDATED_NEW"
}
```

The `ReturnValues` parameter is set to `UPDATED_NEW`, which returns the new values of any attributes that were updated. In this case, it returns 0 (zero).

Whenever someone plays this song, we can use the following `UpdateItem` operation to increment `Plays` by one.

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = Plays + :incr",
    ExpressionAttributeValues: {
        ":incr": 1
    },
    ReturnValues: "UPDATED_NEW"
}
```

Note

For code examples that use `UpdateItem`, see [Getting started with DynamoDB and the AWS SDKs](#).

PartiQL for DynamoDB

With PartiQL, you use the `ExecuteStatement` operation to modify an item in a table, using the PartiQL `Update` statement.

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

```
UPDATE Music
SET RecordLabel = 'Global Records'
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

You can also modify multiple fields at once, such as in the following example.

```
UPDATE Music
SET RecordLabel = 'Global Records'
SET AwardsWon = 10
WHERE Artist = 'No One You Know' AND SongTitle='Call Me Today'
```

Update also supports *atomic counters*, or attributes of type Number that can be incremented or decremented. Atomic counters are similar in many ways to sequence generators, identity columns, or autoincrement fields in SQL databases.

The following is an example of an Update statement to initialize a new attribute (*Plays*) to keep track of the number of times a song has been played.

```
UPDATE Music
SET Plays = 0
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

Whenever someone plays this song, we can use the following Update statement to increment *Plays* by one.

```
UPDATE Music
SET Plays = Plays + 1
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

Note

For code examples using Update and ExecuteStatement, see [PartiQL update statements for DynamoDB](#).

Deleting data from a table

In SQL, the DELETE statement removes one or more rows from a table. Amazon DynamoDB uses the DeleteItem operation to delete one item at a time.

Topics

- [Deleting data from a table with SQL](#)
- [Deleting data from a table in DynamoDB](#)

Deleting data from a table with SQL

In SQL, you use the DELETE statement to delete one or more rows. The WHERE clause determines the rows that you want to modify. The following is an example.

```
DELETE FROM Music  
WHERE Artist = 'The Acme Band' AND SongTitle = 'Look Out, World';
```

You can modify the WHERE clause to delete multiple rows. For example, you could delete all of the songs by a particular artist, as shown in the following example.

```
DELETE FROM Music WHERE Artist = 'The Acme Band'
```

Note

If you omit the WHERE clause, the database attempts to delete all of the rows from the table.

Deleting data from a table in DynamoDB

In DynamoDB, you can use either the DynamoDB API or [PartiQL](#) (a SQL-compatible query language) to delete a single item. If you want to modify multiple items, you must use multiple operations.

DynamoDB API

With the DynamoDB API, you use the DeleteItem operation to delete data from a table, one item at a time. You must specify the item's primary key values.

```
{  
    TableName: "Music",  
    Key: {  
        Artist: "The Acme Band",  
        SongTitle: "Look Out, World"  
    }  
}
```

Note

In addition to `DeleteItem`, Amazon DynamoDB supports a `BatchWriteItem` operation for deleting multiple items at the same time.

`DeleteItem` supports *conditional writes*, where the operation succeeds only if a specific `ConditionExpression` evaluates to true. For example, the following `DeleteItem` operation deletes the item only if it has a `RecordLabel` attribute.

```
{  
    TableName: "Music",  
    Key: {  
        Artist: "The Acme Band",  
        SongTitle: "Look Out, World"  
    },  
    ConditionExpression: "attribute_exists(RecordLabel)"  
}
```

Note

For code examples that use `DeleteItem`, see [Getting started with DynamoDB and the AWS SDKs](#).

PartiQL for DynamoDB

With PartiQL, you use the `Delete` statement through the `ExecuteStatement` operation to delete data from a table, one item at a time. You must specify the item's primary key values.

The primary key for this table consists of `Artist` and `SongTitle`. You must specify values for these attributes.

```
DELETE FROM Music  
WHERE Artist = 'Acme Band' AND SongTitle = 'PartiQL Rocks'
```

You can also specify additional conditions for the operation. The following DELETE operation only deletes the item if it has more than 11 Awards.

```
DELETE FROM Music  
WHERE Artist = 'Acme Band' AND SongTitle = 'PartiQL Rocks' AND Awards > 11
```

Note

For code examples using DELETE and ExecuteStatement, see [PartiQL delete statements for DynamoDB](#).

Removing a table

In SQL, you use the DROP TABLE statement to remove a table. In Amazon DynamoDB, you use the DeleteTable operation.

Topics

- [Removing a table with SQL](#)
- [Removing a table in DynamoDB](#)

Removing a table with SQL

When you no longer need a table and want to discard it permanently, you would use the DROP TABLE statement in SQL.

```
DROP TABLE Music;
```

After a table is dropped, it cannot be recovered. (Some relational databases do allow you to undo a DROP TABLE operation, but this is vendor-specific functionality and it is not widely implemented.)

Removing a table in DynamoDB

In DynamoDB, DeleteTable is a similar operation. In the following example, the table is permanently deleted.

```
{  
    TableName: "Music"  
}
```

Note

For code examples that use `DeleteTable`, see [Getting started with DynamoDB and the AWS SDKs](#).

Additional resources for Amazon DynamoDB

You can use the following additional resources to understand and work with DynamoDB.

Topics

- [Tools for coding and visualization](#)
- [Prescriptive Guidance articles](#)
- [Knowledge Center articles](#)
- [Blog posts, repositories, and guides](#)
- [Data modeling and design pattern presentations](#)
- [Training courses](#)

Tools for coding and visualization

You can use the following coding and visualization tools to work with DynamoDB:

- [NoSQL Workbench for Amazon DynamoDB](#) – A unified, visual tool that helps you design, create, query, and manage DynamoDB tables. It provides data modeling, data visualization, and query development features.
- [Dynobase](#) – A desktop tool that makes it easy to see your DynamoDB tables and work with them, create app code, and edit records with real-time validation.
- [DynamoDB Toolbox](#) – A project from Jeremy Daly that provides helpful utilities for working with data modeling and JavaScript and Node.js.
- [DynamoDB Streams Processor](#) – A simple tool that you can use to work with [DynamoDB streams](#).

Prescriptive Guidance articles

AWS Prescriptive Guidance provides time-tested strategies, guides, and patterns to help accelerate your projects. These resources were developed by AWS technology experts and the global community of AWS Partners, based on their years of experience helping customers achieve their business objectives.

Data modeling and migration

- [A hierarchical data model in DynamoDB](#)
- [Modeling data with DynamoDB](#)
- [Migrate an Oracle database to DynamoDB using AWS DMS](#)

Global tables

- [Using Amazon DynamoDB global tables](#)

Serverless

- [Implement the serverless saga pattern with AWS Step Functions](#)

SaaS architecture

- [Manage tenants across multiple SaaS products on a single control plane](#)
- [Tenant onboarding in SaaS architecture for the silo model using C# and AWS CDK](#)

Data protection and data movement

- [Configure cross-account access to Amazon DynamoDB](#)
- [Full table copy options for DynamoDB](#)
- [Disaster recovery strategy for databases on AWS](#)

Miscellaneous

- [Help enforce tagging in DynamoDB](#)

Prescriptive guidance video walkthroughs

- [Using Serverless Architecture to Create Data Pipelines](#)
- [Novartis - Buying Engine: AI-powered Procurement Portal](#)
- [Veritiv: Enable Insights to Forecast Sales Demand on AWS Data Lakes](#)
- [mimik: Hybrid Edge Cloud Leveraging AWS to Support Edge Microservice Mesh](#)
- [Change Data Capture with Amazon DynamoDB](#)

For additional Prescriptive Guidance articles and videos for DynamoDB, see [Prescriptive Guidance](#).

Knowledge Center articles

The AWS Knowledge Center articles and videos cover the most frequent questions and requests that we receive from AWS customers. The following are some current Knowledge Center articles on specific tasks that relate to DynamoDB:

Cost optimization

- [How do I optimize costs with Amazon DynamoDB?](#)

Throttling and latency

- [Why is my DynamoDB maximum latency metric high when the average latency is normal?](#)
- [Why is my DynamoDB table being throttled?](#)
- [Why is my on-demand DynamoDB table being throttled?](#)

Pagination

- [How do I implement pagination in DynamoDB](#)

Transactions

- [Why is my TransactWriteItems API call failing in DynamoDB](#)

Troubleshooting

- [How do I resolve issues with DynamoDB auto scaling?](#)
- [How do I troubleshoot HTTP 4XX errors in DynamoDB](#)

For additional articles and videos for DynamoDB, see the [Knowledge Center articles](#).

Blog posts, repositories, and guides

In addition to the [DynamoDB Developer Guide](#), there are many useful resources for working with DynamoDB. Here are some selected blog posts, repositories, and guides for working with DynamoDB:

- AWS repository of [DynamoDB code examples](#) in various AWS SDK languages: [Node.js](#), [Java](#), [Python](#), [.Net](#), [Go](#), and [Rust](#).
- [The DynamoDB Book](#) – A comprehensive guide from [Alex DeBrie](#) that teaches a strategy-driven approach to data modeling with DynamoDB.
- [DynamoDB guide](#) – An open guide from [Alex DeBrie](#) that walks through the basic concepts and advanced features of the DynamoDB NoSQL database.
- [How to switch from RDBMS to DynamoDB in 20 easy steps](#) – A list of useful steps for learning data modeling from [Jeremy Daly](#).
- [DynamoDB JavaScript DocumentClient cheat sheet](#) – A cheat sheet to help you get started building applications with DynamoDB in a Node.js or JavaScript environment.
- [DynamoDB Core Concept Videos](#) – This playlist covers many of the core concepts of DynamoDB.

Data modeling and design pattern presentations

You can use the following resources on data modeling and design patterns to help you get the most out of DynamoDB:

- [AWS re:Invent 2019: Data modeling with DynamoDB](#)
 - A talk by [Alex DeBrie](#) that helps you start with the principles of DynamoDB data modeling.
- [AWS re:Invent 2020: Data modeling with DynamoDB – Part 1](#)
- [AWS re:Invent 2020: Data modeling with DynamoDB – Part 2](#)
- [AWS re:Invent 2017: Advanced design patterns](#)
- [AWS re:Invent 2018: Advanced design patterns](#)
- [AWS re:Invent 2019: Advanced design patterns](#)

- Jeremy Daly shares his [12 key takeaways](#) from this session.
- [AWS re:Invent 2020: DynamoDB advanced design patterns – Part 1](#)
- [AWS re:Invent 2020: DynamoDB advanced design patterns – Part 2](#)
- [DynamoDB Office Hours on Twitch](#)

 **Note**

Each session covers different use cases and examples.

Training courses

There are many different training courses and educational options for learning more about DynamoDB. Here are some current examples:

- [Developing with Amazon DynamoDB](#) – Designed by AWS to take you from beginner to expert in developing real-world applications with data modeling for Amazon DynamoDB.
- [DynamoDB deep-dive course](#) – A course from A Cloud Guru.
- [Amazon DynamoDB: Building NoSQL database-driven applications](#) – A course from the AWS Training and Certification team hosted on edX.

Setting up DynamoDB

In addition to the Amazon DynamoDB web service, AWS provides a downloadable version of DynamoDB that you can run on your computer. The downloadable version is helpful for developing and testing your code. It lets you write and test applications locally without accessing the DynamoDB web service.

The topics in this section describe how to set up DynamoDB (downloadable version) and the DynamoDB web service.

Topics

- [Setting up DynamoDB local \(downloadable version\)](#)
- [Setting up DynamoDB \(web service\)](#)

Setting up DynamoDB local (downloadable version)

With the downloadable version of Amazon DynamoDB, you can develop and test applications without accessing the DynamoDB web service. Instead, the database is self-contained on your computer. When you're ready to deploy your application in production, you remove the local endpoint in the code, and then it points to the DynamoDB web service.

Having this local version helps you save on throughput, data storage, and data transfer fees. In addition, you don't need an internet connection while you develop your application.

DynamoDB local is available as a [download](#) (requires JRE), as an [Apache Maven dependency](#), or as a [Docker image](#).

If you prefer to use the Amazon DynamoDB web service instead, see [Setting up DynamoDB \(web service\)](#).

Topics

- [Deploying DynamoDB locally on your computer](#)
- [DynamoDB local usage notes](#)
- [Release history for DynamoDB local](#)
- [Telemetry in DynamoDB local](#)

Deploying DynamoDB locally on your computer

Important

DynamoDB local jar can be downloaded from our AWS CloudFront distribution links referenced here. Starting January 1, 2025, the old S3 distribution buckets will no longer be active and DynamoDB local will be distributed through CloudFront distribution links only.

There are two major versions of DynamoDB local available: DynamoDB local v2.x (Current) and DynamoDB local v1.x (Legacy). Customers should use version 2.x (Current) when possible, as it supports the latest versions of the Java Runtime Environment and is compatible with the jakarta.* namespace for Maven project. DynamoDB local v1.x will reach end of standard support starting on January 1, 2025. After this date, v1.x will no longer receive updates or bug fixes.

Note

DynamoDB local AWS_ACCESS_KEY_ID can contain only letters (A–Z, a–z) and numbers (0–9).

Download DynamoDB local

Follow these steps to set up and run DynamoDB on your computer.

To set up DynamoDB on your computer

1. Download DynamoDB local for free from one of the following locations.

Download Links	Checksums
.tar.gz .zip	.tar.gz.sha256 .zip.sha256

⚠ Important

To run DynamoDB v2.3.0 or greater on your computer, you must have the Java Runtime Environment (JRE) version 17.x or newer. The application doesn't run on earlier JRE versions.

2. After you download the archive, extract the contents and copy the extracted directory to a location of your choice.
3. To start DynamoDB on your computer, open a command prompt window, navigate to the directory where you extracted DynamoDBLocal.jar, and enter the following command.

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
```

ⓘ Note

If you're using Windows PowerShell, be sure to enclose the parameter name or the entire name and value like this:

```
java -D"java.library.path=./DynamoDBLocal_lib" -jar  
DynamoDBLocal.jar
```

DynamoDB processes incoming requests until you stop it. To stop DynamoDB, press Ctrl+C at the command prompt.

DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. For a complete list of DynamoDB runtime options, including –port, enter this command.

```
java -Djava.library.path=./DynamoDBLocal_lib -jar  
DynamoDBLocal.jar -help
```

4. Before you can access DynamoDB programmatically or through the AWS Command Line Interface (AWS CLI), you must configure your credentials to enable authorization for your applications. Downloadable DynamoDB requires any credentials to work, as shown in the following example.

```
AWS Access Key ID: "fakeMyKeyId"  
AWS Secret Access Key: "fakeSecretAccessKey"  
Default Region Name: "fakeRegion"
```

You can use the `aws configure` command of the AWS CLI to set up credentials. For more information, see [Using the AWS CLI](#).

5. Start writing applications. To access DynamoDB running locally with the AWS CLI, use the `--endpoint-url` parameter. For example, use the following command to list DynamoDB tables.

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

Run DynamoDB local as Docker image

The downloadable version of Amazon DynamoDB is available as a Docker image. For more information, see [dynamodb-local](#). To see your current DynamoDB local version, enter the following command:

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -version
```

For an example of using DynamoDB local as part of a REST application built on the AWS Serverless Application Model (AWS SAM), see [SAM DynamoDB application for managing orders](#). This sample application demonstrates how to use DynamoDB local for testing.

If you want to run a multi-container application that also uses the DynamoDB local container, use Docker Compose to define and run all the services in your application, including DynamoDB local.

To install and run DynamoDB local with Docker compose:

1. Download and install [Docker desktop](#).
2. Copy the following code to a file and save it as `docker-compose.yml`.

```
version: '3.8'
services:
  dynamodb-local:
    command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
    image: "amazon/dynamodb-local:latest"
    container_name: dynamodb-local
    ports:
      - "8000:8000"
    volumes:
      - "./docker/dynamodb:/home/dynamodblocal/data"
```

```
working_dir: /home/dynamodblocal
```

If you want your application and DynamoDB local to be in separate containers, use the following yaml file.

```
version: '3.8'
services:
  dynamodb-local:
    command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
    image: "amazon/dynamodb-local:latest"
    container_name: dynamodb-local
    ports:
      - "8000:8000"
    volumes:
      - "./docker/dynamodb:/home/dynamodblocal/data"
    working_dir: /home/dynamodblocal
  app-node:
    depends_on:
      - dynamodb-local
    image: amazon/aws-cli
    container_name: app-node
    ports:
      - "8080:8080"
    environment:
      AWS_ACCESS_KEY_ID: 'DUMMYIDEXAMPLE'
      AWS_SECRET_ACCESS_KEY: 'DUMMYEXAMPLEKEY'
    command:
      dynamodb describe-limits --endpoint-url http://dynamodb-local:8000 --region us-west-2
```

This docker-compose.yml script creates an app-node container and a dynamodb-local container. The script runs a command in the app-node container that uses the AWS CLI to connect to the dynamodb-local container and describes the account and table limits.

To use with your own application image, replace the `image` value in the example below with that of your application.

```
version: '3.8'
services:
  dynamodb-local:
    command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
```

```
image: "amazon/dynamodb-local:latest"
container_name: dynamodb-local
ports:
  - "8000:8000"
volumes:
  - "./docker/dynamodb:/home/dynamodblocal/data"
working_dir: /home/dynamodblocal
app-node:
  image: location-of-your-dynamodb-demo-app:latest
  container_name: app-node
  ports:
    - "8080:8080"
  depends_on:
    - "dynamodb-local"
links:
  - "dynamodb-local"
environment:
  AWS_ACCESS_KEY_ID: 'DUMMYIDEXAMPLE'
  AWS_SECRET_ACCESS_KEY: 'DUMMYEXAMPLEKEY'
  REGION: 'eu-west-1'
```

Note

The YAML scripts require that you specify an AWS access key and an AWS secret key, but they are not required to be valid AWS keys for you to access DynamoDB local.

3. Run the following command-line command:

```
docker-compose up
```

Run DynamoDB local as an Apache Maven dependency

Follow these steps to use Amazon DynamoDB in your application as a dependency.

To deploy DynamoDB as an Apache Maven repository

1. Download and install Apache Maven. For more information, see [Downloading Apache Maven](#) and [Installing Apache Maven](#).
2. Add the DynamoDB Maven repository to your application's Project Object Model (POM) file.

```
<!--Dependency:-->
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>DynamoDBLocal</artifactId>
        <version>2.3.0</version>
    </dependency>
</dependencies>
```

Example template for use with Spring Boot 3 and/or Spring Framework 6:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringMavenDynamoDB</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <spring-boot.version>3.0.1</spring-boot.version>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.1</version>
</parent>

<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>DynamoDBLocal</artifactId>
        <version>2.3.0</version>
    </dependency>
    <!-- Spring Boot -->
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>${spring-boot.version}</version>
</dependency>
<!-- Spring Web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>${spring-boot.version}</version>
</dependency>
<!-- Spring Data JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>${spring-boot.version}</version>
</dependency>
<!-- Other Spring dependencies -->
<!-- Replace the version numbers with the desired version -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.0.0</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>6.0.0</version>
</dependency>
<!-- Add other Spring dependencies as needed -->
<!-- Add any other dependencies your project requires -->
</dependencies>
</project>
```

 **Note**

You can also use the [Maven central repository](#) URL.

For an example of a sample project that showcases multiple approaches to set up and use DynamoDB local, including downloading JAR files, running it as a Docker image, and using it as a Maven dependency, see [DynamoDB Local Sample Java Project](#).

DynamoDB local usage notes

Except for the endpoint, applications that run with the downloadable version of Amazon DynamoDB should also work with the DynamoDB web service. However, when using DynamoDB locally, you should be aware of the following:

- If you use the `-sharedDb` option, DynamoDB creates a single database file named *shared-local-instance.db*. Every program that connects to DynamoDB accesses this file. If you delete the file, you lose any data that you have stored in it.
- If you omit `-sharedDb`, the database file is named *myaccesskeyid_region.db*, with the AWS access key ID and AWS Region as they appear in your application configuration. If you delete the file, you lose any data that you have stored in it.
- If you use the `-inMemory` option, DynamoDB doesn't write any database files at all. Instead, all data is written to memory, and the data is not saved when you terminate DynamoDB.
- If you use the `-inMemory` option, the `-sharedDb` option is also required.
- If you use the `-optimizeDbBeforeStartup` option, you must also specify the `-dbPath` parameter so that DynamoDB can find its database file.
- The AWS SDKs for DynamoDB require that your application configuration specify an access key value and an AWS Region value. Unless you're using the `-sharedDb` or the `-inMemory` option, DynamoDB uses these values to name the local database file. These values don't have to be valid AWS values to run locally. However, you might find it convenient to use valid values so that you can run your code in the cloud later by changing the endpoint you're using.
- DynamoDB local always returns null for `billingModeSummary`.
- DynamoDB local `AWS_ACCESS_KEY_ID` can contain only letters (A–Z, a–z) and numbers (0–9).

Topics

- [Command line options](#)
- [Setting the local endpoint](#)
- [Differences between downloadable DynamoDB and the DynamoDB web service](#)

Command line options

You can use the following command line options with the downloadable version of DynamoDB:

- **-cors value** — Enables support for cross-origin resource sharing (CORS) for JavaScript. You must provide a comma-separated "allow" list of specific domains. The default setting for **-cors** is an asterisk (*), which allows public access.
- **-dbPath value** — The directory where DynamoDB writes its database file. If you don't specify this option, the file is written to the current directory. You can't specify both **-dbPath** and **-inMemory** at once.
- **-delayTransientStatuses** — Causes DynamoDB to introduce delays for certain operations. DynamoDB (downloadable version) can perform some tasks almost instantaneously, such as create/update/delete operations on tables and indexes. However, the DynamoDB service requires more time for these tasks. Setting this parameter helps DynamoDB running on your computer simulate the behavior of the DynamoDB web service more closely. (Currently, this parameter introduces delays only for global secondary indexes that are in either *CREATING* or *DELETING* status.)
- **-help** — Prints a usage summary and options.
- **-inMemory** — DynamoDB runs in memory instead of using a database file. When you stop DynamoDB, none of the data is saved. You can't specify both **-dbPath** and **-inMemory** at once.
- **-optimizeDbBeforeStartup** — Optimizes the underlying database tables before starting DynamoDB on your computer. You also must specify **-dbPath** when you use this parameter.
- **-port value** — The port number that DynamoDB uses to communicate with your application. If you don't specify this option, the default port is 8000.

 **Note**

DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. You can use the **-port** option to specify a different port number. For a complete list of DynamoDB runtime options, including **-port**, type this command:

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar  
-help
```

- **-sharedDb** — If you specify **-sharedDb**, DynamoDB uses a single database file instead of separate files for each credential and Region.
- **-disableTelemetry** — When specified, DynamoDB local will not send any telemetry.
- **-version** — Prints the version of DynamoDB local.

Setting the local endpoint

By default, the AWS SDKs and tools use endpoints for the Amazon DynamoDB web service. To use the SDKs and tools with the downloadable version of DynamoDB, you must specify the local endpoint:

`http://localhost:8000`

AWS Command Line Interface

You can use the AWS Command Line Interface (AWS CLI) to interact with downloadable DynamoDB. For example, you can use it to perform all the steps in [Creating tables and loading data for code examples in DynamoDB](#).

To access DynamoDB running locally, use the `--endpoint-url` parameter. The following is an example of using the AWS CLI to list the tables in DynamoDB on your computer.

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

Note

The AWS CLI can't use the downloadable version of DynamoDB as a default endpoint. Therefore, you must specify `--endpoint-url` with each AWS CLI command.

AWS SDKs

The way you specify an endpoint depends on the programming language and AWS SDK you're using. The following sections describe how to do this:

- [Java: Setting the AWS Region and endpoint](#) (DynamoDB local supports the AWS SDK for Java V1 and V2)
- [.NET: Setting the AWS Region and endpoint](#)

Note

For examples in other programming languages, see [Getting started with DynamoDB and the AWS SDKs](#).

Differences between downloadable DynamoDB and the DynamoDB web service

The downloadable version of DynamoDB is intended for development and testing purposes only. By comparison, the DynamoDB web service is a managed service with scalability, availability, and durability features that make it ideal for production use.

The downloadable version of DynamoDB differs from the web service in the following ways:

- AWS Regions and distinct AWS accounts are not supported at the client level.
- Provisioned throughput settings are ignored in downloadable DynamoDB, even though the `CreateTable` operation requires them. For `CreateTable`, you can specify any numbers you want for provisioned read and write throughput, even though these numbers are not used. You can call `UpdateTable` as many times as you want per day. However, any changes to provisioned throughput values are ignored.
- Scan operations are performed sequentially. Parallel scans are not supported. The `Segment` and `TotalSegments` parameters of the `Scan` operation are ignored.
- The speed of read and write operations on table data is limited only by the speed of your computer. `CreateTable`, `UpdateTable`, and `DeleteTable` operations occur immediately, and table state is always ACTIVE. `UpdateTable` operations that change only the provisioned throughput settings on tables or global secondary indexes occur immediately. If an `UpdateTable` operation creates or deletes any global secondary indexes, then those indexes transition through normal states (such as CREATING and DELETING, respectively) before they become an ACTIVE state. The table remains ACTIVE during this time.
- Read operations are eventually consistent. However, due to the speed of DynamoDB running on your computer, most reads appear to be strongly consistent.
- Item collection metrics and item collection sizes are not tracked. In operation responses, nulls are returned instead of item collection metrics.
- In DynamoDB, there is a 1 MB limit on data returned per result set. Both the DynamoDB web service and the downloadable version enforce this limit. However, when querying an index, the DynamoDB service calculates only the size of the projected key and attributes. By contrast, the downloadable version of DynamoDB calculates the size of the entire item.
- If you're using DynamoDB Streams, the rate at which shards are created might differ. In the DynamoDB web service, shard-creation behavior is partially influenced by table partition activity. When you run DynamoDB locally, there is no table partitioning. In either case, shards are ephemeral, so your application should not be dependent on shard behavior.

- `TransactionConflictExceptions` aren't thrown by downloadable DynamoDB for transactional APIs. We recommend that you use a Java mocking framework to simulate `TransactionConflictExceptions` in the DynamoDB handler to test how your application responds to conflicting transactions.
- In the DynamoDB web service, whether being accessed via the console or the AWS CLI, table names are case sensitive. A table named `Authors` and one named `authors` can both exist as separate tables. In the downloadable version, table names are case insensitive, and attempting to create these two tables would result in an error.
- Tagging is not supported in the downloadable version of DynamoDB.
- The downloadable version of DynamoDB ignores the [Limit](#) parameter in [ExecuteStatement](#).

Release history for DynamoDB local

The following table describes the important changes in each release of *DynamoDB local*.

Version	Change	Description	Date
2.3.0	Jetty and JDK Upgrade	<ul style="list-style-type: none">Upgrading to Jetty 12.0.2Upgrading to JDK 17Upgrading ANTLR4 to 4.10.1	March 14, 2024
2.2.0	Added support for table deletion protection and the <code>ReturnValuesOnConditionCheckFailure</code> parameter	<ul style="list-style-type: none">Added support of Table delete protectionAdded support for <code>ReturnValuesOnConditionCheckFailure</code>Added support for -version flag	December 14, 2023
2.1.0	Support for SQLite Native Libraries for	<ul style="list-style-type: none">Adding telemetry to DynamoDB local	October 23, 2023

Version	Change	Description	Date
	Maven projects and adding telemetry	<ul style="list-style-type: none"> • Dynamically copy SQLLite Native Libraries for Maven projects • Removed io.github.ganadist.sqlite4j ava library from Maven dependency • Upgrading GoogleGuava to 32.1.1-jre 	
2.0.0	Migrating from javax to jakarta namespace and JDK11 Support	<ul style="list-style-type: none"> • Migrating from javax to jakarta namespace and JDK11 support • Fix for handling invalid access and secret key while server startup • Fixing Maven identified vulnerabilities by updating dependencies 	July 5, 2023
1.25.0	Added support for table deletion protection and the ReturnVal uesOnCond itionChec kFailure parameter	<ul style="list-style-type: none"> • Added support of Table delete protection • Added support for ReturnVal uesOnCond itionCheckFailure • Added support for -version flag 	December 18, 2023

Version	Change	Description	Date
1.24.0	Support for SQLLite Native Libraries for Maven projects and adding telemetry	<ul style="list-style-type: none"> • Adding telemetry to DynamoDB local • Dynamically copy SQLLite Native Libraries for Maven projects • Removed io.github.ganadist.sqlite4j ava library from Maven dependency • Upgrading GoogleGuava to 32.1.1-jre 	October 23, 2023
1.23.0	Handle invalid access and secret key while server startup	<ul style="list-style-type: none"> • Fix for handling invalid access and secret key while server startup • Fixing Maven identified vulnerabilities by updating dependencies 	June 28, 2023
1.22.0	Support of Limit Operation for PartiQL	<ul style="list-style-type: none"> • Optimize IN clause for PartiQL • Support for Limit Operation • M1 support for Maven projects 	June 8, 2023

Version	Change	Description	Date
1.21.0	Support for 100 actions per transaction	<ul style="list-style-type: none"> • Increased actions per transaction from 25 to 100 • Upgrading docker image Open JDK to 11 • Fixing the parity for exception thrown when duplicate items in BatchExecuteStatement 	January 26, 2023
1.20.0	Added support for M1 Mac	<ul style="list-style-type: none"> • Added support for M1 Mac • Upgrading Jetty dependency to 9.4.48.v20220622 	September 12, 2022
1.19.0	Upgraded the PartiQL Parser	Upgraded the PartiQL Parser and other related libraries	July 27, 2022
1.18.0	Upgraded log4j-core and Jackson-core	Upgraded log4j-core to 2.17.1 and Jackson-core 2.10.x to 2.12.0	January 10, 2022
1.17.2	Upgraded log4j-core	Upgraded log4j-core dependency to version 2.16	January 16, 2021

Version	Change	Description	Date
1.17.1	Upgraded log4j-core	Updated log4j-core dependency to patch zero-day exploit to prevent remote code execution - Log4Shell	January 10, 2021
1.17.0	Deprecated Javascript Web Shell	<ul style="list-style-type: none">• Updated the AWS SDK dependency to AWS SDK for Java 1.12.x• Deprecated Javascript Web Shell	January 8, 2021

Telemetry in DynamoDB local

At AWS, we develop and launch services based on what we learn from interactions with customers, and we use customer feedback to iterate on our products. Telemetry is additional information that helps us to better understand our customers needs, diagnose issues, and deliver features that improve the customer experience.

DynamoDB local collects telemetry, such as generic usage metrics, systems and environment information, and errors. For details about the types of telemetry collected, see [Types of information collected](#).

DynamoDB local does not collect personal information, such as user names or email addresses. It also does not extract sensitive project-level information.

As a customer, you control whether telemetry is turned on, and you can change your settings at any point in time. If telemetry remains on, DynamoDB local sends telemetry data in the background without requiring any additional customer interaction.

Turn off telemetry using command line options

You can turn off telemetry using command line options when starting DynamoDB local using the option `-disableTelemetry`. For more information, see [Command line options](#)

Turn off telemetry for a single session

In macOS and Linux operating systems, you can turn off telemetry for a single session. To turn off telemetry for your current session, run the following command to set the environment variable DDB_LOCAL_TELEMETRY to false. Repeat the command for each new terminal or session.

```
export DDB_LOCAL_TELEMETRY=0
```

Turn off telemetry for your profile in all sessions

Run the following commands to turn off telemetry for all sessions when you're running DynamoDB local on your operating system.

To turn off telemetry in Linux

1. Run:

```
echo "export DDB_LOCAL_TELEMETRY=0" >>~/.profile
```

2. Run:

```
source ~/.profile
```

To turn off telemetry in macOS

1. Run:

```
echo "export DDB_LOCAL_TELEMETRY=0" >>~/.profile
```

2. Run:

```
source ~/.profile
```

To turn off telemetry in Windows

1. Run:

```
setx DDB_LOCAL_TELEMETRY 0
```

2. Run:

```
refreshenv
```

Types of information collected

- **Usage information** — The generic telemetry like server start/stop and the API or Operation called.
- **System and environment information** — The Java version, operating system (Windows, Linux or macOS), the environment in which DynamoDB local runs (for example, Stand alone JAR, Docker container, or as a Maven Dependency), and hash values of usage attributes.

Learn more

The telemetry data that DynamoDB local collects adheres to the AWS data privacy policies. For more information, see the following:

- [AWS service terms](#)
- [Data privacy FAQ](#)

Setting up DynamoDB (web service)

To use the Amazon DynamoDB web service:

1. [Sign up for AWS](#).
2. [Get an AWS access key](#) (used to access DynamoDB programmatically).

 **Note**

If you plan to interact with DynamoDB only through the AWS Management Console, you don't need an AWS access key, and you can skip ahead to [Using the console](#).

3. [Configure your credentials](#) (used to access DynamoDB programmatically).

Signing up for AWS

To use the DynamoDB service, you must have an AWS account. If you don't already have an account, you are prompted to create one when you sign up. You're not charged for any AWS services that you sign up for unless you use them.

To sign up for AWS

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

Granting programmatic access

Before you can access DynamoDB programmatically or through the AWS Command Line Interface (AWS CLI), you must have programmatic access. You don't need programmatic access if you plan to use the DynamoDB console only.

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use.

Which user needs programmatic access?	To	By
		<ul style="list-style-type: none">For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the <i>AWS Command Line Interface User Guide</i>.For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the <i>AWS SDKs and Tools Reference Guide</i>.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the <i>IAM User Guide</i> .

Which user needs programmatic access?	To	By
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. <ul style="list-style-type: none">• For the AWS CLI, see Authenticating using IAM user credentials in the <i>AWS Command Line Interface User Guide</i>.• For AWS SDKs and tools, see Authenticate using long-term credentials in the <i>AWS SDKs and Tools Reference Guide</i>.• For AWS APIs, see Managing access keys for IAM users in the <i>IAM User Guide</i>.

Configuring your credentials

Before you can access DynamoDB programmatically or through the AWS CLI, you must configure your credentials to enable authorization for your applications.

There are several ways to do this. For example, you can manually create the credentials file to store your access key ID and secret access key. You also can use the AWS CLI command `aws configure` to automatically create the file. Alternatively, you can use environment variables. For more information about configuring your credentials, see the programming-specific AWS SDK developer guide.

To install and configure the AWS CLI, see [Using the AWS CLI](#).

Integrating with other DynamoDB services

You can integrate DynamoDB with many other AWS services. For more information, see the following:

- [Using DynamoDB with other AWS services](#)
- [AWS CloudFormation for DynamoDB](#)
- [Using AWS Backup with DynamoDB](#)
- [AWS Identity and Access Management \(IAM\)](#)
- [Using AWS Identity and Access Management with DynamoDB](#)

Accessing DynamoDB

You can access Amazon DynamoDB using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API.

Topics

- [Using the console](#)
- [Using the AWS CLI](#)
- [Using the API](#)
- [Using the NoSQL workbench for DynamoDB](#)
- [IP address ranges](#)

Using the console

You can access the AWS Management Console for Amazon DynamoDB at <https://console.aws.amazon.com/dynamodb/home>.

You can use the console to do the following in DynamoDB:

- Monitor recent alerts, total capacity, service health, and the latest DynamoDB news on the DynamoDB dashboard.
- Create, update, and delete tables. The capacity calculator provides estimates of how many capacity units to request based on the usage information you provide.
- Manage streams.
- View, add, update, and delete items that are stored in tables. Manage Time to Live (TTL) to define when items in a table expire so that they can be automatically deleted from the database.
- Query and scan a table.
- Set up and view alarms to monitor your table's capacity usage. View your table's top monitoring metrics on real-time graphs from CloudWatch.
- Modify a table's provisioned capacity.
- Modify a table's table class.
- Create and delete global secondary indexes.

- Create triggers to connect DynamoDB streams to AWS Lambda functions.
- Apply tags to your resources to help organize and identify them.
- Purchase reserved capacity.

The console displays an introductory screen that prompts you to create your first table. To view your tables, in the navigation pane on the left side of the console, choose **Tables**.

Here's a high-level overview of the actions available per table within each navigation tab:

- **Overview** – View table details, including item count and metrics.
- **Indexes** – Manage global and local secondary indexes.
- **Monitor** – View alarms, CloudWatch Contributor Insights, and Cloudwatch metrics.
- **Global tables** – Manage table replicas.
- **Backups** – Manage point-in-time recovery and on-demand backups.
- **Exports and streams** – Export your table to Amazon S3 and manage DynamoDB Streams and Kinesis Data Streams.
- **Additional settings** – Manage read/write capacity, Time to Live settings, encryption, and tags.

Using the AWS CLI

You can use the AWS Command Line Interface (AWS CLI) to control multiple AWS services from the command line and automate them through scripts. You can use the AWS CLI for ad hoc operations, such as creating a table. You can also use it to embed Amazon DynamoDB operations within utility scripts.

Before you can use the AWS CLI with DynamoDB, you must get an access key ID and secret access key. For more information, see [Granting programmatic access](#).

For a complete listing of all the commands available for DynamoDB in the AWS CLI, see the [AWS CLI command reference](#).

Topics

- [Downloading and configuring the AWS CLI](#)
- [Using the AWS CLI with DynamoDB](#)
- [Using the AWS CLI with DynamoDB local](#)

Downloading and configuring the AWS CLI

The AWS CLI is available at <http://aws.amazon.com/cli>. It runs on Windows, macOS, or Linux. After you download the AWS CLI, follow these steps to install and configure it:

1. Go to the [AWS Command Line Interface User Guide](#).
2. Follow the instructions for [Installing the AWS CLI](#) and [Configuring the AWS CLI](#).

Using the AWS CLI with DynamoDB

The command line format consists of a DynamoDB operation name followed by the parameters for that operation. The AWS CLI supports a shorthand syntax for the parameter values, as well as JSON.

For example, the following command creates a table named *Music*. The partition key is *Artist*, and the sort key is *SongTitle*. (For easier readability, long commands in this section are broken into separate lines.)

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
    AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
  --key-schema AttributeName=Artist,KeyType=HASH
  AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1 \
  --table-class STANDARD
```

The following commands add new items to the table. These examples use a combination of shorthand syntax and JSON.

```
aws dynamodb put-item \
  --table-name Music \
  --item \
    '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"}, \
    "AlbumTitle": {"S": "Somewhat Famous"}}' \
  --return-consumed-capacity TOTAL

aws dynamodb put-item \
  --table-name Music \
```

```
--item '{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"},  
    "AlbumTitle": {"S": "Songs About Life"} }' \  
--return-consumed-capacity TOTAL
```

On the command line, it can be difficult to compose valid JSON. However, the AWS CLI can read JSON files. For example, consider the following JSON code snippet, which is stored in a file named *key-conditions.json*.

```
{  
    "Artist": {  
        "AttributeValueList": [  
            {  
                "S": "No One You Know"  
            }  
        ],  
        "ComparisonOperator": "EQ"  
    },  
    "SongTitle": {  
        "AttributeValueList": [  
            {  
                "S": "Call Me Today"  
            }  
        ],  
        "ComparisonOperator": "EQ"  
    }  
}
```

You can now issue a Query request using the AWS CLI. In this example, the contents of the *key-conditions.json* file are used for the --key-conditions parameter.

```
aws dynamodb query --table-name Music --key-conditions file://key-conditions.json
```

Using the AWS CLI with DynamoDB local

The AWS CLI can also interact with DynamoDB local (downloadable version) that runs on your computer. To enable this, add the following parameter to each command:

```
--endpoint-url http://localhost:8000
```

The following example uses the AWS CLI to list the tables in a local database.

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

If DynamoDB is using a port number other than the default (8000), modify the `--endpoint-url` value accordingly.

 **Note**

The AWS CLI can't use the DynamoDB local (downloadable version) as a default endpoint. Therefore, you must specify `--endpoint-url` with each command.

Using the API

You can use the AWS Management Console and the AWS Command Line Interface to work interactively with Amazon DynamoDB. However, to get the most out of DynamoDB, you can write application code using the AWS SDKs.

The AWS SDKs provide broad support for DynamoDB in [Java](#), [JavaScript in the browser](#), [.NET](#), [Node.js](#), [PHP](#), [Python](#), [Ruby](#), [C++](#), [Go](#), [Android](#), and [iOS](#). To get started quickly with these languages, see [Getting started with DynamoDB and the AWS SDKs](#).

Before you can use the AWS SDKs with DynamoDB, you must get an AWS access key ID and secret access key. For more information, see [Setting up DynamoDB \(web service\)](#).

For a high-level overview of DynamoDB application programming with the AWS SDKs, see [Programming with DynamoDB and the AWS SDKs](#).

Using the NoSQL workbench for DynamoDB

You can also access DynamoDB by downloading and using the [NoSQL Workbench for DynamoDB](#).

NoSQL Workbench for Amazon DynamoDB is a cross-platform, client-side GUI application that you can use for modern database development and operations. It's available for Windows, macOS, and Linux. NoSQL Workbench is a visual development tool that provides data modeling, data visualization, and query development features to help you design, create, query, and manage DynamoDB tables. NoSQL Workbench now includes DynamoDB local as an optional part of the installation process, which makes it easier to model your data in DynamoDB local. To learn more

about DynamoDB local and its requirements, see [Setting up DynamoDB local \(downloadable version\)](#).

Note

The NoSQL Workbench for DynamoDB currently doesn't support AWS logins that are configured with two-factor authentication (2FA).

Data modeling

With NoSQL Workbench for DynamoDB, you can build new data models from, or design models based on, existing data models that satisfy your application's data access patterns. You can also import and export the designed data model at the end of the process. For more information, see [Building data models with NoSQL Workbench](#).

Data visualization

The data model visualizer provides a canvas where you can map queries and visualize the access patterns (facets) of the application without having to write code. Every facet corresponds to a different access pattern in DynamoDB. You can autogenerate sample data for use in your data model. For more information, see [Visualizing data access patterns](#).

Operation building

NoSQL Workbench provides a rich graphical user interface for you to develop and test queries. You can use the *operation builder* to view, explore, and query live datasets. You can also use the structured operation builder to build and perform data plane operations. It supports projection and condition expression, and lets you generate sample code in multiple languages. For more information, see [Exploring datasets and building operations with NoSQL Workbench](#).

IP address ranges

Amazon Web Services (AWS) publishes its current IP address ranges in JSON format. To view the current ranges, download [ip-ranges.json](#). For more information, see [AWS IP address ranges](#) in the AWS General Reference.

To find the IP address ranges that you can use to [access to DynamoDB tables and indexes](#), search the ip-ranges.json file for the following string: "service": "DYNAMODB".

Note

The IP address ranges do not apply to DynamoDB Streams or DynamoDB Accelerator (DAX).

Getting started with DynamoDB

Use the hands-on tutorials in this section to help you get started and learn more about Amazon DynamoDB.

Topics

- [Basic concepts in DynamoDB](#)
- [Prerequisites - getting started tutorial](#)
- [Step 1: Create a table](#)
- [Step 2: Write data to a table using the console or AWS CLI](#)
- [Step 3: Read data from a table](#)
- [Step 4: Update data in a table](#)
- [Step 5: Query data in a table](#)
- [Step 6: Create a global secondary index](#)
- [Step 7: Query the global secondary index](#)
- [Step 8: \(Optional\) clean up resources](#)
- [Getting started with DynamoDB: Next steps](#)

Basic concepts in DynamoDB

Before you begin, you should familiarize yourself with the basic concepts in Amazon DynamoDB. For more information, see [DynamoDB core components](#).

Then continue on to [Prerequisites](#) to learn about setting up DynamoDB.

Prerequisites - getting started tutorial

Before starting the Amazon DynamoDB tutorial, follow the steps in [Setting up DynamoDB](#). Then continue on to [Step 1: Create a table](#).

Note

- If you plan to interact with DynamoDB only through the AWS Management Console, you don't need an AWS access key. Complete the steps in [Signing up for AWS](#), and then continue on to [Step 1: Create a table](#).
- If you don't want to sign up for a free tier account, you can set up [DynamoDB local \(downloadable version\)](#). Then continue on to [Step 1: Create a table](#).
- There are differences when working with CLI commands in terminals on Linux and Windows. The following guide presents commands formatted for Linux terminals (this includes macOS), and commands formatted for Windows CMD. Choose the command that best fits the terminal application you are using.

Step 1: Create a table

In this step, you create a Music table in Amazon DynamoDB. The table has the following details:

- Partition key — Artist
- Sort key — SongTitle

For more information about table operations, see [Working with tables and data in DynamoDB](#).

Note

Before you begin, make sure that you followed the steps in [Prerequisites - getting started tutorial](#).

AWS Management Console

To create a new Music table using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Dashboard**.
3. On the right side of the console, choose **Create Table**.

Create a table

Create an Amazon DynamoDB table for fast and predictable database performance at any scale.

[Learn more !\[\]\(f5f6fa381223eb14fa65a973ea1d362f_img.jpg\)](#)

Create table

4. Enter the table details as follows:
 - a. For the table name, enter **Music**.
 - b. For the partition key, enter **Artist**.
 - c. Enter **SongTitle** as the sort key.
 - d. Leave **Default settings** selected.
5. Choose **Create** to create the table.

Create table

Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.



1 to 255 characters and case sensitive.

Sort key - optional

You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.



1 to 255 characters and case sensitive.

Settings

Default settings

The fastest way to create your table. You can modify these settings now or after your table has been created.

Customize settings

Use these advanced features to make DynamoDB work better for your needs.

Default settings

Read/write capacity Info

Using provisioned capacity mode. Read and write capacity are set to 5 units each with auto scaling enabled.

Secondary indexes Info

No secondary indexes have been created. Queries will be run by using the table's partition key and sort key only.

Key management for encryption at rest Info

Using the AWS owned customer master key. This key is managed by DynamoDB at no extra cost.

Tags

Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

6. Once the table is in ACTIVE status, it's considered best practice to enable [Point-in-time recovery for DynamoDB](#) on the table by performing the following steps:

- a. Click the table name to open the table.
- b. Click **Backups**.
- c. Click the **Edit** button in the Point-in-time recovery section.
- d. Click the checkbox to turn on point-in-time recovery, then click the **Save changes** button.

Point-in-time recovery is now enabled for your newly created Music table.

AWS CLI

The following AWS CLI example creates a new Music table using `create-table`.

Linux

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
   AttributeName=Artist,AttributeType=S \
   AttributeName=SongTitle,AttributeType=S \
--key-schema \
   AttributeName=Artist,KeyType=HASH \
   AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
    ReadCapacityUnits=5,WriteCapacityUnits=5 \
--table-class STANDARD
```

Windows CMD

```
aws dynamodb create-table ^
--table-name Music ^
--attribute-definitions ^
   AttributeName=Artist,AttributeType=S ^
   AttributeName=SongTitle,AttributeType=S ^
--key-schema ^
   AttributeName=Artist,KeyType=HASH ^
   AttributeName=SongTitle,KeyType=RANGE ^
--provisioned-throughput ^
    ReadCapacityUnits=5,WriteCapacityUnits=5 ^
```

```
--table-class STANDARD
```

Using `create-table` returns the following sample result.

```
{  
    "TableDescription": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "Artist",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "AttributeType": "S"  
            }  
        ],  
        "TableName": "Music",  
        "KeySchema": [  
            {  
                "AttributeName": "Artist",  
                "KeyType": "HASH"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "KeyType": "RANGE"  
            }  
        ],  
        "TableStatus": "CREATING",  
        "CreationDateTime": "2023-03-29T12:11:43.379000-04:00",  
        "ProvisionedThroughput": {  
            "NumberOfDecreasesToday": 0,  
            "ReadCapacityUnits": 5,  
            "WriteCapacityUnits": 5  
        },  
        "TableSizeBytes": 0,  
        "ItemCount": 0,  
        "TableArn": "arn:aws:dynamodb:us-east-1:111122223333:table/Music",  
        "TableId": "60abf404-1839-4917-a89b-a8b0ab2a1b87",  
        "TableClassSummary": {  
            "TableClass": "STANDARD"  
        }  
    }  
}
```

```
}
```

Note that the value of the TableStatus field is set to CREATING.

To verify that DynamoDB has finished creating the Music table, use the describe-table command.

Linux

```
aws dynamodb describe-table --table-name Music | grep TableStatus
```

Windows CMD

```
aws dynamodb describe-table --table-name Music | findstr TableStatus
```

This command returns the following result. When DynamoDB finishes creating the table, the value of the TableStatus field is set to ACTIVE.

```
"TableStatus": "ACTIVE",
```

Once the table is in ACTIVE status, it's considered best practice to enable [Point-in-time recovery for DynamoDB](#) on the table by running the following command:

Linux

```
aws dynamodb update-continuous-backups \
  --table-name Music \
  --point-in-time-recovery-specification \
    PointInTimeRecoveryEnabled=true
```

Windows CMD

```
aws dynamodb update-continuous-backups --table-name Music --point-in-time-recovery-
specification PointInTimeRecoveryEnabled=true
```

This command returns the following result.

```
{  
    "ContinuousBackupsDescription": {  
        "ContinuousBackupsStatus": "ENABLED",  
        "PointInTimeRecoveryDescription": {  
            "PointInTimeRecoveryStatus": "ENABLED",  
            "EarliestRestorableDateTime": "2023-03-29T12:18:19-04:00",  
            "LatestRestorableDateTime": "2023-03-29T12:18:19-04:00"  
        }  
    }  
}
```

Note

There are cost implications to enabling continuous backups with point-in-time recovery. For more information about pricing, see [Amazon DynamoDB pricing](#).

After creating the new table, proceed to [Step 2: Write data to a table using the console or AWS CLI](#).

Step 2: Write data to a table using the console or AWS CLI

In this step, you insert several items into the Music table that you created in [Step 1: Create a table](#).

For more information about write operations, see [Writing an item](#).

AWS Management Console

Follow these steps to write data to the Music table using the DynamoDB console.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. In the table list, choose the **Music** table.

DynamoDB

Dashboard

Tables (1)

Items New

PartiQL editor New

Backups

Exports to S3 New

Reserved capacity

DynamoDB > Tables

Tables (1) Info

Find tables by table name

Any table tag

	Name ▲	Status	Partition key	Sort key
<input type="checkbox"/>	Music	Active	Artist (String)	SongTitle (String)

4. Select **Explore Table Items**.

Music

C Actions ▾ Explore table items

5. In the **Items** view, choose **Create item**.

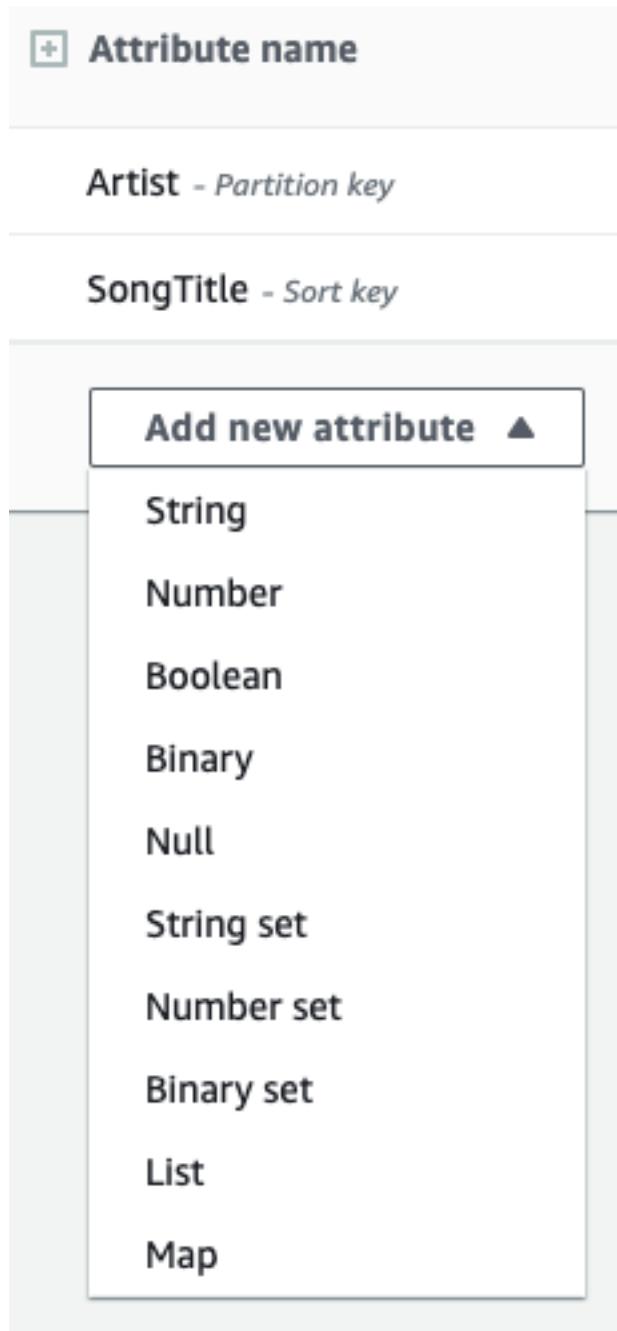
Items returned (0)

Actions ▾ Create item

Find items

< 1 > ⚙️ ✎

6. Choose **Add new attribute**, and then choose **Number**. Name the field **Awards**.



7. Repeat this process to create an **AlbumTitle** of type **String**.
8. Enter the following values for your item:
 - a. For **Artist**, enter **No One You Know** as the value.
 - b. For **SongTitle**, enter **Call Me Today**.
 - c. For **AlbumTitle**, enter **Somewhat Famous**.
 - d. For **Awards**, enter **1**.

9. Choose **Create item**.

Create item

Attributes

Attribute name	Value	Type
Artist - <i>Partition key</i>	No One You Know	String
SongTitle - <i>Sort key</i>	Call Me Today	String
Awards	1	Number
AlbumTitle	Somewhat Famous	String

Add new attribute ▾

Cancel **Create Item**

10. Repeat this process and create another item with the following values:

- For **Artist**, enter **Acme Band**.
- For **SongTitle** enter **Happy Day**.
- For **AlbumTitle**, enter **Songs About Life**.
- For **Awards**, enter **10**.

11. Do this one more time to create another item with the same **Artist** as the previous step, but different values for the other attributes:

- For **Artist**, enter **Acme Band**.
- For **SongTitle** enter **PartiQL Rocks**.
- For **AlbumTitle**, enter **Another Album Title**.
- For **Awards**, enter **8**.

AWS CLI

The following AWS CLI example creates several new items in the Music table. You can do this either through the DynamoDB API or [PartiQL](#), a SQL-compatible query language for DynamoDB.

DynamoDB API

Linux

```
aws dynamodb put-item \
--table-name Music \
--item \
'{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"}, \
"AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}'

aws dynamodb put-item \
--table-name Music \
--item \
'{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Howdy"}, \
"AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "2"}}'

aws dynamodb put-item \
--table-name Music \
--item \
'{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}, \
"AlbumTitle": {"S": "Songs About Life"}, "Awards": {"N": "10"}}'

aws dynamodb put-item \
--table-name Music \
--item \
'{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "PartiQL Rocks"}, \
"AlbumTitle": {"S": "Another Album Title"}, "Awards": {"N": "8"}}'
```

Windows CMD

```
aws dynamodb put-item ^
--table-name Music ^
--item ^
'{"\\"Artist\\": {\\"S\\": \"No One You Know\"}, \\"SongTitle\\": {\\"S\\": \"Call \
Me Today\"}, \\"AlbumTitle\\": {\\"S\\": \"Somewhat Famous\"}, \\"Awards\\": {\\"N\\": \
\"1\"}}'

aws dynamodb put-item ^
--table-name Music ^
--item ^
'{"\\"Artist\\": {\\"S\\": \"No One You Know\"}, \\"SongTitle\\": {\\"S\\": \"Howdy\"}, \
\\"AlbumTitle\\": {\\"S\\": \"Somewhat Famous\"}, \\"Awards\\": {\\"N\\": \"2\"}}'

aws dynamodb put-item ^
--table-name Music ^
--item ^
```

```
"{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day\"},  
\"AlbumTitle\": {\"S\": \"Songs About Life\"}, \"Awards\": {\"N\": \"10\"}}"  
  
aws dynamodb put-item ^  
  --table-name Music ^  
  --item ^  
    "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"PartiQL Rocks\"},  
\"AlbumTitle\": {\"S\": \"Another Album Title\"}, \"Awards\": {\"N\": \"8\"}}"
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "INSERT INTO Music \
    VALUE \
        {'Artist':'No One You Know','SongTitle':'Call Me Today',
'AlbumTitle':'Somewhat Famous', 'Awards':'1'}"

aws dynamodb execute-statement --statement "INSERT INTO Music \
    VALUE \
        {'Artist':'No One You Know','SongTitle':'Howdy',
'AlbumTitle':'Somewhat Famous', 'Awards':'2'}"

aws dynamodb execute-statement --statement "INSERT INTO Music \
    VALUE \
        {'Artist':'Acme Band','SongTitle':'Happy Day', 'AlbumTitle':'Songs
About Life', 'Awards':'10'}"

aws dynamodb execute-statement --statement "INSERT INTO Music \
    VALUE \
        {'Artist':'Acme Band','SongTitle':'PartiQL Rocks',
'AlbumTitle':'Another Album Title', 'Awards':'8'}"
```

Windows CMD

```
aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'No One You Know', 'SongTitle':'Call Me Today', 'AlbumTitle':'Somewhat Famous', 'Awards':'1'}"  
  
aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'No One You Know', 'SongTitle':'Howdy', 'AlbumTitle':'Somewhat Famous', 'Awards':'2'}"
```

```
aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'Acme Band', 'SongTitle':'Happy Day', 'AlbumTitle':'Songs About Life', 'Awards':'10'}"  
  
aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'Acme Band', 'SongTitle':'PartiQL Rocks', 'AlbumTitle':'Another Album Title', 'Awards':'8'}"
```

For more information about writing data with PartiQL, see [PartiQL insert statements](#).

For more information about supported data types in DynamoDB, see [Data types](#).

For more information about how to represent DynamoDB data types in JSON, see [Attribute values](#).

After writing data to your table, proceed to [Step 3: Read data from a table](#).

Step 3: Read data from a table

In this step, you will read back an item that was created in [Step 2: Write data to a table using the console or AWS CLI](#). You can use the DynamoDB console or the AWS CLI to read an item from the Music table by specifying Artist and SongTitle.

For more information about read operations in DynamoDB, see [Reading an item](#).

AWS Management Console

Follow these steps to read data from the Music table using the DynamoDB console.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Select the **Explore table items**.
5. On the **Items** tab, view the list of items stored in the table, sorted by Artist and SongTitle. The first item in the list is the one with the Artist **Acme Band** and the SongTitle **Happy Day**.

The screenshot shows the Amazon DynamoDB Items page. At the top, there's a breadcrumb navigation: DynamoDB > Items. To the right of the breadcrumb is an "Autopreview" button with a blue circular icon. Below the navigation, the word "Items" is followed by an "Info" link. On the left, there's a sidebar with a back arrow and a dropdown menu showing "Music". The main area has tabs for "Query" (which is selected) and "Scan". A "View table details" button is in the top right corner. The "Table or index" dropdown is set to "Music". Under "Artist (Partition key)", there's a text input field with placeholder text "Enter partition key value". Under "SongTitle (Sort key)", there's a dropdown menu set to "Equal to" with a "▼" arrow, an input field for "Enter sort key value", and a checkbox for "Sort descending". Below these fields is a "▶ Filters" button. At the bottom of the query section are "Run" and "Reset" buttons. A green checkmark icon indicates the query is completed with "Read capacity units consumed: 0.5". The results section is titled "Items returned (2)" with "Actions" and "Create item" buttons. It includes a search bar with "Find items" placeholder text and navigation controls for pages 1, 2, and 3. The table headers are "Artist", "SongTitle", "AlbumTitle", and "Awards". The data rows are:

	Artist	SongTitle	AlbumTitle	Awards
<input type="checkbox"/>	Acme Band	Happy Day	Songs Abou...	10
<input type="checkbox"/>	No One You...	Call Me Today	Somewhat ...	1

AWS CLI

The following AWS CLI example reads an item from the Music. You can do this either through the DynamoDB API or [PartiQL](#), a SQL-compatible query language for DynamoDB.

DynamoDB API

Note

The default behavior for DynamoDB is eventually consistent reads. The `consistent-read` parameter is used below to demonstrate strongly consistent reads.

Linux

```
aws dynamodb get-item --consistent-read \
--table-name Music \
--key '{ "Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"} }'
```

Windows CMD

```
aws dynamodb get-item --consistent-read ^
--table-name Music ^
--key "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day\"}}"
```

Using `get-item` returns the following sample result.

```
{
  "Item": {
    "AlbumTitle": {
      "S": "Songs About Life"
    },
    "Awards": {
      "S": "10"
    },
    "Artist": {
      "S": "Acme Band"
    },
    "SongTitle": {
      "S": "Happy Day"
    }
  }
}
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM Music \n WHERE Artist='Acme Band' AND SongTitle='Happy Day'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM Music WHERE Artist='Acme\nBand' AND SongTitle='Happy Day'"
```

Using the PartiQL Select statement returns the following sample result.

```
{\n    "Items": [\n        {\n            "AlbumTitle": {\n                "S": "Songs About Life"\n            },\n            "Awards": {\n                "S": "10"\n            },\n            "Artist": {\n                "S": "Acme Band"\n            },\n            "SongTitle": {\n                "S": "Happy Day"\n            }\n        }\n    ]\n}
```

For more information about reading data with PartiQL, see [PartiQL select statements](#).

To update the data in your table, proceed to [Step 4: Update data in a table](#).

Step 4: Update data in a table

In this step, you update an item that you created in [Step 2: Write data to a table using the console or AWS CLI](#). You can use the DynamoDB console or the AWS CLI to update the **AlbumTitle** of an item in the **Music** table by specifying **Artist**, **SongTitle**, and the updated **AlbumTitle**.

For more information about write operations, see [Writing an item](#).

AWS Management Console

You can use the DynamoDB console to update data in the **Music** table.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose **View items**.
5. Choose the item whose **Artist** value is **Acme Band** and **SongTitle** value is **Happy Day**.
6. Update the **AlbumTitle** value to **Updated Album Title**, and then choose **Save**.

The following image shows the updated item on the console.

Item editor

Form JSON

Attributes		Type
<input type="button" value="Attribute name"/> Artist - Partition key	Acme Band	New String
<input type="button" value="SongTitle - Sort key"/>	Happy Day	New String
<input type="button" value="AlbumTitle"/>	Updated Album Title	String <input type="button" value="Remove"/>
<input type="button" value="Awards"/>	10	Number <input type="button" value="Remove"/>
<input type="button" value="Add new attribute ▾"/>		
		<input type="button" value="Cancel"/> <input type="button" value="Save changes"/>

AWS CLI

The following AWS CLI example updates an item in the Music table. You can do this either through the DynamoDB API or [PartiQL](#), a SQL-compatible query language for DynamoDB.

DynamoDB API

Linux

```
aws dynamodb update-item \  
  --table-name Music \  
  --key '{ "Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"} }' \  
  --update-expression "SET AlbumTitle = :newval" \  
  --expression-attribute-values '{":newval":{"S":"Updated Album Title"}}' \  
  --return-values ALL_NEW
```

Windows CMD

```
aws dynamodb update-item ^  
  --table-name Music ^  
  --key "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day\"}}" ^  
  --update-expression "SET AlbumTitle = :newval" ^  
  --expression-attribute-values "{\":newval\":{\"S\":\"Updated Album Title\"}}" ^  
  --return-values ALL_NEW
```

Using `update-item` returns the following sample result because `return-values ALL_NEW` was specified.

```
{  
  "Attributes": {  
    "AlbumTitle": {  
      "S": "Updated Album Title"  
    },  
    "Awards": {  
      "S": "10"  
    },  
    "Artist": {  
      "S": "Acme Band"  
    },  
    "SongTitle": {  
      "S": "Happy Day"  
    }  
  }  
}
```

```
        "S": "Happy Day"
    }
}
}
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "UPDATE Music \
SET AlbumTitle='Updated Album Title' \
WHERE Artist='Acme Band' AND SongTitle='Happy Day' \
RETURNING ALL NEW *"
```

Windows CMD

```
aws dynamodb execute-statement --statement "UPDATE Music SET AlbumTitle='Updated
Album Title' WHERE Artist='Acme Band' AND SongTitle='Happy Day' RETURNING ALL NEW
*"
```

Using the Update statement returns the following sample result because RETURNING ALL NEW * was specified.

```
{
  "Items": [
    {
      "AlbumTitle": {
        "S": "Updated Album Title"
      },
      "Awards": {
        "S": "10"
      },
      "Artist": {
        "S": "Acme Band"
      },
      "SongTitle": {
        "S": "Happy Day"
      }
    }
  ]
}
```

For more information about updating data with PartiQL, see [PartiQL update statements](#).

To query the data in the Music table, proceed to [Step 5: Query data in a table](#).

Step 5: Query data in a table

In this step, you query the data that you wrote to the Music table in [the section called "Step 2: Write data"](#) by specifying Artist. This will display all songs that are associated with the partition key: Artist.

For more information about query operations, see [Query operations in DynamoDB](#).

AWS Management Console

Follow these steps to use the DynamoDB console to query data in the Music table.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose **Explore table items**.
5. Choose **Query**.
6. For **Partition key**, enter **Acme Band**, and then choose **Run**.

The screenshot shows the 'Items' page in the Amazon DynamoDB console. At the top, there's a breadcrumb navigation: 'DynamoDB > Items'. On the right, there's an 'Autopreview' button with a blue circular icon. Below the navigation, the table name 'Music' is selected under 'Table or index'. The 'Artist (Partition key)' is set to 'Acme Band'. Under 'SongTitle (Sort key)', there's a dropdown set to 'Equal to' with the value 'Enter sort key value'. A checkbox for 'Sort descending' is unchecked. At the bottom, there are 'Run' and 'Reset' buttons.

AWS CLI

The following AWS CLI example queries an item in the `Music` table. You can do this either through the DynamoDB API or [PartiQL](#), a SQL-compatible query language for DynamoDB.

DynamoDB API

You query an item through the DynamoDB API by using `query` and providing the partition key.

Linux

```
aws dynamodb query \
--table-name Music \
--key-condition-expression "Artist = :name" \
--expression-attribute-values '{":name":{"S":"Acme Band"}}'
```

Windows CMD

```
aws dynamodb query ^
--table-name Music ^
--key-condition-expression "Artist = :name" ^
--expression-attribute-values "{\":name\":{\"S\":\"Acme Band\"}}"
```

Using query returns all the songs associated with this particular Artist.

```
{
  "Items": [
    {
      "AlbumTitle": {
        "S": "Updated Album Title"
      },
      "Awards": {
        "N": "10"
      },
      "Artist": {
        "S": "Acme Band"
      },
      "SongTitle": {
        "S": "Happy Day"
      }
    },
    {
      "AlbumTitle": {
        "S": "Another Album Title"
      },
      "Awards": {
        "N": "8"
      },
      "Artist": {
        "S": "Acme Band"
      },
      "SongTitle": {
        "S": "PartiQL Rocks"
      }
    }
  ],
  "Count": 2,
  "ScannedCount": 2,
  "ConsumedCapacity": null
}
```

PartiQL for DynamoDB

You query an item through PartiQL by using the `Select` statement and providing the partition key.

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM Music \
WHERE Artist='Acme Band'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM Music WHERE Artist='Acme
Band'"
```

Using the `Select` statement in this way returns all the songs associated with this particular Artist.

```
{
  "Items": [
    {
      "AlbumTitle": {
        "S": "Updated Album Title"
      },
      "Awards": {
        "S": "10"
      },
      "Artist": {
        "S": "Acme Band"
      },
      "SongTitle": {
        "S": "Happy Day"
      }
    },
    {
      "AlbumTitle": {
        "S": "Another Album Title"
      },
      "Awards": {
        "S": "8"
      },
      "Artist": {
```

```
        "S": "Acme Band"
    },
    "SongTitle": {
        "S": "PartiQL Rocks"
    }
}
}
```

For more information about querying data with PartiQL, see [PartiQL select statements](#).

To create a global secondary index for your table, proceed to [Step 6: Create a global secondary index](#).

Step 6: Create a global secondary index

In this step, you create a global secondary index for the Music table that you created in [Step 1: Create a table](#).

For more information about global secondary indexes, see [Using Global Secondary Indexes in DynamoDB](#).

AWS Management Console

To use the Amazon DynamoDB console to create a global secondary index AlbumTitle-index for the Music table:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Indexes** tab for the Music table.
5. Choose **Create index**.



6. For the **Partition key**, enter **AlbumTitle**.
7. For **Index name**, enter **AlbumTitle-index**.
8. Leave the other settings on their default values and choose **Create index**.

Create global secondary index [Info](#)

Global secondary indexes allow you to perform queries on attributes that are not part of a table's primary key. Note that global secondary index read and write capacity settings are separate from those of the table, and they will incur additional costs.

Index details [Info](#)

Partition key Data type

1 to 255 characters.

Sort key - optional Data type

1 to 255 characters.

Index name

Between 3 and 255 characters. Only A–Z, a–z, 0–9, underscore characters, hyphens, and periods allowed.

Index capacity [Info](#)

Read capacity

Read capacity settings

- Copy from base table
- Customize settings

Auto scaling [Info](#)

Dynamically adjusts provisioned throughput capacity on your behalf in response to actual traffic patterns.

- On
- Off

Minimum capacity units

Maximum capacity units

Target utilization (%)

Write capacity

Write capacity settings

- Copy from base table

Auto scaling [Info](#)

Dynamically adjusts provisioned throughput capacity on your behalf in response to actual traffic patterns.

- On

AWS CLI

The following AWS CLI example creates a global secondary index `AlbumTitle-index` for the `Music` table using `update-table`.

Linux

```
aws dynamodb update-table \
--table-name Music \
--attribute-definitions AttributeName=AlbumTitle,AttributeType=S \
--global-secondary-index-updates \
"[{\\"Create\\":{\\\"IndexName\\\": \\"AlbumTitle-index\\\",\\\"KeySchema\\\": \
[{\\"AttributeName\\\":\\\"AlbumTitle\\\",\\\"KeyType\\\":\\\"HASH\\\"}], \
\\\"ProvisionedThroughput\\\": {\\\"ReadCapacityUnits\\\": 10, \\\"WriteCapacityUnits\\\": \
5},\\\"Projection\\\":{\\\"ProjectionType\\\":\\\"ALL\\\"}}}]"
```

Windows CMD

```
aws dynamodb update-table ^
--table-name Music ^
--attribute-definitions AttributeName=AlbumTitle,AttributeType=S ^
--global-secondary-index-updates "[{\\"Create\\":{\\\"IndexName\\\": \\"AlbumTitle-
index\\\",\\\"KeySchema\\\": [{\\\"AttributeName\\\":\\\"AlbumTitle\\\",\\\"KeyType\\\":\\\"HASH\\\"}], \
\\\"ProvisionedThroughput\\\": {\\\"ReadCapacityUnits\\\": 10, \\\"WriteCapacityUnits\\\": 5}, \
\\\"Projection\\\":{\\\"ProjectionType\\\":\\\"ALL\\\"}}}]"
```

Using `update-table` returns the following sample result.

```
{
  "TableDescription": {
    "TableArn": "arn:aws:dynamodb:us-west-2:111122223333:table/Music",
    "AttributeDefinitions": [
      {
        "AttributeName": "AlbumTitle",
        "AttributeType": "S"
      },
      {
        "AttributeName": "Artist",
        "AttributeType": "S"
      },
      {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
      }
    ],
    "IndexDefinitions": [
      {
        "IndexName": "AlbumTitle-index",
        "KeySchema": [
          {
            "AttributeName": "AlbumTitle",
            "KeyType": "HASH"
          }
        ],
        "ProjectionType": "ALL"
      }
    ],
    "TableStatus": "ACTIVE",
    "ItemCount": 0,
    "TableSizeBytes": 0
  }
}
```

```
        },
    ],
    "GlobalSecondaryIndexes": [
        {
            "IndexSizeBytes": 0,
            "IndexName": "AlbumTitle-index",
            "Projection": {
                "ProjectionType": "ALL"
            },
            "ProvisionedThroughput": {
                "NumberOfDecreasesToday": 0,
                "WriteCapacityUnits": 5,
                "ReadCapacityUnits": 10
            },
            "IndexStatus": "CREATING",
            "Backfilling": false,
            "KeySchema": [
                {
                    "KeyType": "HASH",
                    "AttributeName": "AlbumTitle"
                }
            ],
            "IndexArn": "arn:aws:dynamodb:us-west-2:111122223333:table/Music/index/AlbumTitle-index",
            "ItemCount": 0
        }
    ],
    "ProvisionedThroughput": {
        "NumberOfDecreasesToday": 0,
        "WriteCapacityUnits": 5,
        "ReadCapacityUnits": 10
    },
    "TableSizeBytes": 0,
    "TableName": "Music",
    "TableStatus": "UPDATING",
    "TableId": "a04b7240-0a46-435b-a231-b54091ab1017",
    "KeySchema": [
        {
            "KeyType": "HASH",
            "AttributeName": "Artist"
        },
        {
            "KeyType": "RANGE",
            "AttributeName": "SongTitle"
        }
    ]
}
```

```
        },
    ],
    "ItemCount": 0,
    "CreationDateTime": 1558028402.69
}
}
```

Note that the value of the IndexStatus field is set to CREATING.

To verify that DynamoDB has finished creating the AlbumTitle-index global secondary index, use the describe-table command.

Linux

```
aws dynamodb describe-table --table-name Music | grep IndexStatus
```

Windows CMD

```
aws dynamodb describe-table --table-name Music | findstr IndexStatus
```

This command returns the following result. The index is ready for use when the value of the IndexStatus field returned is set to ACTIVE.

```
"IndexStatus": "ACTIVE",
```

Next, you can query the global secondary index. For details, see [Step 7: Query the global secondary index](#).

Step 7: Query the global secondary index

In this step, you query a global secondary index on the Music table using the Amazon DynamoDB console or the AWS CLI.

For more information about global secondary indexes, see [Using Global Secondary Indexes in DynamoDB](#).

AWS Management Console

Follow these steps to use the DynamoDB console to query data through the AlbumTitle-index global secondary index.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Select the **View items**.
5. Choose **Query**.
6. In the drop-down list under **Query**, choose **AlbumTitle-index**.

For **AlbumTitle**, enter **Somewhat Famous**, and then choose **Run**.

The screenshot shows the 'Items' page for the 'Music' table. At the top, there's a 'Query' button highlighted in grey, indicating it's selected. Below it is a dropdown menu labeled 'Table or index' containing the following options:

- AlbumTitle-index (highlighted with a blue border)
- Table
- Music
- Index
- AlbumTitle-index (highlighted with a blue border)

AWS CLI

The following AWS CLI example queries a global secondary index `AlbumTitle-index` on the `Music` table. You can do this either through the DynamoDB API or [PartiQL](#), a SQL-compatible query language for DynamoDB.

DynamoDB API

You query the global secondary index through the DynamoDB API by using `query` and providing the index name.

Linux

```
aws dynamodb query \
--table-name Music \
--index-name AlbumTitle-index \
```

```
--key-condition-expression "AlbumTitle = :name" \
--expression-attribute-values  '{"":name":{"S":"Somewhat Famous"}}'
```

Windows CMD

```
aws dynamodb query ^
--table-name Music ^
--index-name AlbumTitle-index ^
--key-condition-expression "AlbumTitle = :name" ^
--expression-attribute-values  "{\"":name\":{\"S\":\"Somewhat Famous\"}}"
```

Using query returns the following sample result.

```
{
  "Items": [
    {
      "AlbumTitle": {
        "S": "Somewhat Famous"
      },
      "Awards": {
        "S": "1"
      },
      "Artist": {
        "S": "No One You Know"
      },
      "SongTitle": {
        "S": "Call Me Today"
      }
    },
    {
      "AlbumTitle": {
        "S": "Somewhat Famous"
      },
      "Awards": {
        "N": "2"
      },
      "Artist": {
        "S": "No One You Know"
      },
      "SongTitle": {
        "S": "Howdy"
      }
    }
  ]
}
```

```
],  
"Count": 2,  
"ScannedCount": 2,  
"ConsumedCapacity": null  
}
```

PartiQL for DynamoDB

You query the global secondary index through PartiQL by using the `Select` statement and providing the index name.

Note

You'll need to escape the double quotes around `Music` and `AlbumTitle-index` since you're doing this through the CLI.

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM \"Music\".\"AlbumTitle-  
index\"  \  
WHERE AlbumTitle='Somewhat Famous'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM \"Music\".\"AlbumTitle-  
index\" WHERE AlbumTitle='Somewhat Famous'"
```

Using the `Select` statement in this way returns the following sample result.

```
{  
    "Items": [  
        {  
            "AlbumTitle": {  
                "S": "Somewhat Famous"  
            },  
            "Awards": {  
                "S": "1"  
            },  
            "Artist": {  
                "S": "The Artist"  
            }  
        }  
    ]  
}
```

```
        "S": "No One You Know"
    },
    "SongTitle": {
        "S": "Call Me Today"
    }
},
{
    "AlbumTitle": {
        "S": "Somewhat Famous"
    },
    "Awards": {
        "S": "2"
    },
    "Artist": {
        "S": "No One You Know"
    },
    "SongTitle": {
        "S": "Howdy"
    }
}
]
```

For more information about querying data with PartiQL, see [PartiQL select statements](#).

Step 8: (Optional) clean up resources

If you no longer need the Amazon DynamoDB table that you created for the tutorial, you can delete it. This step helps ensure that you aren't charged for resources that you aren't using. You can use the DynamoDB console or the AWS CLI to delete the Music table that you created in [Step 1: Create a table](#).

For more information about table operations in DynamoDB, see [Working with tables and data in DynamoDB](#).

AWS Management Console

To delete the Music table using the console:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.

3. Choose the **Music** table from the table list.
4. Choose the **Indexes** tab for the Music table.
5. Choose **Delete table**.

The screenshot shows the 'Actions' dropdown menu for the 'Music' table. The menu items are: Actions ▲, View items, Edit capacity, Create index, Create access control policy, and Delete table. The 'Delete table' option is circled in red.

AWS CLI

The following AWS CLI example deletes the `Music` table using `delete-table`.

```
aws dynamodb delete-table --table-name Music
```

Getting started with DynamoDB: Next steps

For more information about using Amazon DynamoDB, see the following topics:

- [Working with tables and data in DynamoDB](#)
- [Working with items and attributes](#)
- [Query operations in DynamoDB](#)
- [Using Global Secondary Indexes in DynamoDB](#)
- [Working with transactions](#)
- [In-memory acceleration with DynamoDB Accelerator \(DAX\)](#)
- [Getting started with DynamoDB and the AWS SDKs](#)
- [Programming with DynamoDB and the AWS SDKs](#)

Getting started with DynamoDB and the AWS SDKs

Use the hands-on tutorials in this section to get started with Amazon DynamoDB and the AWS SDKs. You can run the code examples on either the downloadable version of DynamoDB or the DynamoDB web service.

Topics

- [Create a DynamoDB table](#)
- [Write an item to a DynamoDB table](#)
- [Read an item from a DynamoDB table](#)
- [Update an item in a DynamoDB table](#)
- [Delete an item in a DynamoDB table](#)
- [Query a DynamoDB table](#)
- [Scan a DynamoDB table](#)
- [Using DynamoDB with an AWS SDK](#)

Create a DynamoDB table

You can create a table using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on tables, see [Core components of Amazon DynamoDB](#).

Create a DynamoDB table using an AWS SDK

The following code examples show how to create a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Creates a new Amazon DynamoDB table and then waits for the new
/// table to become active.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="tableName">The name of the table to create.</param>
/// <returns>A Boolean value indicating the success of the operation.</returns>
public static async Task<bool> CreateMovieTableAsync(AmazonDynamoDBClient
client, string tableName)
{
    var response = await client.CreateTableAsync(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "title",
                AttributeType = ScalarAttributeType.S,
            },
            new AttributeDefinition
            {
                AttributeName = "year",
                AttributeType = ScalarAttributeType.N,
            },
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement
            {
                AttributeName = "year",
                KeyType = KeyType.HASH,
            },
            new KeySchemaElement
            {
                AttributeName = "title",
                KeyType = KeyType.RANGE,
            },
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 5,
```

```
        WriteCapacityUnits = 5,
    },
});

// Wait until the table is ACTIVE and then report success.
Console.WriteLine("Waiting for table to become active...");

var request = new DescribeTableRequest
{
    TableName = response.TableDescription.TableName,
};

TableStatus status;

int sleepDuration = 2000;

do
{
    System.Threading.Thread.Sleep(sleepDuration);

    var describeTableResponse = await
client.DescribeTableAsync(request);
    status = describeTableResponse.Table.TableStatus;

    Console.WriteLine(".");
}
while (status != "ACTIVE");

return status == TableStatus.ACTIVE;
}
```

- For API details, see [CreateTable](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#   -n table_name -- The name of the table to create.
#   -a attribute_definitions -- JSON file path of a list of attributes and
#     their types.
#   -k key_schema -- JSON file path of a list of attributes and their key
#     types.
#   -p provisioned_throughput -- Provisioned throughput settings for the
#     table.
#
# Returns:
#   0 - If successful.
#   1 - If it fails.
#####
function dynamodb_create_table() {
    local table_name attribute_definitions key_schema provisioned_throughput
    response
    local option OPTARG # Required to use getopt command in a function.

    #####
    # Function usage explanation
    #####
    function usage() {
        echo "function dynamodb_create_table"
        echo "Creates an Amazon DynamoDB table."
        echo " -n table_name -- The name of the table to create."
        echo " -a attribute_definitions -- JSON file path of a list of attributes and
        their types."
    }
}
```

```
echo " -k key_schema -- JSON file path of a list of attributes and their key
types."
echo " -p provisioned_throughput -- Provisioned throughput settings for the
table."
echo ""
}

# Retrieve the calling parameters.
while getopts "n:a:k:p:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        a) attribute_definitions="${OPTARG}" ;;
        k) key_schema="${OPTARG}" ;;
        p) provisioned_throughput="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) 
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$attribute_definitions" ]]; then
    errecho "ERROR: You must provide an attribute definitions json file path the
-a parameter."
    usage
    return 1
fi

if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
parameter."
    usage
```

```
        return 1
    fi

    if [[ -z "$provisioned_throughput" ]]; then
        errecho "ERROR: You must provide a provisioned throughput json file path the
-p parameter."
        usage
        return 1
    fi

    iecho "Parameters:\n"
    iecho "    table_name:      $table_name"
    iecho "    attribute_definitions:  $attribute_definitions"
    iecho "    key_schema:      $key_schema"
    iecho "    provisioned_throughput:   $provisioned_throughput"
    iecho ""

response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --key-schema file://"$key_schema" \
    --provisioned-throughput "$provisioned_throughput")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
fi

return 0
}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
```

```
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#       $1 - The error code returned by the AWS CLI.
#
# Returns:
#       0: - Success.
#
#####

function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
```

```
    errecho "  The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho "  255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [CreateTable in AWS CLI Command Reference](#).

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Create an Amazon DynamoDB table.
/*
 * \sa createTable()
 * \param tableName: Name for the DynamoDB table.
 * \param primaryKey: Primary key for the DynamoDB table.
 * \param clientConfiguration: AWS client configuration.
 * \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createTable(const Aws::String &tableName,
                                    const Aws::String &primaryKey,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::cout << "Creating table " << tableName <<
        " with a simple primary key: \" " << primaryKey << "\"." <<
        std::endl;

    Aws::DynamoDB::Model::CreateTableRequest request;

    Aws::DynamoDB::Model::AttributeDefinition hashKey;
```

```
hashKey.SetAttributeName(primaryKey);
hashKey.SetAttributeType(Aws::DynamoDB::Model::ScalarAttributeType::S);
request.AddAttributeDefinitions(hashKey);

Aws::DynamoDB::Model::KeySchemaElement keySchemaElement;
keySchemaElement.WithAttributeName(primaryKey).WithKeyType(
    Aws::DynamoDB::Model::KeyType::HASH);
request.AddKeySchema(keySchemaElement);

Aws::DynamoDB::Model::ProvisionedThroughput throughput;
throughput.WithReadCapacityUnits(5).WithWriteCapacityUnits(5);
request.SetProvisionedThroughput(throughput);
request.SetTableName(tableName);

const Aws::DynamoDB::Model::CreateTableOutcome &outcome =
dynamoClient.CreateTable(
    request);
if (outcome.IsSuccess()) {
    std::cout << "Table \""
        << outcome.GetResult().GetTableDescription().GetTableName() <<
    " created!" << std::endl;
}
else {
    std::cerr << "Failed to create table: " <<
outcome.GetError().GetMessage()
        << std::endl;
}

return outcome.IsSuccess();
}
```

- For API details, see [CreateTable](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To create a table with tags

The following `create-table` example uses the specified attributes and key schema to create a table named `MusicCollection`. This table uses provisioned throughput and is

encrypted at rest using the default AWS owned CMK. The command also applies a tag to the table, with a key of Owner and a value of blueTeam.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

Output:

```
{
  "TableDescription": {
    "AttributeDefinitions": [
      {
        "AttributeName": "Artist",
        "AttributeType": "S"
      },
      {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
      }
    ],
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "WriteCapacityUnits": 5,
      "ReadCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "TableName": "MusicCollection",
    "TableStatus": "CREATING",
    "KeySchema": [
      {
        "KeyType": "HASH",
        "AttributeName": "Artist"
      },
      {
        "KeyType": "RANGE",
        "AttributeName": "SongTitle"
      }
    ]
  }
}
```

```
        ],
        "ItemCount": 0,
        "CreationDateTime": "2020-05-26T16:04:41.627000-07:00",
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    }
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To create a table in On-Demand Mode

The following example creates a table called MusicCollection using on-demand mode, rather than provisioned throughput mode. This is useful for tables with unpredictable workloads.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {

```

```
        "AttributeName": "Artist",
        "KeyType": "HASH"
    },
    {
        "AttributeName": "SongTitle",
        "KeyType": "RANGE"
    }
],
"TableStatus": "CREATING",
"CreationDateTime": "2020-05-27T11:44:10.807000-07:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 0,
    "WriteCapacityUnits": 0
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"BillingModeSummary": {
    "BillingMode": "PAY_PER_REQUEST"
}
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To create a table and encrypt it with a Customer Managed CMK

The following example creates a table named MusicCollection and encrypts it using a customer managed CMK.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-a123-ab1234a1b234
```

Output:

```
{  
    "TableDescription": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "Artist",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "AttributeType": "S"  
            }  
        ],  
        "TableName": "MusicCollection",  
        "KeySchema": [  
            {  
                "AttributeName": "Artist",  
                "KeyType": "HASH"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "KeyType": "RANGE"  
            }  
        ],  
        "TableStatus": "CREATING",  
        "CreationDateTime": "2020-05-27T11:12:16.431000-07:00",  
        "ProvisionedThroughput": {  
            "NumberOfDecreasesToday": 0,  
            "ReadCapacityUnits": 5,  
            "WriteCapacityUnits": 5  
        },  
        "TableSizeBytes": 0,  
        "ItemCount": 0,  
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/  
MusicCollection",  
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",  
        "SSEDescription": {  
            "Status": "ENABLED",  
            "SSEType": "KMS",  
            "KMSMasterKeyArn": "arn:aws:kms:us-west-2:123456789012:key/abcd1234-  
abcd-1234-a123-ab1234a1b234"  
        }  
    }  
}
```

{}

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 4: To create a table with a Local Secondary Index

The following example uses the specified attributes and key schema to create a table named MusicCollection with a Local Secondary Index named AlbumTitleIndex.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S
    AttributeName=SongTitle,AttributeType=S AttributeName=AlbumTitle,AttributeType=S
\
    --key-schema AttributeName=Artist,KeyType=HASH
    AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
    "[
        {
            \"IndexName\": \"AlbumTitleIndex\",
            \"KeySchema\": [
                {\"AttributeName\": \"Artist\", \"KeyType\": \"HASH\"},
                {\"AttributeName\": \"AlbumTitle\", \"KeyType\": \"RANGE\"}
            ],
            \"Projection\": {
                \"ProjectionType\": \"INCLUDE\",
                \"NonKeyAttributes\": [\"Genre\", \"Year\"]"
            }
        }
    ]"
```

Output:

```
{ "TableDescription": { "AttributeDefinitions": [ { "AttributeName": "AlbumTitle", "AttributeType": "S" }, { "AttributeName": "SongTitle", "AttributeType": "S" } ], "IndexDescriptions": [ { "IndexName": "AlbumTitleIndex", "KeySchema": [ { "AttributeName": "Artist", "KeyType": "HASH" }, { "AttributeName": "AlbumTitle", "KeyType": "RANGE" } ], "Projection": { "ProjectionType": "INCLUDE", "NonKeyAttributes": [ "Genre", "Year" ] } } ], "TableStatus": "ACTIVE", "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection", "ItemCount": 0, "TableSizeBytes": 0, "BillingMode": "PAY_PER_REQUEST" } }
```

```
        "AttributeName": "Artist",
        "AttributeType": "S"
    },
    {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
    }
],
"TableName": "MusicCollection",
"KeySchema": [
    {
        "AttributeName": "Artist",
        "KeyType": "HASH"
    },
    {
        "AttributeName": "SongTitle",
        "KeyType": "RANGE"
    }
],
"TableStatus": "CREATING",
"CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"LocalSecondaryIndexes": [
    {
        "IndexName": "AlbumTitleIndex",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "AlbumTitle",
                "KeyType": "RANGE"
            }
        ],
    }
],
```

```
        "Projection": {
            "ProjectionType": "INCLUDE",
            "NonKeyAttributes": [
                "Genre",
                "Year"
            ]
        },
        "IndexSizeBytes": 0,
        "ItemCount": 0,
        "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/index/AlbumTitleIndex"
    }
]
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 5: To create a table with a Global Secondary Index

The following example creates a table named GameScores with a Global Secondary Index called GameTitleIndex. The base table has a partition key of UserId and a sort key of GameTitle, allowing you to find an individual user's best score for a specific game efficiently, whereas the GSI has a partition key of GameTitle and a sort key of TopScore, allowing you to quickly find the overall highest score for a particular game.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S \
    AttributeName=GameTitle,AttributeType=S AttributeName=TopScore,AttributeType=N \
    --key-schema AttributeName=UserId,KeyType=HASH \
        AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
    "[
        {
            \"IndexName\": \"GameTitleIndex\",
            \"KeySchema\": [
                {\"AttributeName\": \"GameTitle\", \"KeyType\": \"HASH\"},
                {\"AttributeName\": \"TopScore\", \"KeyType\": \"RANGE\"}
            ],
            \"Projection\": {
```

```
\\"ProjectionType\\":\\"INCLUDE\\",
\\"NonKeyAttributes\\":[\\"UserId\\"]
},
\\"ProvisionedThroughput\\": {
    \\"ReadCapacityUnits\\": 10,
    \\"WriteCapacityUnits\\": 5
}
}"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T17:28:15.602000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
        }
    }
}
```

```
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
    "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "GlobalSecondaryIndexes": [
        {
            "IndexName": "GameTitleIndex",
            "KeySchema": [
                {
                    "AttributeName": "GameTitle",
                    "KeyType": "HASH"
                },
                {
                    "AttributeName": "TopScore",
                    "KeyType": "RANGE"
                }
            ],
            "Projection": {
                "ProjectionType": "INCLUDE",
                "NonKeyAttributes": [
                    "UserId"
                ]
            },
            "IndexStatus": "CREATING",
            "ProvisionedThroughput": {
                "NumberOfDecreasesToday": 0,
                "ReadCapacityUnits": 10,
                "WriteCapacityUnits": 5
            },
            "IndexSizeBytes": 0,
            "ItemCount": 0,
            "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
        }
    ]
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 6: To create a table with multiple Global Secondary Indexes at once

The following example creates a table named GameScores with two Global Secondary Indexes. The GSI schemas are passed via a file, rather than on the command line.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S
    AttributeName=GameTitle,AttributeType=S AttributeName=TopScore,AttributeType=N
    AttributeName=Date,AttributeType=S \
    --key-schema AttributeName=UserId,KeyType=HASH
    AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes file://gsi.json
```

Contents of gsi.json:

```
[

{
    "IndexName": "GameTitleIndex",
    "KeySchema": [
        {
            "AttributeName": "GameTitle",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "TopScore",
            "KeyType": "RANGE"
        }
    ],
    "Projection": {
        "ProjectionType": "ALL"
    },
    "ProvisionedThroughput": {
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
    }
},
{
    "IndexName": "GameDateIndex",
    "KeySchema": [
        {
            "AttributeName": "GameTitle",
            "KeyType": "HASH"
        }
    ],
    "Projection": {
        "ProjectionType": "ALL"
    },
    "ProvisionedThroughput": {
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
    }
}]
```

```
        "KeyType": "HASH"
    },
    {
        "AttributeName": "Date",
        "KeyType": "RANGE"
    }
],
"Projection": {
    "ProjectionType": "ALL"
},
"ProvisionedThroughput": {
    "ReadCapacityUnits": 5,
    "WriteCapacityUnits": 5
}
}
]
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Date",
                "AttributeType": "S"
            },
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ]
    }
}
```

```
        "KeyType": "HASH"
    },
    {
        "AttributeName": "GameTitle",
        "KeyType": "RANGE"
    }
],
"TableStatus": "CREATING",
"CreationDateTime": "2020-08-04T16:40:55.524000-07:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"GlobalSecondaryIndexes": [
    {
        "IndexName": "GameTitleIndex",
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "TopScore",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "IndexStatus": "CREATING",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "IndexSizeBytes": 0,
        "ItemCount": 0,
        "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/GameTitleIndex"
    }
]
```

```
        },
        {
            "IndexName": "GameDateIndex",
            "KeySchema": [
                {
                    "AttributeName": "GameTitle",
                    "KeyType": "HASH"
                },
                {
                    "AttributeName": "Date",
                    "KeyType": "RANGE"
                }
            ],
            "Projection": {
                "ProjectionType": "ALL"
            },
            "IndexStatus": "CREATING",
            "ProvisionedThroughput": {
                "NumberOfDecreasesToday": 0,
                "ReadCapacityUnits": 5,
                "WriteCapacityUnits": 5
            },
            "IndexSizeBytes": 0,
            "ItemCount": 0,
            "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/GameDateIndex"
        }
    ]
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 7: To create a table with Streams enabled

The following example creates a table called GameScores with DynamoDB Streams enabled. Both new and old images of each item will be written to the stream.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S \
    AttributeName=GameTitle,AttributeType=S \
```

```
--key-schema AttributeName=UserId,KeyType=HASH
AttributeName=GameTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--stream-specification StreamEnabled=TRUE,StreamViewType=NEW_AND_OLD_IMAGES
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T10:49:34.056000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
```

```
        "StreamViewType": "NEW_AND_OLD_IMAGES"
    },
    "LatestStreamLabel": "2020-05-27T17:49:34.056",
    "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2020-05-27T17:49:34.056"
}
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 8: To create a table with Keys-Only Stream enabled

The following example creates a table called GameScores with DynamoDB Streams enabled. Only the key attributes of modified items are written to the stream.

```
aws dynamodb create-table \
--table-name GameScores \
--attribute-definitions AttributeName=UserId,AttributeType=S
AttributeName=GameTitle,AttributeType=S \
--key-schema AttributeName=UserId,KeyType=HASH
AttributeName=GameTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--stream-specification StreamEnabled=TRUE,StreamViewType=KEYS_ONLY
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",

```

```
        "KeyType": "HASH"
    },
    {
        "AttributeName": "GameTitle",
        "KeyType": "RANGE"
    }
],
"TableStatus": "CREATING",
"CreationDateTime": "2023-05-25T18:45:34.140000+00:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"StreamSpecification": {
    "StreamEnabled": true,
    "StreamViewType": "KEYS_ONLY"
},
"LatestStreamLabel": "2023-05-25T18:45:34.140",
"LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2023-05-25T18:45:34.140",
"DeletionProtectionEnabled": false
}
}
```

For more information, see [Change data capture for DynamoDB Streams](#) in the *Amazon DynamoDB Developer Guide*.

Example 9: To create a table with the Standard Infrequent Access class

The following example creates a table called GameScores and assigns the Standard-Infrequent Access (DynamoDB Standard-IA) table class. This table class is optimized for storage being the dominant cost.

```
aws dynamodb create-table \
--table-name GameScores \
--attribute-definitions AttributeName=UserId,AttributeType=S \
AttributeType=S \
AttributeType=S \
```

```
--key-schema AttributeName=UserId,KeyType=HASH
AttributeName=GameTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--table-class STANDARD_INFREQUENT_ACCESS
```

Output:

```
{
  "TableDescription": {
    "AttributeDefinitions": [
      {
        "AttributeName": "GameTitle",
        "AttributeType": "S"
      },
      {
        "AttributeName": "UserId",
        "AttributeType": "S"
      }
    ],
    "TableName": "GameScores",
    "KeySchema": [
      {
        "AttributeName": "UserId",
        "KeyType": "HASH"
      },
      {
        "AttributeName": "GameTitle",
        "KeyType": "RANGE"
      }
    ],
    "TableStatus": "CREATING",
    "CreationDateTime": "2023-05-25T18:33:07.581000+00:00",
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 10,
      "WriteCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
    "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "TableClassSummary": {
      "TableClass": "STANDARD_INFREQUENT_ACCESS"
    }
}
```

```
    },
    "DeletionProtectionEnabled": false
}
}
```

For more information, see [Table classes](#) in the *Amazon DynamoDB Developer Guide*.

Example 10: To Create a table with Delete Protection enabled

The following example creates a table called GameScores and enables deletion protection.

```
aws dynamodb create-table \
--table-name GameScores \
--attribute-definitions AttributeName=UserId,AttributeType=S \
AttributeName=GameTitle,AttributeType=S \
--key-schema AttributeName=UserId,KeyType=HASH \
AttributeName=GameTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--deletion-protection-enabled
```

Output:

```
{
  "TableDescription": {
    "AttributeDefinitions": [
      {
        "AttributeName": "GameTitle",
        "AttributeType": "S"
      },
      {
        "AttributeName": "UserId",
        "AttributeType": "S"
      }
    ],
    "TableName": "GameScores",
    "KeySchema": [
      {
        "AttributeName": "UserId",
        "KeyType": "HASH"
      },
      {
        "AttributeName": "GameTitle",
        "KeyType": "RANGE"
      }
    ]
  }
}
```

```
        },
    ],
    "TableStatus": "CREATING",
    "CreationDateTime": "2023-05-25T23:02:17.093000+00:00",
    "ProvisionedThroughput": {
        "NumberOfDecreasesToday": 0,
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
    "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "DeletionProtectionEnabled": true
}
}
```

For more information, see [Using deletion protection](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [CreateTable](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the examples.
// It contains a DynamoDB service client that is used to act on the specified table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}
```

```
// CreateMovieTable creates a DynamoDB table with a composite primary key defined
// as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable() (*types.TableDescription, error) {
    var tableDesc *types.TableDescription
    table, err := basics.DynamoDbClient.CreateTable(context.TODO(),
        &dynamodb.CreateTableInput{
            AttributeDefinitions: []types.AttributeDefinition{
                {
                    AttributeName: aws.String("year"),
                    AttributeType: types.ScalarAttributeTypeN,
                },
                {
                    AttributeName: aws.String("title"),
                    AttributeType: types.ScalarAttributeTypeS,
                },
            },
            KeySchema: []types.KeySchemaElement{
                {
                    AttributeName: aws.String("year"),
                    KeyType:       types.KeyTypeHash,
                },
                {
                    AttributeName: aws.String("title"),
                    KeyType:       types.KeyTypeRange,
                },
            },
            TableName: aws.String(basics.TableName),
            ProvisionedThroughput: &types.ProvisionedThroughput{
                ReadCapacityUnits: aws.Int64(10),
                WriteCapacityUnits: aws.Int64(10),
            },
        },
    )
    if err != nil {
        log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
    } else {
        waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
        err = waiter.Wait(context.TODO(), &dynamodb.DescribeTableInput{
            TableName: aws.String(basics.TableName)}, 5*time.Minute)
        if err != nil {
            log.Printf("Wait for table exists failed. Here's why: %v\n", err)
        }
        tableDesc = table.TableDescription
    }
    return tableDesc, err
}
```

- For API details, see [CreateTable in AWS SDK for Go API Reference](#).

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class CreateTable {
    public static void main(String[] args) {
        final String usage = """

```

```
Usage:  
      <tableName> <key>  
  
Where:  
      tableName - The Amazon DynamoDB table to create (for example,  
      Music3).  
      key - The key for the Amazon DynamoDB table (for example,  
      Artist).  
      """;  
  
if (args.length != 2) {  
    System.out.println(usage);  
    System.exit(1);  
}  
  
String tableName = args[0];  
String key = args[1];  
System.out.println("Creating an Amazon DynamoDB table " + tableName + "  
with a simple primary key: " + key);  
Region region = Region.US_EAST_1;  
DynamoDbClient ddb = DynamoDbClient.builder()  
    .region(region)  
    .build();  
  
String result = createTable(ddb, tableName, key);  
System.out.println("New table is " + result);  
ddb.close();  
}  
  
public static String createTable(DynamoDbClient ddb, String tableName, String  
key) {  
    DynamoDbWaiter dbWaiter = ddb.waiter();  
    CreateTableRequest request = CreateTableRequest.builder()  
        .attributeDefinitions(AttributeDefinition.builder()  
            .attributeName(key)  
            .attributeType(ScalarAttributeType.S)  
            .build())  
        .keySchema(KeySchemaElement.builder()  
            .attributeName(key)  
            .keyType(KeyType.HASH)  
            .build())  
        .provisionedThroughput(ProvisionedThroughput.builder()  
            .readCapacityUnits(10L)
```

```
        .writeCapacityUnits(10L)
        .build())
    .tableName(tableName)
    .build();
```

```
String newTable;
try {
    CreateTableResponse response = ddb.createTable(request);
    DescribeTableRequest tableRequest = DescribeTableRequest.builder()
        .tableName(tableName)
        .build();

    // Wait until the Amazon DynamoDB table is created.
    WaiterResponse<DescribeTableResponse> waiterResponse =
    dbWaiter.waitUntilTableExists(tableRequest);
    waiterResponse.matched().response().ifPresent(System.out::println);
    newTable = response.tableDescription().tableName();
    return newTable;

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
return "";
}
```

```
}
```

- For API details, see [CreateTable in AWS SDK for Java 2.x API Reference](#).

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    ProvisionedThroughput: {
      ReadCapacityUnits: 1,
      WriteCapacityUnits: 1,
    },
  });
}

const response = await client.send(command);
console.log(response);
return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTable](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "CUSTOMER_LIST",
```

```
    StreamSpecification: {
        StreamEnabled: false,
    },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Table Created", data);
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTable](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun createNewTable(tableNameVal: String, key: String): String? {
    val attDef = AttributeDefinition {
        attributeName = key
        attributeType = ScalarAttributeType.S
    }

    val keySchemaVal = KeySchemaElement {
        attributeName = key
        keyType = KeyType.Hash
    }

    val provisionedVal = ProvisionedThroughput {
        readCapacityUnits = 10
    }
}
```

```
        writeCapacityUnits = 10
    }

    val request = CreateTableRequest {
        attributeDefinitions = listOf(attDef)
        keySchema = listOf(keySchemaVal)
        provisionedThroughput = provisionedVal
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->

        var tableArn: String
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists { // suspend call
            tableName = tableNameVal
        }
        tableArn = response.tableDescription!!.tableArn.toString()
        println("Table $tableArn is ready")
        return tableArn
    }
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a table.

```
$tableName = "ddb_demo_table_$uuid";
$service->createTable(
    $tableName,
    [
```

```
        new DynamoDBAttribute('year', 'N', 'HASH'),
        new DynamoDBAttribute('title', 'S', 'RANGE')
    ]
);

public function createTable(string $tableName, array $attributes)
{
    $keySchema = [];
    $attributeDefinitions = [];
    foreach ($attributes as $attribute) {
        if (is_a($attribute, DynamoDBAttribute::class)) {
            $keySchema[] = ['AttributeName' => $attribute->AttributeName,
'KeyType' => $attribute->KeyType];
            $attributeDefinitions[] =
                ['AttributeName' => $attribute->AttributeName,
'AttributeType' => $attribute->AttributeType];
        }
    }

    $this->dynamoDbClient->createTable([
        'TableName' => $tableName,
        'KeySchema' => $keySchema,
        'AttributeDefinitions' => $attributeDefinitions,
        'ProvisionedThroughput' => ['ReadCapacityUnits' => 10,
'WriteCapacityUnits' => 10],
    ]);
}
```

- For API details, see [CreateTable in AWS SDK for PHP API Reference](#).

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a table for storing movie data.

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
  
    def create_table(self, table_name):  
        """  
        Creates an Amazon DynamoDB table that can be used to store movie data.  
        The table uses the release year of the movie as the partition key and the  
        title as the sort key.  
  
        :param table_name: The name of the table to create.  
        :return: The newly created table.  
        """  
        try:  
            self.table = self.dyn_resource.create_table(  
                TableName=table_name,  
                KeySchema=[  
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition  
key  
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key  
                ],  
                AttributeDefinitions=[  
                    {"AttributeName": "year", "AttributeType": "N"},  
                    {"AttributeName": "title", "AttributeType": "S"},  
                ],  
                ProvisionedThroughput={  
                    "ReadCapacityUnits": 10,  
                    "WriteCapacityUnits": 10,  
                },  
            )  
            self.table.wait_until_exists()  
        except ClientError as err:  
            logger.error(  
                "Couldn't create table %s. Here's why: %s: %s",
```

```
        table_name,
        err.response["Error"]["Code"],
        err.response["Error"]["Message"],
    )
    raise
else:
    return self.table
```

- For API details, see [CreateTable](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
    attr_reader :dynamo_resource
    attr_reader :table_name
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table_name = table_name
        @table = nil
        @logger = Logger.new($stdout)
        @logger.level = Logger::DEBUG
    end

    # Creates an Amazon DynamoDB table that can be used to store movie data.
    # The table uses the release year of the movie as the partition key and the
    # title as the sort key.
    #
    # @param table_name [String] The name of the table to create.
```

```
# @return [Aws::DynamoDB::Table] The newly created table.
def create_table(table_name)
  @table = @dynamo_resource.create_table(
    table_name: table_name,
    key_schema: [
      {attribute_name: "year", key_type: "HASH"}, # Partition key
      {attribute_name: "title", key_type: "RANGE"} # Sort key
    ],
    attribute_definitions: [
      {attribute_name: "year", attribute_type: "N"},
      {attribute_name: "title", attribute_type: "S"}
    ],
    provisioned_throughput: {read_capacity_units: 10, write_capacity_units: 10})
  @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
  @table
rescue Aws::DynamoDB::Errors::ServiceError => e
  @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
  raise
end
```

- For API details, see [CreateTable](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn create_table(
  client: &Client,
  table: &str,
  key: &str,
) -> Result< CreateTableOutput, Error> {
  let a_name: String = key.into();
  let table_name: String = table.into();
```

```
let ad = AttributeDefinition::builder()
    .attribute_name(&a_name)
    .attribute_type(ScalarAttributeType::S)
    .build()
    .map_err(Error::BuildError)?;

let ks = KeySchemaElement::builder()
    .attribute_name(&a_name)
    .key_type(KeyType::Hash)
    .build()
    .map_err(Error::BuildError)?;

let pt = ProvisionedThroughput::builder()
    .read_capacity_units(10)
    .write_capacity_units(5)
    .build()
    .map_err(Error::BuildError)?;

let create_table_response = client
    .create_table()
    .table_name(table_name)
    .key_schema(ks)
    .attribute_definitions(ad)
    .provisioned_throughput(pt)
    .send()
    .await;

match create_table_response {
    Ok(out) => {
        println!("Added table {} with key {}", table, key);
        Ok(out)
    }
    Err(e) => {
        eprintln!("Got an error creating table:");
        eprintln!("{}: {}", e);
        Err(Error::unhandled(e))
    }
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
    ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                         iv_keytype = 'HASH' ) )
    ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                         iv_keytype = 'RANGE' ) ) ).

DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
    ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                         iv_attributetype = 'N' ) )
    ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                         iv_attributetype = 'S' ) ) ).

" Adjust read/write capacities as desired.
DATA(lo_dynprovthroughput) = NEW /aws1/cl_dynprovthroughput(
    iv_readcapacityunits = 5
    iv_writecapacityunits = 5 ).
oo_result = lo_dyn->createtable(
    it_keyschema = lt_keyschema
    iv_tablename = iv_table_name
    it_attributedefinitions = lt_attributedefinitions
    io_provisionedthroughput = lo_dynprovthroughput ).

" Table creation can take some time. Wait till table exists before
returning.
lo_dyn->get_waiter( )->tableexists(
    iv_max_wait_time = 200
    iv_tablename      = iv_table_name ).
MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
" This exception can happen if the table already exists.
CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
    DATA(lv_error) = |"{
        lo_resourceinuseex->av_err_code
    }" -
    { lo_resourceinuseex->av_err_msg }|.
```

```
MESSAGE lv_error TYPE 'E'.
ENDTRY.
```

- For API details, see [CreateTable](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
///
/// Create a movie table in the Amazon DynamoDB data store.
///
private func createTable() async throws {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = CreateTableInput(
        attributeDefinitions: [
            DynamoDBClientTypes.AttributeDefinition(attributeName: "year",
attributeType: .n),
            DynamoDBClientTypes.AttributeDefinition(attributeName: "title",
attributeType: .s),
        ],
        keySchema: [
            DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
```

```
DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
],
provisionedThroughput: DynamoDBClientTypes.ProvisionedThroughput(
    readCapacityUnits: 10,
    writeCapacityUnits: 10
),
tableName: self.tableName
)
let output = try await client.createTable(input: input)
if output.tableDescription == nil {
    throw MoviesError.TableNotFound
}
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Write an item to a DynamoDB table

You can write items to DynamoDB tables using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on items, see [Core components of Amazon DynamoDB](#).

Write an item to a DynamoDB table using an AWS SDK

The following code examples show how to write an item to a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
```

```
    /// Adds a new item to the table.  
    /// </summary>  
    /// <param name="client">An initialized Amazon DynamoDB client object.</param>  
    /// <param name="newMovie">A Movie object containing information for  
    /// the movie to add to the table.</param>  
    /// <param name="tableName">The name of the table where the item will be  
    added.</param>  
    /// <returns>A Boolean value that indicates the results of adding the  
    item.</returns>  
    public static async Task<bool> PutItemAsync(AmazonDynamoDBClient client,  
        Movie newMovie, string tableName)  
    {  
        var item = new Dictionary<string, AttributeValue>  
        {  
            ["title"] = new AttributeValue { S = newMovie.Title },  
            ["year"] = new AttributeValue { N = newMovie.Year.ToString() },  
        };  
  
        var request = new PutItemRequest  
        {  
            TableName = tableName,  
            Item = item,  
        };  
  
        var response = await client.PutItemAsync(request);  
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;  
    }  
}
```

- For API details, see [PutItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -i item -- Path to json file containing the item values.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_put_item() {
    local table_name item response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -i item -- Path to json file containing the item values."
    echo ""
}

while getopt "n:i:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        i) item="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
```

```
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    item: $item"
iecho ""
iecho ""

response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
fi

return 0
}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
```

```
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#       $1 - The error code returned by the AWS CLI.
#
# Returns:
#       0: - Success.
#
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
```

```
    errecho "  The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho "  The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho "  255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [PutItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Put an item in an Amazon DynamoDB table.
/*
\sa putItem()
\param tableName: The table name.
\param artistKey: The artist key. This is the partition key for the table.
\param artistValue: The artist value.
\param albumTitleKey: The album title key.
\param albumTitleValue: The album title value.
\param awardsKey: The awards key.
\param awardsValue: The awards value.
\param songTitleKey: The song title key.
\param songTitleValue: The song title value.
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::putItem(const Aws::String &tableName,
                                const Aws::String &artistKey,
                                const Aws::String &artistValue,
                                const Aws::String &albumTitleKey,
```

```
        const Aws::String &albumTitleValue,
        const Aws::String &awardsKey,
        const Aws::String &awardsValue,
        const Aws::String &songTitleKey,
        const Aws::String &songTitleValue,
        const Aws::Client::ClientConfiguration

&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::PutItemRequest putItemRequest;
    putItemRequest.SetTableName(tableName);

    putItemRequest.AddItem(artistKey,
Aws::DynamoDB::Model::AttributeValue().SetS(
        artistValue)); // This is the hash key.
    putItemRequest.AddItem(albumTitleKey,
Aws::DynamoDB::Model::AttributeValue().SetS(
        albumTitleValue));
    putItemRequest.AddItem(awardsKey,

Aws::DynamoDB::Model::AttributeValue().SetS(awardsValue));
    putItemRequest.AddItem(songTitleKey,

Aws::DynamoDB::Model::AttributeValue().SetS(songTitleValue));

    const Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
        putItemRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully added Item!" << std::endl;
    }
    else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see [PutItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To add an item to a table

The following `put-item` example adds a new item to the *MusicCollection* table.

```
aws dynamodb put-item \  
  --table-name MusicCollection \  
  --item file://item.json \  
  --return-consumed-capacity TOTAL \  
  --return-item-collection-metrics SIZE
```

Contents of `item.json`:

```
{  
    "Artist": {"S": "No One You Know"},  
    "SongTitle": {"S": "Call Me Today"},  
    "AlbumTitle": {"S": "Greatest Hits"}  
}
```

Output:

```
{  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 1.0  
    },  
    "ItemCollectionMetrics": {  
        "ItemCollectionKey": {  
            "Artist": {  
                "S": "No One You Know"  
            }  
        },  
        "SizeEstimateRangeGB": [  
            0.0,  
            1.0  
        ]  
    }  
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To conditionally overwrite an item in a table

The following put-item example overwrites an existing item in the MusicCollection table only if that existing item has an AlbumTitle attribute with a value of Greatest Hits. The command returns the previous value of the item.

```
aws dynamodb put-item \  
  --table-name MusicCollection \  
  --item file://item.json \  
  --condition-expression "#A = :A" \  
  --expression-attribute-names file://names.json \  
  --expression-attribute-values file://values.json \  
  --return-values ALL_OLD
```

Contents of item.json:

```
{  
    "Artist": {"S": "No One You Know"},  
    "SongTitle": {"S": "Call Me Today"},  
    "AlbumTitle": {"S": "Somewhat Famous"}  
}
```

Contents of names.json:

```
{  
    "#A": "AlbumTitle"  
}
```

Contents of values.json:

```
{  
    ":A": {"S": "Greatest Hits"}  
}
```

Output:

```
{  
    "Attributes": {  
        "AlbumTitle": {  
            "S": "Greatest Hits"  
        },  
    },  
}
```

```
        "Artist": {  
            "S": "No One You Know"  
        },  
        "SongTitle": {  
            "S": "Call Me Today"  
        }  
    }  
}
```

If the key already exists, you should see the following output:

A client error (ConditionalCheckFailedException) occurred when calling the PutItem operation: The conditional request failed.

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [PutItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the  
// examples.  
// It contains a DynamoDB service client that is used to act on the specified  
// table.  
type TableBasics struct {  
    DynamoDbClient *dynamodb.Client  
    TableName      string  
}  
  
  
// AddMovie adds a movie to the DynamoDB table.  
func (basics TableBasics) AddMovie(movie Movie) error {
```

```
item, err := attributevalue.MarshalMap(movie)
if err != nil {
    panic(err)
}
_, err = basics.DynamoDbClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
    TableName: aws.String(basics.TableName), Item: item,
})
if err != nil {
    log.Printf("Couldn't add item to table. Here's why: %v\n", err)
}
return err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",

```

```
    movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [PutItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Puts an item into a table using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
```

```
public static void main(String[] args) {
    final String usage = """

        Usage:
            <tableName> <key> <keyVal> <albumtitle> <albumtitleval>
<awards> <awardsval> <Songtitle> <songtitleval>

        Where:
            tableName - The Amazon DynamoDB table in which an item is
placed (for example, Music3).
            key - The key used in the Amazon DynamoDB table (for example,
Artist).
            keyval - The key value that represents the item to get (for
example, Famous Band).
            albumTitle - The Album title (for example, AlbumTitle).
            AlbumTitleValue - The name of the album (for example, Songs
About Life .
            Awards - The awards column (for example, Awards).
            AwardVal - The value of the awards (for example, 10).
            SongTitle - The song title (for example, SongTitle).
            SongTitleVal - The value of the song title (for example,
Happy Day).

        **Warning** This program will place an item that you specify
into a table!
        """;

    if (args.length != 9) {
        System.out.println(usage);
        System.exit(1);
    }

    String tableName = args[0];
    String key = args[1];
    String keyVal = args[2];
    String albumTitle = args[3];
    String albumTitleValue = args[4];
    String awards = args[5];
    String awardVal = args[6];
    String songTitle = args[7];
    String songTitleVal = args[8];

    Region region = Region.US_EAST_1;
    DynamoDbClient ddb = DynamoDbClient.builder()
        .region(region)
```

```
        .build();

    putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
            songTitleVal);
    System.out.println("Done!");
    ddb.close();
}

public static void putItemInTable(DynamoDbClient ddb,
        String tableName,
        String key,
        String keyVal,
        String albumTitle,
        String albumTitleValue,
        String awards,
        String awardVal,
        String songTitle,
        String songTitleVal) {

    HashMap<String,AttributeValue> itemValues = new HashMap<>();
    itemValues.put(key, AttributeValue.builder().s(keyVal).build());
    itemValues.put(songTitle,
AttributeValue.builder().s(songTitleVal).build());
    itemValues.put(albumTitle,
AttributeValue.builder().s(albumTitleValue).build());
    itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

    PutItemRequest request = PutItemRequest.builder()
        .tableName(tableName)
        .item(itemValues)
        .build();

    try {
        PutItemResponse response = ddb.putItem(request);
        System.out.println(tableName + " was successfully updated. The
request id is "
                + response.responseMetadata().requestId());

    } catch (ResourceNotFoundException e) {
        System.err.format("Error: The Amazon DynamoDB table \'%s\' can't be
found.\n", tableName);
        System.err.println("Be sure that it exists and that you've typed its
name correctly!");
    }
}
```

```
        System.exit(1);
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
```

- For API details, see [PutItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [PutCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
    const command = new PutCommand({
        TableName: "HappyAnimals",
        Item: {
            CommonName: "Shiba Inu",
        },
    });
}

const response = await docClient.send(command);
console.log(response);
return response;
```

```
};
```

- For API details, see [PutItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Put an item in a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

Put an item in a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutItem](#) in [AWS SDK for JavaScript API Reference](#).

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun putItemInTable(
    tableNameVal: String,
```

```
    key: String,  
    keyVal: String,  
    albumTitle: String,  
    albumTitleValue: String,  
    awards: String,  
    awardVal: String,  
    songTitle: String,  
    songTitleVal: String  
) {  
    val itemValues = mutableMapOf<String, AttributeValue>()  
  
    // Add all content to the table.  
    itemValues[key] = AttributeValue.S(keyVal)  
    itemValues[songTitle] = AttributeValue.S(songTitleVal)  
    itemValues[albumTitle] = AttributeValue.S(albumTitleValue)  
    itemValues[awards] = AttributeValue.S(awardVal)  
  
    val request = PutItemRequest {  
        tableName = tableNameVal  
        item = itemValues  
    }  
  
    DynamoDbClient { region = "us-east-1" }.use { ddb ->  
        ddb.putItem(request)  
        println(" A new item was placed into $tableNameVal.")  
    }  
}
```

- For API details, see [PutItem](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
echo "What's the name of the last movie you watched?\n";
```

```
while (empty($movieName)) {
    $movieName = testable_readline("Movie name: ");
}
echo "And what year was it released?\n";
$movieYear = "year";
while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
    $movieYear = testable_readline("Year released: ");
}

$service->putItem([
    'Item' => [
        'year' => [
            'N' => "$movieYear",
        ],
        'title' => [
            'S' => $movieName,
        ],
    ],
    'TableName' => $tableName,
]);
}

public function putItem(array $array)
{
    $this->dynamoDbClient->putItem($array);
}
```

- For API details, see [PutItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""
```

```
def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None

def add_movie(self, title, year, plot, rating):
    """
    Adds a movie to the table.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :param plot: The plot summary of the movie.
    :param rating: The quality rating of the movie.
    """
    try:
        self.table.put_item(
            Item={
                "year": year,
                "title": title,
                "info": {"plot": plot, "rating": Decimal(str(rating))},
            }
        )
    except ClientError as err:
        logger.error(
            "Couldn't add movie %s to table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [PutItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Adds a movie to the table.
    #
    # @param movie [Hash] The title, year, plot, and rating of the movie.
    def add_item(movie)
        @table.put_item(
            item: {
                "year" => movie[:year],
                "title" => movie[:title],
                "info" => {"plot" => movie[:plot], "rating" => movie[:rating]}))
    rescue Aws::DynamoDB::Errors::ServiceError => e
        puts("Couldn't add movie #{title} to table #{@table.name}. Here's why:")
        puts("\t#{e.code}: #{e.message}")
        raise
    end
end
```

- For API details, see [PutItem](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn add_item(client: &Client, item: Item, table: &String) ->
Result<ItemOut, Error> {
    let user_av = AttributeValue::S(item.username);
    let type_av = AttributeValue::S(item.p_type);
    let age_av = AttributeValue::S(item.age);
    let first_av = AttributeValue::S(item.first);
    let last_av = AttributeValue::S(item.last);

    let request = client
        .put_item()
        .table_name(table)
        .item("username", user_av)
        .item("account_type", type_av)
        .item("age", age_av)
        .item("first_name", first_av)
        .item("last_name", last_av);

    println!("Executing request [{request:?}] to add item...");

    let resp = request.send().await?;

    let attributes = resp.attributes().unwrap();

    let username = attributes.get("username").cloned();
    let first_name = attributes.get("first_name").cloned();
    let last_name = attributes.get("last_name").cloned();
    let age = attributes.get("age").cloned();
    let p_type = attributes.get("p_type").cloned();

    println!(
        "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
        username, first_name, last_name, age, p_type
    )
}
```

```
);

Ok(ItemOut {
    p_type,
    age,
    username,
    first_name,
    last_name,
})
}
```

- For API details, see [PutItem](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.
  DATA(lo_resp) = lo_dyn->putitem(
    iv_tablename = iv_table_name
    it_item      = it_item ).
  MESSAGE '1 row inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
  CATCH /aws1/cx_dyncondalcheckfaile00.
  MESSAGE 'A condition specified in the operation could not be evaluated.' TYPE 'E'.
  CATCH /aws1/cx_dynresourcenotfoundex.
  MESSAGE 'The table or index does not exist' TYPE 'E'.
  CATCH /aws1/cx_dynctransactconflictex.
  MESSAGE 'Another transaction is using the item' TYPE 'E'.
ENDTRY.
```

- For API details, see [PutItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
/// table.
///
/// - Parameter movie: The `Movie` to add to the table.
///
func add(movie: Movie) async throws {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    // Get a DynamoDB item containing the movie data.
    let item = try await movie.getAsItem()

    // Send the `PutItem` request to Amazon DynamoDB.

    let input = PutItemInput(
        item: item,
        tableName: self.tableName
    )
    _ = try await client.putItem(input: input)
}

///
/// Return an array mapping attribute names to Amazon DynamoDB attribute
/// values, representing the contents of the `Movie` record as a DynamoDB
```

```
/// item.  
///  
/// - Returns: The movie item as an array of type  
///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.  
///  
func getAsItem() async throws ->  
[Swift.String:DynamoDBClientTypes.AttributeValue] {  
    // Build the item record, starting with the year and title, which are  
    // always present.  
  
    var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [  
        "year": .n(String(self.year)),  
        "title": .s(self.title)  
    ]  
  
    // Add the `info` field with the rating and/or plot if they're  
    // available.  
  
    var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]  
    if (self.info.rating != nil || self.info.plot != nil) {  
        if self.info.rating != nil {  
            details["rating"] = .n(String(self.info.rating!))  
        }  
        if self.info.plot != nil {  
            details["plot"] = .s(self.info.plot!)  
        }  
    }  
    item["info"] = .m(details)  
  
    return item  
}
```

- For API details, see [PutItem](#) in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Read an item from a DynamoDB table

You can read items from DynamoDB tables using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on items, see [Core components of Amazon DynamoDB](#).

Read an item from a DynamoDB table using an AWS SDK

The following code examples show how to read an item from a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Gets information about an existing movie from the table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="newMovie">A Movie object containing information about the movie to retrieve.</param>
/// <param name="tableName">The name of the table containing the movie.</param>
/// <returns>A Dictionary object containing information about the item retrieved.</returns>
public static async Task<Dictionary<string, AttributeValue>>
GetItemAsync(AmazonDynamoDBClient client, Movie newMovie, string tableName)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };

    var request = new GetItemRequest
    {
        Key = key,
        TableName = tableName,
    };
}
```

```
        var response = await client.GetItemAsync(request);
        return response.Item;
    }
```

- For API details, see [GetItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to get.
#     [-q query] -- Optional JMESPath query expression.
#
# Returns:
#     The item as text output.
# And:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_get_item() {
    local table_name keys query response
    local option OPTARG # Required to use getopt command in a function.

    # #####
    # Function usage explanation
    #####
```

```
function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -k keys -- Path to json file containing the keys that identify the
item to get."
    echo " [-q query] -- Optional JMESPath query expression."
    echo ""
}

query=""
while getopts "n:k:q:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        k) keys="${OPTARG}" ;;
        q) query="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) 
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
fi

if [[ -n "$query" ]]; then
    response=$(aws dynamodb get-item \
        --table-name "$table_name" \
        --key file://"$keys" \
```

```
--output text \
--query "$query")
else
    response=$(
        aws dynamodb get-item \
        --table-name "$table_name" \
        --key file://"${keys}" \
        --output text
    )
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
fi

if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\\t/s/\\t//1" # Remove initial tab that the JMSEPath
query inserts on some strings.
else
    echo "$response"
fi

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
```

```
#  
# This function is used to log the error messages from the AWS CLI.  
#  
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-  
help-return-codes.  
#  
# The function expects the following argument:  
#           $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#           0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}
```

- For API details, see [GetItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Get an item from an Amazon DynamoDB table.  
/*!  
 \sa getItem()  
 \param tableName: The table name.  
 \param partitionKey: The partition key.  
 \param partitionValue: The value for the partition key.  
 \param clientConfiguration: AWS client configuration.  
 \return bool: Function succeeded.  
 */  
  
bool AwsDoc::DynamoDB::getItem(const Aws::String &tableName,  
                                 const Aws::String &partitionKey,  
                                 const Aws::String &partitionValue,  
                                 const Aws::Client::ClientConfiguration  
&clientConfiguration) {  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
    Aws::DynamoDB::Model::GetItemRequest request;  
  
    // Set up the request.  
    request.SetTableName(tableName);  
    request.AddKey(partitionKey,  
                  Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));  
  
    // Retrieve the item's fields and values.  
    const Aws::DynamoDB::Model::GetItemOutcome &outcome =  
        dynamoClient.GetItem(request);  
    if (outcome.IsSuccess()) {  
        // Reference the retrieved fields/values.  
        const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> &item =  
            outcome.GetResult().GetItem();  
        if (!item.empty()) {  
            // Output each retrieved field and its value.  
        }  
    }  
}
```

```
        for (const auto &i: item)
            std::cout << "Values: " << i.first << ":" << i.second.GetS()
                           << std::endl;
    }
else {
    std::cout << "No item found with the key " << partitionKey <<
std::endl;
}
else {
    std::cerr << "Failed to get item: " << outcome.GetError().GetMessage();
}

return outcome.IsSuccess();
}
```

- For API details, see [GetItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To read an item in a table

The following get-item example retrieves an item from the MusicCollection table. The table has a hash-and-range primary key (Artist and SongTitle), so you must specify both of these attributes. The command also requests information about the read capacity consumed by the operation.

```
aws dynamodb get-item \
--table-name MusicCollection \
--key file://key.json \
--return-consumed-capacity TOTAL
```

Contents of key.json:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Output:

```
{  
    "Item": {  
        "AlbumTitle": {  
            "S": "Songs About Life"  
        },  
        "SongTitle": {  
            "S": "Happy Day"  
        },  
        "Artist": {  
            "S": "Acme Band"  
        }  
    },  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 0.5  
    }  
}
```

For more information, see [Reading an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To read an item using a consistent read

The following example retrieves an item from the MusicCollection table using strongly consistent reads.

```
aws dynamodb get-item \  
  --table-name MusicCollection \  
  --key file://key.json \  
  --consistent-read \  
  --return-consumed-capacity TOTAL
```

Contents of key.json:

```
{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"}  
}
```

Output:

```
{  
    "Item": {  
        "AlbumTitle": {  
            "S": "Songs About Life"  
        },  
        "SongTitle": {  
            "S": "Happy Day"  
        },  
        "Artist": {  
            "S": "Acme Band"  
        }  
    },  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 1.0  
    }  
}
```

For more information, see [Reading an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To retrieve specific attributes of an item

The following example uses a projection expression to retrieve only three attributes of the desired item.

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key '{"Id": {"N": "102"}}' \  
  --projection-expression "#T, #C, #P" \  
  --expression-attribute-names file://names.json
```

Contents of names.json:

```
{  
    "#T": "Title",  
    "#C": "ProductCategory",  
    "#P": "Price"  
}
```

Output:

```
{
```

```
"Item": {  
    "Price": {  
        "N": "20"  
    },  
    "Title": {  
        "S": "Book 102 Title"  
    },  
    "ProductCategory": {  
        "S": "Book"  
    }  
}
```

For more information, see [Reading an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [GetItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the  
// examples.  
// It contains a DynamoDB service client that is used to act on the specified  
// table.  
type TableBasics struct {  
    DynamoDbClient *dynamodb.Client  
    TableName      string  
}  
  
  
// GetMovie gets movie data from the DynamoDB table by using the primary  
// composite key  
// made of title and year.
```

```
func (basics TableBasics) GetMovie(title string, year int) (Movie, error) {
    movie := Movie{Title: title, Year: year}
    response, err := basics.DynamoDbClient.GetItem(context.TODO(),
        &dynamodb.GetItemInput{
            Key: movie.GetKey(), TableName: aws.String(basics.TableName),
        })
    if err != nil {
        log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
    } else {
        err = attributevalue.UnmarshalMap(response.Item, &movie)
        if err != nil {
            log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
        }
    }
    return movie, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}
```

```
// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [GetItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Gets an item from a table by using the `DynamoDbClient`.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
```

```
* its better practice to use the
* Enhanced Client, see the EnhancedGetItem example.
*/
public class GetItem {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key> <keyVal>

            Where:
            tableName - The Amazon DynamoDB table from which an item is
            retrieved (for example, Music3).\s
            key - The key used in the Amazon DynamoDB table (for example,
            Artist).\s
            keyval - The key value that represents the item to get (for
            example, Famous Band).
            """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal,
        tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName,
    String key, String keyVal) {
        HashMap<String,AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
            .s(keyVal)
            .build());
    }
}
```

```
GetItemRequest request = GetItemRequest.builder()
    .key(keyToGet)
    .tableName(tableName)
    .build();

try {
    // If there is no matching item, GetItem does not return any data.
    Map<String, AttributeValue> returnedItem =
ddb.getItem(request).item();
    if (returnedItem.isEmpty())
        System.out.format("No item found with the key %s!\n", key);
    else {
        Set<String> keys = returnedItem.keySet();
        System.out.println("Amazon DynamoDB table attributes: \n");
        for (String key1 : keys) {
            System.out.format("%s: %s\n", key1,
returnedItem.get(key1).toString());
        }
    }
}

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

- For API details, see [GetItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [GetCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [GetItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get an item from a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
var params = {
    TableName: "TABLE",
    Key: {
        KEY_NAME: { N: "001" },
    },
    ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success", data.Item);
    }
});
```

Get an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
    TableName: "EPISODES_TABLE",
    Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success", data.Item);
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).

- For API details, see [GetItem](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun getSpecificItem(tableNameVal: String, keyName: String, keyVal: String) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request = GetItemRequest {
        key = keyToGet
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

- For API details, see [GetItem](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$movie = $service->getItemByKey($tableName, $key);
echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";

public function getItemByKey(string $tableName, array $key)
{
    return $this->dynamoDbClient->getItem([
        'Key' => $key['Item'],
        'TableName' => $tableName,
    ]);
}
```

- For API details, see [GetItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
```

```
"""
:param dyn_resource: A Boto3 DynamoDB resource.
"""

self.dyn_resource = dyn_resource
# The table variable is set during the scenario in the call to
# 'exists' if the table exists. Otherwise, it is set by 'create_table'.
self.table = None


def get_movie(self, title, year):
    """
    Gets movie data from the table for a specific movie.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :return: The data about the requested movie.
    """

    try:
        response = self.table.get_item(Key={"year": year, "title": title})
    except ClientError as err:
        logger.error(
            "Couldn't get movie %s from table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Item"]
```

- For API details, see [GetItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Gets movie data from the table for a specific movie.
    #
    # @param title [String] The title of the movie.
    # @param year [Integer] The release year of the movie.
    # @return [Hash] The data about the requested movie.
    def get_item(title, year)
        @table.get_item(key: {"year" => year, "title" => title})
    rescue Aws::DynamoDB::Errors::ServiceError => e
        puts("Couldn't get movie #{title} (#{year}) from table #{@table.name}:\n")
        puts("\t#{e.code}: #{e.message}")
        raise
    end

```

- For API details, see [GetItem](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
oo_item = lo_dyn->getitem(
    iv_tablename           = iv_table_name
    it_key                 = it_key ).
DATA(lt_attr) = oo_item->get_item( ).
DATA(lo_title) = lt_attr[ key = 'title' ]-value.
DATA(lo_year) = lt_attr[ key = 'year' ]-value.
DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
MESSAGE 'Movie name is: ' && lo_title->get_s( )
&& 'Movie year is: ' && lo_year->get_n( )
&& 'Moving rating is: ' && lo_rating->get_n( ) TYPE 'I'.
CATCH /aws1/cx_dynresourcenotfoundex.
MESSAGE 'The table or index does not exist' TYPE 'E'.
ENDTRY.
```

- For API details, see [GetItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Return a `Movie` record describing the specified movie from the Amazon
/// DynamoDB table.
///
/// - Parameters:
///   - title: The movie's title (`String`).
///   - year: The movie's release year (`Int`).
///
/// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
///
/// - Returns: A `Movie` record with the movie's details.
func get(title: String, year: Int) async throws -> Movie {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = GetItemInput(
        key: [
            "year": .n(String(year)),
            "title": .s(title)
        ],
        tableName: self.tableName
    )
    let output = try await client.getItem(input: input)
    guard let item = output.item else {
        throw MoviesError.ItemNotFound
    }

    let movie = try Movie(withItem: item)
    return movie
}
```

- For API details, see [GetItem](#) in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Update an item in a DynamoDB table

You can update items from DynamoDB tables using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on items, see [Core components of Amazon DynamoDB](#).

Update an item in a DynamoDB table using an AWS SDK

The following code examples show how to update an item in a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Updates an existing item in the movies table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="newMovie">A Movie object containing information for the movie to update.</param>
/// <param name="newInfo">A MovieInfo object that contains the information that will be changed.</param>
/// <param name="tableName">The name of the table that contains the movie.</param>
/// <returns>A Boolean value that indicates the success of the operation.</returns>
public static async Task<bool> UpdateItemAsync(
    AmazonDynamoDBClient client,
    Movie newMovie,
    MovieInfo newInfo,
    string tableName)
{
    var key = new Dictionary<string,AttributeValue>
    {
```

```
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };
    var updates = new Dictionary<string, AttributeValueUpdate>
    {
        ["info.plot"] = new AttributeValueUpdate
        {
            Action = AttributeAction.PUT,
            Value = new AttributeValue { S = newInfo.Plot },
        },

        ["info.rating"] = new AttributeValueUpdate
        {
            Action = AttributeAction.PUT,
            Value = new AttributeValue { N = newInfo.Rank.ToString() },
        },
    };

    var request = new UpdateItemRequest
    {
        AttributeUpdates = updates,
        Key = key,
        TableName = tableName,
    };

    var response = await client.UpdateItemAsync(request);

    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to update.
#     -e update_expression -- An expression that defines one or more
#                           attributes to be updated.
#     -v values -- Path to json file containing the update values.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_update_item() {
    local table_name keys update_expression values response
    local option OPTARG # Required to use getopts command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_update_item"
    echo "Update an item in a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -k keys -- Path to json file containing the keys that identify the
item to update."
    echo " -e update_expression -- An expression that defines one or more
attributes to be updated."
    echo " -v values -- Path to json file containing the update values."
    echo ""
}

while getopts "n:k:e:v:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        k) keys="${OPTARG}" ;;
        e) update_expression="${OPTARG}" ;;
        v) values="${OPTARG}" ;;
```

```
h)
    usage
    return 0
    ;;
\?) 
    echo "Invalid parameter"
    usage
    return 1
    ;;
esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
fi
if [[ -z "$update_expression" ]]; then
    errecho "ERROR: You must provide an update expression with the -e parameter."
    usage
    return 1
fi

if [[ -z "$values" ]]; then
    errecho "ERROR: You must provide a values json file path the -v parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    keys: $keys"
iecho "    update_expression: $update_expression"
iecho "    values: $values"

response=$(aws dynamodb update-item \
--table-name "$table_name" \
```

```
--key file://"${keys}" \
--update-expression "$update_expression" \
--expression-attribute-values file://"${values}")\n\nlocal error_code=$?\n\nif [[ $error_code -ne 0 ]]; then\n    aws_cli_error_log $error_code\n    errecho "ERROR: AWS reports update-item operation failed.$response"\n    return 1\nfi\n\nreturn 0\n}\n\n
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
```

```
# This function is used to log the error messages from the AWS CLI.  
#  
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-  
# help-return-codes.  
#  
# The function expects the following argument:  
#       $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#       0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}
```

- For API details, see [UpdateItem in AWS CLI Command Reference](#).

C++

SDK for C++**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Update an Amazon DynamoDB table item.
/*!
 \sa updateItem()
 \param tableName: The table name.
 \param partitionKey: The partition key.
 \param partitionValue: The value for the partition key.
 \param attributeKey: The key for the attribute to be updated.
 \param attributeValue: The value for the attribute to be updated.
 \param clientConfiguration: AWS client configuration.
 \return bool: Function succeeded.
 */

/*
 * The example code only sets/updates an attribute value. It processes
 * the attribute value as a string, even if the value could be interpreted
 * as a number. Also, the example code does not remove an existing attribute
 * from the key value.
 */

bool AwsDoc::DynamoDB::updateItem(const Aws::String &tableName,
                                    const Aws::String &partitionKey,
                                    const Aws::String &partitionValue,
                                    const Aws::String &attributeKey,
                                    const Aws::String &attributeValue,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // *** Define UpdateItem request arguments.
    // Define TableName argument.
    Aws::DynamoDB::Model::UpdateItemRequest request;
    request.SetTableName(tableName);
```

```
// Define KeyName argument.
Aws::DynamoDB::Model::AttributeValue attribValue;
attribValue.SetS(partitionValue);
request.AddKey(partitionKey, attribValue);

// Construct the SET update expression argument.
Aws::String update_expression("SET #a = :valueA");
request.SetUpdateExpression(update_expression);

// Construct attribute name argument.
Aws::Map<Aws::String, Aws::String> expressionAttributeNames;
expressionAttributeNames["#a"] = attributeKey;
request.SetExpressionAttributeNames(expressionAttributeNames);

// Construct attribute value argument.
Aws::DynamoDB::Model::AttributeValue attributeUpdatedValue;
attributeUpdatedValue.SetS(attributeValue);
Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
expressionAttributeValues;
expressionAttributeValues[":valueA"] = attributeUpdatedValue;
request.SetExpressionAttributeValues(expressionAttributeValues);

// Update the item.
const Aws::DynamoDB::Model::UpdateItemOutcome &outcome =
dynamoClient.UpdateItem(
    request);
if (outcome.IsSuccess()) {
    std::cout << "Item was updated" << std::endl;
}
else {
    std::cerr << outcome.GetError().GetMessage() << std::endl;
}

return outcome.IsSuccess();
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To update an item in a table

The following update-item example updates an item in the MusicCollection table. It adds a new attribute (Year) and modifies the AlbumTitle attribute. All of the attributes in the item, as they appear after the update, are returned in the response.

```
aws dynamodb update-item \  
  --table-name MusicCollection \  
  --key file://key.json \  
  --update-expression "SET #Y = :y, #AT = :t" \  
  --expression-attribute-names file://expression-attribute-names.json \  
  --expression-attribute-values file://expression-attribute-values.json \  
  --return-values ALL_NEW \  
  --return-consumed-capacity TOTAL \  
  --return-item-collection-metrics SIZE
```

Contents of key.json:

```
{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"}  
}
```

Contents of expression-attribute-names.json:

```
{  
    "#Y":"Year", "#AT":"AlbumTitle"  
}
```

Contents of expression-attribute-values.json:

```
{  
    ":y":{"N": "2015"},  
    ":t":{"S": "Louder Than Ever"}  
}
```

Output:

```
{  
    "Attributes": {  
        "AlbumTitle": {  
            "S": "Louder Than Ever"  
        },  
        "Awards": {  
            "N": "10"  
        },  
        "Artist": {  
            "S": "Acme Band"  
        },  
        "Year": {  
            "N": "2015"  
        },  
        "SongTitle": {  
            "S": "Happy Day"  
        }  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 3.0  
    "ItemCollectionMetrics": {  
        "ItemCollectionKey": {  
            "Artist": {  
                "S": "Acme Band"  
            }  
        "SizeEstimateRangeGB": [  
            0.0,  
            1.0  
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To update an item conditionally

The following example updates an item in the MusicCollection table, but only if the existing item does not already have a Year attribute.

```
aws dynamodb update-item \
```

```
--table-name MusicCollection \
--key file://key.json \
--update-expression "SET #Y = :y, #AT = :t" \
--expression-attribute-names file://expression-attribute-names.json \
--expression-attribute-values file://expression-attribute-values.json \
--condition-expression "attribute_not_exists(#Y)"
```

Contents of key.json:

```
{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"}  
}
```

Contents of expression-attribute-names.json:

```
{  
    "#Y": "Year",  
    "#AT": "AlbumTitle"  
}
```

Contents of expression-attribute-values.json:

```
{  
    ":y": {"N": "2015"},  
    ":t": {"S": "Louder Than Ever"}  
}
```

If the item already has a Year attribute, DynamoDB returns the following output.

An error occurred (ConditionalCheckFailedException) when calling the UpdateItem operation: The conditional request failed

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [UpdateItem](#) in *AWS CLI Command Reference*.

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
// update
// expression.
func (basics TableBasics) UpdateMovie(movie Movie)
    (map[string]map[string]interface{}, error) {
    var err error
    var response *dynamodb.UpdateItemOutput
    var attributeMap map[string]map[string]interface{}
    update := expression.Set(expression.Name("info.rating"),
        expression.Value(movie.Info["rating"]))
    update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
    expr, err := expression.NewBuilder().WithUpdate(update).Build()
    if err != nil {
        log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
    } else {
        response, err = basics.DynamoDbClient.UpdateItem(context.TODO(),
            &dynamodb.UpdateItemInput{
                TableName:           aws.String(basics.TableName),
                Key:                movie.GetKey(),
```

```
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    UpdateExpression:         expr.Update(),
    ReturnValue:              types.ReturnValueUpdatedNew,
)
if err != nil {
    log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
} else {
    err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
    if err != nil {
        log.Printf("Couldn't unmarshal update response. Here's why: %v\n", err)
    }
}
}

return attributeMap, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}
```

```
// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [UpdateItem in AWS SDK for Go API Reference](#).

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Updates an item in a table using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To update an Amazon DynamoDB table using the AWS SDK for Java V2, its better
```

```
* practice to use the
* Enhanced Client, See the EnhancedModifyItem example.
*/
public class UpdateItem {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key> <keyVal> <name> <updateVal>
            Where:
            tableName - The Amazon DynamoDB table (for example, Music3).
            key - The name of the key in the table (for example, Artist).
            keyVal - The value of the key (for example, Famous Band).
            name - The name of the column where the value is updated (for
example, Awards).
            updateVal - The value used to update an item (for example,
14).

        Example:
            UpdateItem Music3 Artist Famous Band Awards 14
            """;
        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String name = args[3];
        String updateVal = args[4];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
        updateTableItem(ddb, tableName, key, keyVal, name, updateVal);
        ddb.close();
    }

    public static void updateTableItem(DynamoDbClient ddb,
        String tableName,
        String key,
```

```
        String keyVal,
        String name,
        String updateVal) {

    HashMap<String,AttributeValue> itemKey = new HashMap<>();
    itemKey.put(key,AttributeValue.builder()
        .s(keyVal)
        .build());

    HashMap<String,AttributeValueUpdate> updatedValues = new HashMap<>();
    updatedValues.put(name,AttributeValueUpdate.builder()
        .value(AttributeValue.builder().s(updateVal).build())
        .action(AttributeAction.PUT)
        .build());

    UpdateItemRequest request = UpdateItemRequest.builder()
        .tableName(tableName)
        .key(itemKey)
        .attributeUpdates(updatedValues)
        .build();

    try {
        ddb.updateItem(request);
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    System.out.println("The Amazon DynamoDB table was updated!");
}
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [UpdateCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [UpdateItem](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun updateTableItem(
```

```
    tableNameVal: String,  
    keyName: String,  
    keyVal: String,  
    name: String,  
    updateVal: String  
) {  
    val itemKey = mutableMapOf<String, AttributeValue>()  
    itemKey[keyName] = AttributeValue.S(keyVal)  
  
    val updatedValues = mutableMapOf<String, AttributeValueUpdate>()  
    updatedValues[name] = AttributeValueUpdate {  
        value = AttributeValue.S(updateVal)  
        action = AttributeAction.Put  
    }  
  
    val request = UpdateItemRequest {  
        tableName = tableNameVal  
        key = itemKey  
        attributeUpdates = updatedValues  
    }  
  
    DynamoDbClient { region = "us-east-1" }.use { ddb ->  
        ddb.updateItem(request)  
        println("Item in $tableNameVal was updated")  
    }  
}
```

- For API details, see [UpdateItem in AWS SDK for Kotlin API reference](#).

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
echo "What rating would you like to give {$movie['Item']['title']['S']}?  
\n";
```

```
$rating = 0;
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
    $rating = testable_readline("Rating (1-10): ");
}
$service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
$rating);

public function updateItemAttributeByKey(
    string $tableName,
    array $key,
    string $attributeName,
    string $attributeType,
    string $newValue
) {
    $this->dynamoDbClient->updateItem([
        'Key' => $key['Item'],
        'TableName' => $tableName,
        'UpdateExpression' => "set #NV=:NV",
        'ExpressionAttributeNames' => [
            '#NV' => $attributeName,
        ],
        'ExpressionAttributeValues' => [
            ':NV' => [
                $attributeType => $newValue
            ]
        ],
    ]);
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Update an item by using an update expression.

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
  
    def update_movie(self, title, year, rating, plot):  
        """  
        Updates rating and plot data for a movie in the table.  
  
        :param title: The title of the movie to update.  
        :param year: The release year of the movie to update.  
        :param rating: The updated rating to give the movie.  
        :param plot: The updated plot summary to give the movie.  
        :return: The fields that were updated, with their new values.  
        """  
        try:  
            response = self.table.update_item(  
                Key={"year": year, "title": title},  
                UpdateExpression="set info.rating=:r, info.plot=:p",  
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":  
plot},  
                ReturnValues="UPDATED_NEW",  
            )  
        except ClientError as err:  
            logger.error(  
                "Couldn't update movie %s in table %s. Here's why: %s: %s",  
                title,  
                self.table.name,  
                err.response["Error"]["Code"],  
                err.response["Error"]["Message"],  
            )  
            raise  
        else:  
            return response["Attributes"]
```

Update an item by using an update expression that includes an arithmetic operation.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table

    def update_rating(self, title, year, rating_change):
        """
        Updates the quality rating of a movie in the table by using an arithmetic
        operation in the update expression. By specifying an arithmetic
        operation,
        you can adjust a value in a single request, rather than first getting its
        value and then setting its new value.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating_change: The amount to add to the current rating for the
        movie.
        :return: The updated rating.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating = info.rating + :val",
                ExpressionAttributeValues={":val": Decimal(str(rating_change))},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

Update an item only when it meets certain conditions.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table

    def remove_actors(self, title, year, actor_threshold):
        """
        Removes an actor from a movie, but only when the number of actors is
        greater
        than a specified threshold. If the movie does not list more than the
        threshold,
        no actors are removed.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param actor_threshold: The threshold of actors to check.
        :return: The movie data after the update.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="remove info.actors[0]",
                ConditionExpression="size(info.actors) > :num",
                ExpressionAttributeValues={":num": actor_threshold},
                ReturnValues="ALL_NEW",
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
            "ConditionalCheckFailedException":
                logger.warning(
                    "Didn't update %s because it has fewer than %s actors.",
                    title,
                    actor_threshold + 1,
                )
            else:
                logger.error(
                    "Couldn't update movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
```

```
        err.response["Error"]["Message"],
    )
raise
else:
    return response["Attributes"]
```

- For API details, see [UpdateItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
  attr_reader :dynamo_resource
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Updates rating and plot data for a movie in the table.
  #
  # @param movie [Hash] The title, year, plot, rating of the movie.
  def update_item(movie)

    response = @table.update_item(
      key: {"year" => movie[:year], "title" => movie[:title]},
      update_expression: "set info.rating=:r",
      expression_attribute_values: { ":r" => movie[:rating] },
      return_values: "UPDATED_NEW")
    rescue Aws::DynamoDB::Errors::ServiceError => e
```

```
    puts("Couldn't update movie #{movie[:title]} (#{$movie[:year]}) in table  
#{@table.name}\n")  
    puts("\t#{$e.code}: #{$e.message}")  
    raise  
else  
    response.attributes  
end
```

- For API details, see [UpdateItem](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.  
    oo_output = lo_dyn->updateitem(  
        iv_tablename      = iv_table_name  
        it_key            = it_item_key  
        it_attributeupdates = it_attribute_updates ).  
    MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.  
    CATCH /aws1/cx_dyncondalcheckfaile00.  
        MESSAGE 'A condition specified in the operation could not be evaluated.'  
        TYPE 'E'.  
        CATCH /aws1/cx_dyncsourcenotfoundex.  
            MESSAGE 'The table or index does not exist' TYPE 'E'.  
        CATCH /aws1/cx_dyntransactconflictex.  
            MESSAGE 'Another transaction is using the item' TYPE 'E'.  
    ENDTRY.
```

- For API details, see [UpdateItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Update the specified movie with new `rating` and `plot` information.  
///  
/// - Parameters:  
///   - title: The title of the movie to update.  
///   - year: The release year of the movie to update.  
///   - rating: The new rating for the movie.  
///   - plot: The new plot summary string for the movie.  
///  
/// - Returns: An array of mappings of attribute names to their new  
///   listing each item actually changed. Items that didn't need to change  
///   aren't included in this list. `nil` if no changes were made.  
///  
func update(title: String, year: Int, rating: Double? = nil, plot: String? =  
nil) async throws  
    -> [Swift.String:DynamoDBClientTypes.AttributeValue]? {  
guard let client = self.ddbClient else {  
    throw MoviesError.UninitializedClient  
}  
  
// Build the update expression and the list of expression attribute  
// values. Include only the information that's changed.  
  
var expressionParts: [String] = []  
var attrValues: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
```

```
if rating != nil {
    expressionParts.append("info.rating=:r")
    attrValues[":r"] = .n(String(rating!))
}
if plot != nil {
    expressionParts.append("info.plot=:p")
    attrValues[":p"] = .s(plot!)
}
let expression: String = "set \\" + (expressionParts.joined(separator: ", ")) + "\""

let input = UpdateItemInput(
    // Create substitution tokens for the attribute values, to ensure
    // no conflicts in expression syntax.
    expressionAttributeValues: attrValues,
    // The key identifying the movie to update consists of the release
    // year and title.
    key: [
        "year": .n(String(year)),
        "title": .s(title)
    ],
    returnValues: .updatedNew,
    tableName: self.tableName,
    updateExpression: expression
)
let output = try await client.updateItem(input: input)

guard let attributes: [Swift.String:DynamoDBClientTypes.AttributeValue] =
output.attributes else {
    throw MoviesError.InvalidAttributes
}
return attributes
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Delete an item in a DynamoDB table

You can delete items from DynamoDB tables using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on items, see [Core components of Amazon DynamoDB](#).

Delete an item in a DynamoDB table using an AWS SDK

The following code examples show how to delete an item in a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Deletes a single item from a DynamoDB table.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="tableName">The name of the table from which the item
/// will be deleted.</param>
/// <param name="movieToDelete">A movie object containing the title and
/// year of the movie to delete.</param>
/// <returns>A Boolean value indicating the success or failure of the
/// delete operation.</returns>
public static async Task<bool> DeleteItemAsync(
    AmazonDynamoDBClient client,
    string tableName,
    Movie movieToDelete)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = movieToDelete.Title },
        ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
    };

    var request = new DeleteItemRequest
    {
        TableName = tableName,
        Key = key,
    };
}
```

```
        var response = await client.DeleteItemAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- For API details, see [DeleteItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_delete_item
#
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to delete.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_delete_item() {
    local table_name keys response
    local option OPTARG # Required to use getopt command in a function.

    # #####
    # Function usage explanation
    #####
    function usage() {
        echo "function dynamodb_delete_item"
```

```
echo "Delete an item from a DynamoDB table."
echo " -n table_name -- The name of the table."
echo " -k keys -- Path to json file containing the keys that identify the
item to delete."
echo ""
}
while getopts "n:k:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    k) keys="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?) 
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi

iecho "Parameters:\n"
iecho "  table_name: $table_name"
iecho "  keys: $keys"
iecho ""

response=$(aws dynamodb delete-item \
  --table-name "$table_name" \
  --key file://"$keys")
```

```
local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
fi

return 0

}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.

```

```
#  
# The function expects the following argument:  
#       $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#       0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}  
#####
```

- For API details, see [DeleteItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Delete an item from an Amazon DynamoDB table.  
/*!  
 \sa deleteItem()  
 \param tableName: The table name.  
 \param partitionKey: The partition key.  
 \param partitionValue: The value for the partition key.  
 \param clientConfiguration: AWS client configuration.  
 \return bool: Function succeeded.  
 */  
  
bool AwsDoc::DynamoDB::deleteItem(const Aws::String &tableName,  
                                    const Aws::String &partitionKey,  
                                    const Aws::String &partitionValue,  
                                    const Aws::Client::ClientConfiguration  
&clientConfiguration) {  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
  
    Aws::DynamoDB::Model::DeleteItemRequest request;  
  
    request.AddKey(partitionKey,  
                  Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));  
    request.SetTableName(tableName);  
  
    const Aws::DynamoDB::Model::DeleteItemOutcome &outcome =  
        dynamoClient.DeleteItem(  
            request);  
    if (outcome.IsSuccess()) {  
        std::cout << "Item '" << partitionValue << "' deleted!" << std::endl;  
    }  
    else {  
        std::cerr << "Failed to delete item: " << outcome.GetError().GetMessage()  
              << std::endl;  
    }  
  
    return outcome.IsSuccess();  
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To delete an item

The following `delete-item` example deletes an item from the `MusicCollection` table and requests details about the item that was deleted and the capacity used by the request.

```
aws dynamodb delete-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-values ALL_OLD \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `key.json`:

```
{  
    "Artist": {"S": "No One You Know"},  
    "SongTitle": {"S": "Scared of My Shadow"}  
}
```

Output:

```
{  
    "Attributes": {  
        "AlbumTitle": {  
            "S": "Blue Sky Blues"  
        },  
        "Artist": {  
            "S": "No One You Know"  
        },  
        "SongTitle": {  
            "S": "Scared of My Shadow"  
        }  
    },  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 2.0  
    },  
    "ItemCollectionMetrics": {  
        "ItemCollectionKey": {  
    }
```

```
        "Artist": {  
            "S": "No One You Know"  
        },  
        "SizeEstimateRangeGB": [  
            0.0,  
            1.0  
        ]  
    }  
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To delete an item conditionally

The following example deletes an item from the ProductCatalog table only if its ProductCategory is either Sporting Goods or Gardening Supplies and its price is between 500 and 600. It returns details about the item that was deleted.

```
aws dynamodb delete-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"456"}}' \  
  --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (#P  
between :lo and :hi)" \  
  --expression-attribute-names file://names.json \  
  --expression-attribute-values file://values.json \  
  --return-values ALL_OLD
```

Contents of names.json:

```
{  
    "#P": "Price"  
}
```

Contents of values.json:

```
{  
    ":cat1": {"S": "Sporting Goods"},  
    ":cat2": {"S": "Gardening Supplies"},  
    ":lo": {"N": "500"},  
    ":hi": {"N": "600"}  
}
```

Output:

```
{  
    "Attributes": {  
        "Id": {  
            "N": "456"  
        },  
        "Price": {  
            "N": "550"  
        },  
        "ProductCategory": {  
            "S": "Sporting Goods"  
        }  
    }  
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [DeleteItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the  
// examples.  
// It contains a DynamoDB service client that is used to act on the specified  
// table.  
type TableBasics struct {  
    DynamoDbClient *dynamodb.Client  
    TableName      string  
}
```

```
// DeleteMovie removes a movie from the DynamoDB table.
func (basics TableBasics) DeleteMovie(movie Movie) error {
    _, err := basics.DynamoDbClient.DeleteItem(context.TODO(),
    &dynamodb.DeleteItemInput{
        TableName: aws.String(basics.TableName), Key: movie.GetKey(),
    })
    if err != nil {
        log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
        err)
    }
    return err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",

```

```
    movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class DeleteItem {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key> <keyval>
            Where:
        """;
    }
}
```

```
        tableName - The Amazon DynamoDB table to delete the item from  
(for example, Music3).  
        key - The key used in the Amazon DynamoDB table (for example,  
Artist).\s  
        keyval - The key value that represents the item to delete  
(for example, Famous Band).  
        """;  
  
    if (args.length != 3) {  
        System.out.println(usage);  
        System.exit(1);  
    }  
  
    String tableName = args[0];  
    String key = args[1];  
    String keyVal = args[2];  
    System.out.format("Deleting item \"%s\" from %s\n", keyVal, tableName);  
    Region region = Region.US_EAST_1;  
    DynamoDbClient ddb = DynamoDbClient.builder()  
        .region(region)  
        .build();  
  
    deleteDynamoDBItem(ddb, tableName, key, keyVal);  
    ddb.close();  
}  
  
public static void deleteDynamoDBItem(DynamoDbClient ddb, String tableName,  
String key, String keyVal) {  
    HashMap<String,AttributeValue> keyToGet = new HashMap<>();  
    keyToGet.put(key, AttributeValue.builder()  
        .s(keyVal)  
        .build());  
  
    DeleteItemRequest deleteReq = DeleteItemRequest.builder()  
        .tableName(tableName)  
        .key(keyToGet)  
        .build();  
  
    try {  
        ddb.deleteItem(deleteReq);  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    }
```

```
    }  
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [DeleteCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";  
import { DynamoDBDocumentClient, DeleteCommand } from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
  const command = new DeleteCommand({  
    TableName: "Sodas",  
    Key: {  
      Flavor: "Cola",  
    },  
  });  
  
  const response = await docClient.send(command);  
  console.log(response);  
  return response;  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an item from a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

Delete an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

docClient.delete(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteItem](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun deleteDynamoDBItem(tableNameVal: String, keyName: String, keyVal: String) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request = DeleteItemRequest {
        tableName = tableNameVal
        key = keyToGet
    }
```

```
DynamoDbClient { region = "us-east-1" }.use { ddb ->
    ddb.deleteItem(request)
    println("Item with key matching $keyVal was deleted")
}
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$key = [
    'Item' => [
        'title' => [
            'S' => $movieName,
        ],
        'year' => [
            'N' => $movieYear,
        ],
    ]
];

@Service->deleteItemByKey($tableName, $key);
echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";

public function deleteItemByKey(string $tableName, array $key)
{
    $this->dynamoDbClient->deleteItem([
        'Key' => $key['Item'],
        'TableName' => $tableName,
    ]);
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
    def delete_movie(self, title, year):  
        """  
        Deletes a movie from the table.  
  
        :param title: The title of the movie to delete.  
        :param year: The release year of the movie to delete.  
        """  
        try:  
            self.table.delete_item(Key={"year": year, "title": title})  
        except ClientError as err:  
            logger.error(  
                "Couldn't delete movie %s. Here's why: %s: %s",  
                title,  
                err.response["Error"]["Code"],  
                err.response["Error"]["Message"],
```

```
)  
raise
```

You can specify a condition so that an item is deleted only when it meets certain criteria.

```
class UpdateQueryWrapper:  
    def __init__(self, table):  
        self.table = table  
  
    def delete_underrated_movie(self, title, year, rating):  
        """  
        Deletes a movie only if it is rated below a specified value. By using a  
        condition expression in a delete operation, you can specify that an item  
        is  
        deleted only when it meets certain criteria.  
  
        :param title: The title of the movie to delete.  
        :param year: The release year of the movie to delete.  
        :param rating: The rating threshold to check before deleting the movie.  
        """  
        try:  
            self.table.delete_item(  
                Key={"year": year, "title": title},  
                ConditionExpression="info.rating <= :val",  
                ExpressionAttributeValues={":val": Decimal(str(rating))},  
            )  
        except ClientError as err:  
            if err.response["Error"]["Code"] ==  
                "ConditionalCheckFailedException":  
                logger.warning(  
                    "Didn't delete %s because its rating is greater than %s.",  
                    title,  
                    rating,  
                )  
            else:  
                logger.error(  
                    "Couldn't delete movie %s. Here's why: %s: %s",  
                    title,  
                    err.response["Error"]["Code"],  
                    err.response["Error"]["Message"],  
                )
```

```
    )  
    raise
```

- For API details, see [DeleteItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics  
  attr_reader :dynamo_resource  
  attr_reader :table  
  
  def initialize(table_name)  
    client = Aws::DynamoDB::Client.new(region: "us-east-1")  
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)  
    @table = @dynamo_resource.table(table_name)  
  end  
  
  # Deletes a movie from the table.  
  #  
  # @param title [String] The title of the movie to delete.  
  # @param year [Integer] The release year of the movie to delete.  
  def delete_item(title, year)  
    @table.delete_item(key: {"year" => year, "title" => title})  
  rescue Aws::DynamoDB::Errors::ServiceError => e  
    puts("Couldn't delete movie #{title}. Here's why:")  
    puts("\t#{e.code}: #{e.message}")  
    raise  
  end
```

- For API details, see [DeleteItem](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn delete_item(
    client: &Client,
    table: &str,
    key: &str,
    value: &str,
) -> Result<DeleteItemOutput, Error> {
    match client
        .delete_item()
        .table_name(table)
        .key(key, AttributeValue::S(value.into()))
        .send()
        .await
    {
        Ok(out) => {
            println!("Deleted item from table");
            Ok(out)
        }
        Err(e) => Err(Error::unhandled(e)),
    }
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.  
  DATA(lo_resp) = lo_dyn->deleteitem(  
    iv_tablename          = iv_table_name  
    it_key                = it_key_input ).  
  MESSAGE 'Deleted one item.' TYPE 'I'.  
  CATCH /aws1/cx_dyncondalcheckfaile00.  
    MESSAGE 'A condition specified in the operation could not be evaluated.'  
    TYPE 'E'.  
    CATCH /aws1/cx_dynresourcenotfoundex.  
      MESSAGE 'The table or index does not exist' TYPE 'E'.  
    CATCH /aws1/cx_dyntransactconflictex.  
      MESSAGE 'Another transaction is using the item' TYPE 'E'.  
  ENDTRY.
```

- For API details, see [DeleteItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Delete a movie, given its title and release year.  
///  
/// - Parameters:  
///   - title: The movie's title.  
///   - year: The movie's release year.  
///  
func delete(title: String, year: Int) async throws {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = DeleteItemInput(  
        key: [  
            "year": .n(String(year)),  
            "title": .s(title)  
        ],  
        tableName: self.tableName  
    )  
    _ = try await client.deleteItem(input: input)  
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Query a DynamoDB table

You can perform a query on a DynamoDB table using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on queries, see [Query operations in DynamoDB](#).

Query a DynamoDB table using an AWS SDK

The following code examples show how to query a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public static async Task<int> QueryMoviesAsync(AmazonDynamoDBClient
client, string tableName, int year)
{
    var movieTable = Table.LoadTable(client, tableName);
    var filter = new QueryFilter("year", QueryOperator.Equal, year);

    Console.WriteLine("\nFind movies released in: {year}:");

    var config = new QueryOperationConfig()
    {
        Limit = 10, // 10 items per page.
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string>
        {
            "title",
            "year",
        },
        ConsistentRead = true,
        Filter = filter,
    };

    // Value used to track how many movies match the
```

```
// supplied criteria.  
var moviesFound = 0;  
  
Search search = movieTable.Query(config);  
do  
{  
    var movieList = await search.GetNextSetAsync();  
    moviesFound += movieList.Count;  
  
    foreach (var movie in movieList)  
    {  
        DisplayDocument(movie);  
    }  
}  
while (!search.IsDone);  
  
return moviesFound;  
}
```

- For API details, see [Query](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####  
# function dynamodb_query  
#  
# This function queries a DynamoDB table.  
#  
# Parameters:  
#     -n table_name -- The name of the table.  
#     -k key_condition_expression -- The key condition expression.  
#     -a attribute_names -- Path to JSON file containing the attribute names.
```

```
#      -v attribute_values -- Path to JSON file containing the attribute values.
#      [-p projection_expression] -- Optional projection expression.
#
# Returns:
#      The items as json output.
# And:
#      0 - If successful.
#      1 - If it fails.
#####
function dynamodb_query() {
    local table_name key_condition_expression attribute_names attribute_values
    projection_expression response
    local option OPTARG # Required to use getopts command in a function.

    #####
    # Function usage explanation
#####

    function usage() {
        echo "function dynamodb_query"
        echo "Query a DynamoDB table."
        echo " -n table_name -- The name of the table."
        echo " -k key_condition_expression -- The key condition expression."
        echo " -a attribute_names -- Path to JSON file containing the attribute
names."
        echo " -v attribute_values -- Path to JSON file containing the attribute
values."
        echo " [-p projection_expression] -- Optional projection expression."
        echo ""
    }

    while getopts "n:k:a:v:p:h" option; do
        case "${option}" in
            n) table_name="${OPTARG}" ;;
            k) key_condition_expression="${OPTARG}" ;;
            a) attribute_names="${OPTARG}" ;;
            v) attribute_values="${OPTARG}" ;;
            p) projection_expression="${OPTARG}" ;;
            h)
                usage
                return 0
                ;;
            \?) 
                echo "Invalid parameter"
                usage
                ;;
        esac
    done
}
```

```
        return 1
    ;;
esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$key_condition_expression" ]]; then
    errecho "ERROR: You must provide a key condition expression with the -k parameter."
    usage
    return 1
fi

if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
fi

if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
fi

if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
    --table-name "$table_name" \
    --key-condition-expression "$key_condition_expression" \
    --expression-attribute-names file://"$attribute_names" \
    --expression-attribute-values file://"$attribute_values")
else
    response=$(aws dynamodb query \
    --table-name "$table_name" \
    --key-condition-expression "$key_condition_expression" \
    --expression-attribute-names file://"$attribute_names" \
    --expression-attribute-values file://"$attribute_values" \
    --projection-expression "$projection_expression")
```

```
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
fi

echo "$response"

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#     $1 - The error code returned by the AWS CLI.
#
# Returns:
#     0: - Success.
#
#####
function aws_cli_error_log() {
```

```
local err_code=$1
errecho "Error code : $err_code"
if [ "$err_code" == 1 ]; then
    errecho " One or more S3 transfers failed."
elif [ "$err_code" == 2 ]; then
    errecho " Command line failed to parse."
elif [ "$err_code" == 130 ]; then
    errecho " Process received SIGINT."
elif [ "$err_code" == 252 ]; then
    errecho " Command syntax invalid."
elif [ "$err_code" == 253 ]; then
    errecho " The system environment or configuration was invalid."
elif [ "$err_code" == 254 ]; then
    errecho " The service returned an error."
elif [ "$err_code" == 255 ]; then
    errecho " 255 is a catch-all error."
fi

return 0
}
```

- For API details, see [Query](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Perform a query on an Amazon DynamoDB Table and retrieve items.
/*!
 \sa queryItem()
 \param tableName: The table name.
 \param partitionKey: The partition key.
 \param partitionValue: The value for the partition key.
 \param projectionExpression: The projections expression, which is ignored if
 empty.
```

```
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/



/*
 * The partition key attribute is searched with the specified value. By default,
all fields and values
 * contained in the item are returned. If an optional projection expression is
 * specified on the command line, only the specified fields and values are
 * returned.
*/



bool AwsDoc::DynamoDB::queryItems(const Aws::String &tableName,
                                    const Aws::String &partitionKey,
                                    const Aws::String &partitionValue,
                                    const Aws::String &projectionExpression,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::QueryRequest request;

    request.SetTableName(tableName);

    if (!projectionExpression.empty()) {
        request.SetProjectionExpression(projectionExpression);
    }

    // Set query key condition expression.
    request.SetKeyConditionExpression(partitionKey + " = :valueToMatch");

    // Set Expression AttributeValues.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> attributeValues;
    attributeValues.emplace(":valueToMatch", partitionValue);

    request.SetExpressionAttributeValues(attributeValues);

    bool result = true;

    // "exclusiveStartKey" is used for pagination.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
    do {
        if (!exclusiveStartKey.empty()) {
            request.SetExclusiveStartKey(exclusiveStartKey);
```

```
        exclusiveStartKey.clear();
    }
    // Perform Query operation.
    const Aws::DynamoDB::Model::QueryOutcome &outcome =
dynamoClient.Query(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved items.
        const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
        if (!items.empty()) {
            std::cout << "Number of items retrieved from Query: " <<
items.size()
                << std::endl;
            // Iterate each item and print.
            for (const auto &item: items) {
                std::cout
                    <<
"*****"
                    << std::endl;
                // Output each retrieved field and its value.
                for (const auto &i: item)
                    std::cout << i.first << ":" << i.second.GetS() <<
std::endl;
            }
        }
        else {
            std::cout << "No item found in table: " << tableName <<
std::endl;
        }
    }

        exclusiveStartKey = outcome.GetResult().GetLastEvaluatedKey();
}
else {
    std::cerr << "Failed to Query items: " <<
outcome.GetError().GetMessage();
    result = false;
    break;
}
} while (!exclusiveStartKey.empty());

return result;
}
```

- For API details, see [Query](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To query a table

The following query example queries items in the MusicCollection table. The table has a hash-and-range primary key (Artist and SongTitle), but this query only specifies the hash key value. It returns song titles by the artist named "No One You Know".

```
aws dynamodb query \  
    --table-name MusicCollection \  
    --projection-expression "SongTitle" \  
    --key-condition-expression "Artist = :v1" \  
    --expression-attribute-values file://expression-attributes.json \  
    --return-consumed-capacity TOTAL
```

Contents of expression-attributes.json:

```
{  
    ":v1": {"S": "No One You Know"}  
}
```

Output:

```
{  
    "Items": [  
        {  
            "SongTitle": {  
                "S": "Call Me Today"  
            },  
            "SongTitle": {  
                "S": "Scared of My Shadow"  
            }  
        }  
    ],  
    "Count": 2,  
    "ScannedCount": 2,  
    "ConsumedCapacity": {
```

```
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To query a table using strongly consistent reads and traverse the index in descending order

The following example performs the same query as the first example, but returns results in reverse order and uses strongly consistent reads.

```
aws dynamodb query \
--table-name MusicCollection \
--projection-expression "SongTitle" \
--key-condition-expression "Artist = :v1" \
--expression-attribute-values file://expression-attributes.json \
--consistent-read \
--no-scan-index-forward \
--return-consumed-capacity TOTAL
```

Contents of expression-attributes.json:

```
{
  ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
  "Items": [
    {
      "SongTitle": {
        "S": "Scared of My Shadow"
      }
    },
    {
      "SongTitle": {
        "S": "Call Me Today"
      }
    }
  ]
}
```

```
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To filter out specific results

The following example queries the MusicCollection but excludes results with specific values in the AlbumTitle attribute. Note that this does not affect the ScannedCount or ConsumedCapacity, because the filter is applied after the items have been read.

```
aws dynamodb query \
--table-name MusicCollection \
--key-condition-expression "#n1 = :v1" \
--filter-expression "NOT (#n2 IN (:v2, :v3))" \
--expression-attribute-names file://names.json \
--expression-attribute-values file://values.json \
--return-consumed-capacity TOTAL
```

Contents of values.json:

```
{
    ":v1": {"S": "No One You Know"},
    ":v2": {"S": "Blue Sky Blues"},
    ":v3": {"S": "Greatest Hits"}
}
```

Contents of names.json:

```
{
    "#n1": "Artist",
    "#n2": "AlbumTitle"
}
```

```
}
```

Output:

```
{  
    "Items": [  
        {  
            "AlbumTitle": {  
                "S": "Somewhat Famous"  
            },  
            "Artist": {  
                "S": "No One You Know"  
            },  
            "SongTitle": {  
                "S": "Call Me Today"  
            }  
        }  
    ],  
    "Count": 1,  
    "ScannedCount": 2,  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 0.5  
    }  
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 4: To retrieve only an item count

The following example retrieves a count of items matching the query, but does not retrieve any of the items themselves.

```
aws dynamodb query \  
    --table-name MusicCollection \  
    --select COUNT \  
    --key-condition-expression "Artist = :v1" \  
    --expression-attribute-values file://expression-attributes.json
```

Contents of expression-attributes.json:

```
{  
  ":v1": {"S": "No One You Know"}  
}
```

Output:

```
{  
  "Count": 2,  
  "ScannedCount": 2,  
  "ConsumedCapacity": null  
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 5: To query an index

The following example queries the local secondary index `AlbumTitleIndex`. The query returns all attributes from the base table that have been projected into the local secondary index. Note that when querying a local secondary index or global secondary index, you must also provide the name of the base table using the `table-name` parameter.

```
aws dynamodb query \  
  --table-name MusicCollection \  
  --index-name AlbumTitleIndex \  
  --key-condition-expression "Artist = :v1" \  
  --expression-attribute-values file://expression-attributes.json \  
  --select ALL_PROJECTED_ATTRIBUTES \  
  --return-consumed-capacity INDEXES
```

Contents of `expression-attributes.json`:

```
{  
  ":v1": {"S": "No One You Know"}  
}
```

Output:

```
{
```

```
"Items": [
    {
        "AlbumTitle": {
            "S": "Blue Sky Blues"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Scared of My Shadow"
        }
    },
    {
        "AlbumTitle": {
            "S": "Somewhat Famous"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
],
"Count": 2,
"ScannedCount": 2,
"ConsumedCapacity": {
    "TableName": "MusicCollection",
    "CapacityUnits": 0.5,
    "Table": {
        "CapacityUnits": 0.0
    },
    "LocalSecondaryIndexes": {
        "AlbumTitleIndex": {
            "CapacityUnits": 0.5
        }
    }
}
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [Query](#) in *AWS CLI Command Reference*.

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// Query gets all movies in the DynamoDB table that were released in the
// specified year.
// The function uses the `expression` package to build the key condition
// expression
// that is used in the query.
func (basics TableBasics) Query(releaseYear int) ([]Movie, error) {
    var err error
    var response *dynamodb.QueryOutput
    var movies []Movie
    keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
    expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
    if err != nil {
        log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
    } else {
        queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
&dynamodb.QueryInput{
    TableName:           aws.String(basics.TableName),
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    KeyConditionExpression:   expr.KeyCondition(),
```

```
        })
    for queryPaginator.HasMorePages() {
        response, err = queryPaginator.NextPage(context.TODO())
        if err != nil {
            log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
            releaseYear, err)
            break
        } else {
            var moviePage []Movie
            err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
            if err != nil {
                log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
                break
            } else {
                movies = append(movies, moviePage...)
            }
        }
    }
    return movies, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
```

```
    panic(err)
}
return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [Query](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Queries a table by using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
*
* To query items from an Amazon DynamoDB table using the AWS SDK for Java V2,
* its better practice to use the
* Enhanced Client. See the EnhancedQueryRecords example.
*/
public class Query {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <partitionKeyName> <partitionKeyVal>
            Where:
            tableName - The Amazon DynamoDB table to put the item in (for
            example, Music3).
            partitionKeyName - The partition key name of the Amazon
            DynamoDB table (for example, Artist).
            partitionKeyVal - The value of the partition key that should
            match (for example, Famous Band).
            """;
        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }
        String tableName = args[0];
        String partitionKeyName = args[1];
        String partitionKeyVal = args[2];
        // For more information about an alias, see:
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
        Expressions.ExpressionAttributeNames.html
        String partitionAlias = "#a";
        System.out.format("Querying %s", tableName);
        System.out.println("");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
    }
}
```

```
        int count = queryTable(ddb, tableName, partitionKeyName, partitionKeyVal,
partitionAlias);
        System.out.println("There were " + count + " record(s) returned");
        ddb.close();
    }

    public static int queryTable(DynamoDbClient ddb, String tableName, String
partitionKeyName, String partitionKeyVal,
        String partitionAlias) {
        // Set up an alias for the partition key name in case it's a reserved
word.
        HashMap<String, String> attrNameAlias = new HashMap<String, String>();
        attrNameAlias.put(partitionAlias, partitionKeyName);

        // Set up mapping of the partition name with the value.
        HashMap<String, AttributeValue> attrValues = new HashMap<>();
        attrValues.put ":" + partitionKeyName, AttributeValue.builder()
            .s(partitionKeyVal)
            .build());

        QueryRequest queryReq = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(partitionAlias + " = :" +
partitionKeyName)
            .expressionAttributeNames(attrNameAlias)
            .expressionAttributeValues(attrValues)
            .build();

        try {
            QueryResponse response = ddb.query(queryReq);
            return response.count();

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return -1;
    }
}
```

Queries a table by using `DynamoDbClient` and a secondary index.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * Create the Movies table by running the Scenario example and loading the Movie
 * data from the JSON file. Next create a secondary
 * index for the Movies table that uses only the year column. Name the index
 * **year-index**. For more information, see:
 *
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
 */
public class QueryItemsUsingIndex {
    public static void main(String[] args) {
        String tableName = "Movies";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        queryIndex(ddb, tableName);
        ddb.close();
    }

    public static void queryIndex(DynamoDbClient ddb, String tableName) {
        try {
            Map<String, String> expressionAttributeNames = new HashMap<>();
            expressionAttributeNames.put("#year", "year");
            Map<String, AttributeValue> expressionAttributeValues = new
            HashMap<>();
        
```

```
        expressionAttributeValues.put(":yearValue",
AttributeValue.builder().n("2013").build());

QueryRequest request = QueryRequest.builder()
        .tableName(tableName)
        .indexName("year-index")
        .keyConditionExpression("#year = :yearValue")
        .expressionAttributeNames(expressionAttributesNames)
        .expressionAttributeValues(expressionAttributeValues)
        .build();

System.out.println("== Movie Titles ==");
QueryResponse response = ddb.query(request);
response.items()
        .forEach(movie ->
System.out.println(movie.get("title").s()));

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

- For API details, see [Query](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [QueryCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
    const command = new QueryCommand({  
        TableName: "CoffeeCrop",  
        KeyConditionExpression:  
            "OriginCountry = :originCountry AND RoastDate > :roastDate",  
        ExpressionAttributeValues: {  
            ":originCountry": "Ethiopia",  
            ":roastDate": "2023-05-01",  
        },  
        ConsistentRead: true,  
    });  
  
    const response = await docClient.send(command);  
    console.log(response);  
    return response;  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Query](#) in [AWS SDK for JavaScript API Reference](#).

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create DynamoDB document client  
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });  
  
var params = {
```

```
        ExpressionAttributeValues: {
          ":s": 2,
          ":e": 9,
          ":topic": "PHRASE",
        },
        KeyConditionExpression: "Season = :s and Episode > :e",
        FilterExpression: "contains (Subtitle, :topic)",
        TableName: "EPISODES_TABLE",
      };

      docClient.query(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else {
          console.log("Success", data.Items);
        }
      });
    };
  };
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Query](#) in [AWS SDK for JavaScript API Reference](#).

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun queryDynTable(
  tableNameVal: String,
  partitionKeyName: String,
  partitionKeyVal: String,
  partitionAlias: String
): Int {
  val attrNameAlias = mutableMapOf<String, String>()
  attrNameAlias[partitionAlias] = partitionKeyName
}
```

```
// Set up mapping of the partition name with the value.  
val attrValues = mutableMapOf<String, AttributeValue>()  
attrValues[":$partitionKeyName"] = AttributeValue.S(partitionKeyVal)  
  
val request = QueryRequest {  
    tableName = tableNameVal  
    keyConditionExpression = "$partitionAlias = :$partitionKeyName"  
    expressionAttributeNames = attrNameAlias  
    this.expressionAttributeValues = attrValues  
}  
  
DynamoDbClient { region = "us-east-1" }.use { ddb ->  
    val response = ddb.query(request)  
    return response.count  
}  
}
```

- For API details, see [Query](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$birthKey = [  
    'Key' => [  
        'year' => [  
            'N' => "$birthYear",  
        ],  
    ],  
];  
$result = $service->query($tableName, $birthKey);  
  
public function query(string $tableName, $key)  
{  
    $expressionAttributeValues = [];
```

```
$expressionAttributeNames = [];
$keyConditionExpression = "";
$index = 1;
foreach ($key as $name => $value) {
    $keyConditionExpression .= "#" . array_key_first($value) . " = :v" .
    $index ",";
    $expressionAttributeNames["#" . array_key_first($value)] =
    array_key_first($value);
    $hold = array_pop($value);
    $expressionAttributeValues[":v$index"] = [
        array_key_first($hold) => array_pop($hold),
    ];
}
$keyConditionExpression = substr($keyConditionExpression, 0, -1);
$query = [
    'ExpressionAttributeValues' => $expressionAttributeValues,
    'ExpressionAttributeNames' => $expressionAttributeNames,
    'KeyConditionExpression' => $keyConditionExpression,
    'TableName' => $tableName,
];
return $this->dynamoDbClient->query($query);
}
```

- For API details, see [Query](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Query items by using a key condition expression.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
```

```
"""
:param dyn_resource: A Boto3 DynamoDB resource.
"""

self.dyn_resource = dyn_resource
# The table variable is set during the scenario in the call to
# 'exists' if the table exists. Otherwise, it is set by 'create_table'.
self.table = None


def query_movies(self, year):
    """
    Queries for movies that were released in the specified year.

    :param year: The year to query.
    :return: The list of movies that were released in the specified year.
    """

    try:
        response =
self.table.query(KeyConditionExpression=Key("year").eq(year))
    except ClientError as err:
        logger.error(
            "Couldn't query for movies released in %s. Here's why: %s: %s",
            year,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Items"]
```

Query items and project them to return a subset of data.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def query_and_project_movies(self, year, title_bounds):
        """
        Query for movies that were released in a specified year and that have
titles
```

that start within a range of letters. A projection expression is used to return a subset of data for each movie.

```
:param year: The release year to query.
:param title_bounds: The range of starting letters to query.
:return: The list of movies.
"""
try:
    response = self.table.query(
        ProjectionExpression="#yr, title, info.genres, info.actors[0]",
        ExpressionAttributeNames={"#yr": "year"},
        KeyConditionExpression=(
            Key("year").eq(year)
            & Key("title").between(
                title_bounds["first"], title_bounds["second"]
            )
        ),
    )
except ClientError as err:
    if err.response["Error"]["Code"] == "ValidationException":
        logger.warning(
            "There's a validation error. Here's the message: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
    else:
        logger.error(
            "Couldn't query for movies. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
else:
    return response["Items"]
```

- For API details, see [Query](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Queries for movies that were released in the specified year.
    #
    # @param year [Integer] The year to query.
    # @return [Array] The list of movies that were released in the specified year.
    def query_items(year)
        response = @table.query(
            key_condition_expression: "#yr = :year",
            expression_attribute_names: {"#yr" => "year"},
            expression_attribute_values: {":year" => year})
        rescue Aws::DynamoDB::Errors::ServiceError => e
            puts("Couldn't query for movies released in #{year}. Here's why:")
            puts("\t#{e.code}: #{e.message}")
            raise
        else
            response.items
        end
    end
```

- For API details, see [Query](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Find the movies made in the specified year.

```
pub async fn movies_in_year(
    client: &Client,
    table_name: &str,
    year: u16,
) -> Result<Vec<Movie>, MovieError> {
    let results = client
        .query()
        .table_name(table_name)
        .key_condition_expression("#yr = :yyyy")
        .expression_attribute_names("#yr", "year")
        .expression_attribute_values(":yyyy",
AttributeValue::N(year.to_string())))
        .send()
        .await?;

    if let Some(items) = results.items {
        let movies = items.iter().map(|v| v.into()).collect();
        Ok(movies)
    } else {
        Ok(vec![])
    }
}
```

- For API details, see [Query](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
" Query movies for a given year .
DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
    ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_year }| ) ) ).
DATA(lt_key_conditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
    ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
        key = 'year'
        value = NEW /aws1/cl_dyncondition(
            it_attributevaluelist = lt_attributelist
            iv_comparisonoperator = |EQ|
        ) ) ) .
oo_result = lo_dyn->query(
    iv_tablename = iv_table_name
    it_keyconditions = lt_key_conditions ).
DATA(lt_items) = oo_result->get_items( ).
"You can loop over the results to get item attributes.
LOOP AT lt_items INTO DATA(lt_item).
    DATA(lo_title) = lt_item[ key = 'title' ]-value.
    DATA(lo_year) = lt_item[ key = 'year' ]-value.
ENDLOOP.
DATA(lv_count) = oo_result->get_count( ).
MESSAGE 'Item count is: ' && lv_count TYPE 'I'.
CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table or index does not exist' TYPE 'E'.
ENDTRY.
```

- For API details, see [Query](#) in AWS SDK for SAP ABAP API reference.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Get all the movies released in the specified year.  
///  
/// - Parameter year: The release year of the movies to return.  
///  
/// - Returns: An array of `Movie` objects describing each matching movie.  
  
func getMovies(fromYear year: Int) async throws -> [Movie] {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = QueryInput(  
        expressionAttributeNames: [  
            "#y": "year"  
        ],  
        expressionAttributeValues: [  
            ":y": .n(String(year))  
        ],  
        keyConditionExpression: "#y = :y",  
        tableName: self.tableName  
    )  
    let output = try await client.query(input: input)  
  
    guard let items = output.items else {  
        throw MoviesError.ItemNotFound  
    }  
    return items  
}
```

```
}

// Convert the found movies into `Movie` objects and return an array
// of them.

var movieList: [Movie] = []
for item in items {
    let movie = try Movie(withItem: item)
    movieList.append(movie)
}
return movieList
}
```

- For API details, see [Query](#) in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Scan a DynamoDB table

You can perform a scan on a DynamoDB table using the AWS Management Console, the AWS CLI, or an AWS SDK. For more information on scans, see [Working with scans in DynamoDB](#).

Scan a DynamoDB table using an AWS SDK

The following code examples show how to scan a DynamoDB table using an AWS SDK.

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public static async Task<int> ScanTableAsync(
    AmazonDynamoDBClient client,
```

```
        string tableName,
        int startYear,
        int endYear)
{
    var request = new ScanRequest
    {
        TableName = tableName,
        ExpressionAttributeNames = new Dictionary<string, string>
        {
            { "#yr", "year" },
        },
        ExpressionAttributeValues = new Dictionary<string,
AttributeValue>
        {
            { ":y_a", new AttributeValue { N = startYear.ToString() } },
            { ":y_z", new AttributeValue { N = endYear.ToString() } },
        },
        FilterExpression = "#yr between :y_a and :y_z",
        ProjectionExpression = "#yr, title, info.actors[0],
info.directors, info.running_time_secs",
        Limit = 10 // Set a limit to demonstrate using the
LastEvaluatedKey.
    };
}

// Keep track of how many movies were found.
int foundCount = 0;

var response = new ScanResponse();
do
{
    response = await client.ScanAsync(request);
    foundCount += response.Items.Count;
    response.Items.ForEach(i => DisplayItem(i));
    request.ExclusiveStartKey = response.LastEvaluatedKey;
}
while (response.LastEvaluatedKey.Count > 0);
return foundCount;
}
```

- For API details, see [Scan](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_scan
#
# This function scans a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -f filter_expression -- The filter expression.
#     -a expression_attribute_names -- Path to JSON file containing the
#         expression attribute names.
#     -v expression_attribute_values -- Path to JSON file containing the
#         expression attribute values.
#     [-p projection_expression] -- Optional projection expression.
#
# Returns:
#     The items as json output.
# And:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_scan() {
    local table_name filter_expression expression_attribute_names
    expression_attribute_values projection_expression response
    local option OPTARG # Required to use getopt command in a function.

    # #####
    # Function usage explanation
    #####
    function usage() {
        echo "function dynamodb_scan"
        echo "Scan a DynamoDB table."
        echo " -n table_name -- The name of the table."
    }
}
```

```
echo " -f filter_expression -- The filter expression."
echo " -a expression_attribute_names -- Path to JSON file containing the
expression attribute names."
echo " -v expression_attribute_values -- Path to JSON file containing the
expression attribute values."
echo " [-p projection_expression] -- Optional projection expression."
echo ""

}

while getopts "n:f:a:v:p:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    f) filter_expression="${OPTARG}" ;;
    a) expression_attribute_names="${OPTARG}" ;;
    v) expression_attribute_values="${OPTARG}" ;;
    p) projection_expression="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?)
      echo "Invalid parameter"
      usage
      return 1
      ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$filter_expression" ]]; then
  errecho "ERROR: You must provide a filter expression with the -f parameter."
  usage
  return 1
fi

if [[ -z "$expression_attribute_names" ]]; then
  errecho "ERROR: You must provide expression attribute names with the -a
parameter."
```

```
usage
return 1
fi

if [[ -z "$expression_attribute_values" ]]; then
    errecho "ERROR: You must provide expression attribute values with the -v
parameter."
    usage
    return 1
fi

if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb scan \
        --table-name "$table_name" \
        --filter-expression "$filter_expression" \
        --expression-attribute-names file://"$expression_attribute_names" \
        --expression-attribute-values file://"$expression_attribute_values")
else
    response=$(aws dynamodb scan \
        --table-name "$table_name" \
        --filter-expression "$filter_expression" \
        --expression-attribute-names file://"$expression_attribute_names" \
        --expression-attribute-values file://"$expression_attribute_values" \
        --projection-expression "$projection_expression")
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports scan operation failed.$response"
    return 1
fi

echo "$response"

return 0
}
```

The utility functions used in this example.

```
#####
#####
```

```
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#       $1 - The error code returned by the AWS CLI.
#
# Returns:
#       0: - Success.
#
#####

function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho " The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho " 255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [Scan](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Scan an Amazon DynamoDB table.
/*
\sa scanTable()
\param tableName: Name for the DynamoDB table.
\param projectionExpression: An optional projection expression, ignored if empty.
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/

bool AwsDoc::DynamoDB::scanTable(const Aws::String &tableName,
                                  const Aws::String &projectionExpression,
                                  const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::ScanRequest request;
    request.SetTableName(tableName);

    if (!projectionExpression.empty())
        request.SetProjectionExpression(projectionExpression);

    // Perform scan on table.
    const Aws::DynamoDB::Model::ScanOutcome &outcome =
dynamoClient.Scan(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved items.
        const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
    }
}
```

```
if (!items.empty()) {
    std::cout << "Number of items retrieved from scan: " << items.size()
        << std::endl;
    // Iterate each item and print.
    for (const Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue> &itemMap: items) {
        std::cout <<
"*****"
        << std::endl;
        // Output each retrieved field and its value.
        for (const auto &itemEntry: itemMap)
            std::cout << itemEntry.first << ":" <<
itemEntry.second.GetS()
                << std::endl;
    }
}

else {
    std::cout << "No item found in table: " << tableName << std::endl;
}
else {
    std::cerr << "Failed to Scan items: " << outcome.GetError().GetMessage()
        << std::endl;
}

return outcome.IsSuccess();
}
```

- For API details, see [Scan](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To scan a table

The following scan example scans the entire MusicCollection table, and then narrows the results to songs by the artist "No One You Know". For each item, only the album title and song title are returned.

```
aws dynamodb scan \
    --table-name MusicCollection \
    --filter-expression "Artist = :a" \
    --projection-expression "#ST, #AT" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json
```

Contents of expression-attribute-names.json:

```
{  
    "#ST": "SongTitle",  
    "#AT": "AlbumTitle"  
}
```

Contents of expression-attribute-values.json:

```
{  
    ":a": {"S": "No One You Know"}  
}
```

Output:

```
{  
    "Count": 2,  
    "Items": [  
        {  
            "SongTitle": {  
                "S": "Call Me Today"  
            },  
            "AlbumTitle": {  
                "S": "Somewhat Famous"  
            }  
        },  
        {  
            "SongTitle": {  
                "S": "Scared of My Shadow"  
            },  
            "AlbumTitle": {  
                "S": "Blue Sky Blues"  
            }  
        }  
    ],  
}
```

```
        "ScannedCount": 3,  
        "ConsumedCapacity": null  
    }
```

For more information, see [Working with Scans in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [Scan](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the  
// examples.  
// It contains a DynamoDB service client that is used to act on the specified  
// table.  
type TableBasics struct {  
    DynamoDbClient *dynamodb.Client  
    TableName      string  
}  
  
  
// Scan gets all movies in the DynamoDB table that were released in a range of  
// years  
// and projects them to return a reduced set of fields.  
// The function uses the `expression` package to build the filter and projection  
// expressions.  
func (basics TableBasics) Scan(startYear int, endYear int) ([]Movie, error) {  
    var movies []Movie  
    var err error  
    var response *dynamodb.ScanOutput  
    filtEx := expression.Name("year").Between(expression.Value(startYear),  
        expression.Value(endYear))
```

```
projEx := expression.NamesList(
    expression.Name("year"), expression.Name("title"),
    expression.Name("info.rating"))
expr, err :=
    expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
if err != nil {
    log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
} else {
    scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
&dynamodb.ScanInput{
    TableName:                 aws.String(basics.TableName),
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    FilterExpression:         expr.Filter(),
    ProjectionExpression:     expr.Projection(),
})
for scanPaginator.HasMorePages() {
    response, err = scanPaginator.NextPage(context.TODO())
    if err != nil {
        log.Printf("Couldn't scan for movies released between %v and %v. Here's why:
%v\n",
            startYear, endYear, err)
        break
    } else {
        var moviePage []Movie
        err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
        if err != nil {
            log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
            break
        } else {
            movies = append(movies, moviePage...)
        }
    }
}
return movies, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
```

```
// and Info is additional data.

type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [Scan](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Scans an Amazon DynamoDB table using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To scan items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, See the EnhancedScanRecords example.
 */

public class DynamoDBScanItems {
    public static void main(String[] args) {

        final String usage = """

            Usage:
            <tableName>

            Where:
            tableName - The Amazon DynamoDB table to get information from
            (for example, Music3).
            """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
```

```
Region region = Region.US_EAST_1;
DynamoDbClient ddb = DynamoDbClient.builder()
    .region(region)
    .build();

scanItems(ddb, tableName);
ddb.close();
}

public static void scanItems(DynamoDbClient ddb, String tableName) {
    try {
        ScanRequest scanRequest = ScanRequest.builder()
            .tableName(tableName)
            .build();

        ScanResponse response = ddb.scan(scanRequest);
        for (Map<String, AttributeValue> item : response.items()) {
            Set<String> keys = item.keySet();
            for (String key : keys) {
                System.out.println("The key name is " + key + "\n");
                System.out.println("The value is " + item.get(key).s());
            }
        }
    } catch (DynamoDbException e) {
        e.printStackTrace();
        System.exit(1);
    }
}
}
```

- For API details, see [Scan](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [ScanCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, ScanCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ScanCommand({
    ProjectionExpression: "#Name, Color, AvgLifeSpan",
    ExpressionAttributeNames: { "#Name": "Name" },
    TableName: "Birds",
  });

  const response = await docClient.send(command);
  for (const bird of response.Items) {
    console.log(`#${bird.Name} - (${bird.Color}, ${bird.AvgLifeSpan})`);
  }
  return response;
};
```

- For API details, see [Scan](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object.
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
const params = {
    // Specify which items in the results are returned.
    FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",
    // Define the expression attribute value, which are substitutes for the values
    // you want to compare.
    ExpressionAttributeValues: {
        ":topic": { S: "SubTitle2" },
        ":s": { N: 1 },
        ":e": { N: 2 },
    },
    // Set the projection expression, which are the attributes that you want.
    ProjectionExpression: "Season, Episode, Title, Subtitle",
    TableName: "EPISODES_TABLE",
};

ddb.scan(params, function (err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success", data);
        data.Items.forEach(function (element, index, array) {
            console.log(
                "printing",
                element.Title.S + " (" + element.Subtitle.S + ")"
            );
        });
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Scan](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun scanItems(tableNameVal: String) {  
    val request = ScanRequest {  
        tableName = tableNameVal  
    }  
  
    DynamoDbClient { region = "us-east-1" }.use { ddb ->  
        val response = ddb.scan(request)  
        response.items?.forEach { item ->  
            item.keys.forEach { key ->  
                println("The key name is $key\n")  
                println("The value is ${item[key]}")  
            }  
        }  
    }  
}
```

- For API details, see [Scan](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$yearsKey = [
```

```
'Key' => [
    'year' => [
        'N' => [
            'minRange' => 1990,
            'maxRange' => 1999,
        ],
    ],
],
];
$filter = "year between 1990 and 1999";
echo "\nHere's a list of all the movies released in the 90s:\n";
$result = $service->scan($tableName, $yearsKey, $filter);
foreach ($result['Items'] as $movie) {
    $movie = $marshal->unmarshalItem($movie);
    echo $movie['title'] . "\n";
}

public function scan(string $tableName, array $key, string $filters)
{
    $query = [
        'ExpressionAttributeNames' => ['#year' => 'year'],
        'ExpressionAttributeValues' => [
            ":min" => ['N' => '1990'],
            ":max" => ['N' => '1999'],
        ],
        'FilterExpression' => "#year between :min and :max",
        'TableName' => $tableName,
    ];
    return $this->dynamoDbClient->scan($query);
}
```

- For API details, see [Scan](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
  
    def scan_movies(self, year_range):  
        """  
        Scans for movies that were released in a range of years.  
        Uses a projection expression to return a subset of data for each movie.  
  
        :param year_range: The range of years to retrieve.  
        :return: The list of movies released in the specified years.  
        """  
        movies = []  
        scan_kwargs = {  
            "FilterExpression": Key("year").between(  
                year_range["first"], year_range["second"]  
            ),  
            "ProjectionExpression": "#yr, title, info.rating",  
            "ExpressionAttributeNames": {"#yr": "year"},  
        }  
        try:  
            done = False  
            start_key = None  
            while not done:  
                if start_key:  
                    scan_kwargs["ExclusiveStartKey"] = start_key  
                response = self.table.scan(**scan_kwargs)  
                movies.extend(response.get("Items", []))  
                start_key = response.get("LastEvaluatedKey", None)  
                done = start_key is None  
            except ClientError as err:  
                logger.error(  
                    "Couldn't scan for movies. Here's why: %s: %s",  
                    err.response["Error"]["Code"],
```

```
        err.response["Error"]["Message"],
    )
raise

return movies
```

- For API details, see [Scan](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
  attr_reader :dynamo_resource
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Scans for movies that were released in a range of years.
  # Uses a projection expression to return a subset of data for each movie.
  #
  # @param year_range [Hash] The range of years to retrieve.
  # @return [Array] The list of movies released in the specified years.
  def scan_items(year_range)
    movies = []
    scan_hash = {
      filter_expression: "#yr between :start_yr and :end_yr",
      projection_expression: "#yr, title, info.rating",
      expression_attribute_names: {"#yr" => "year"},
      expression_attribute_values: {
```

```
    ":start_yr" => year_range[:start], ":end_yr" => year_range[:end]})  
}  
done = false  
start_key = nil  
until done  
    scan_hash[:exclusive_start_key] = start_key unless start_key.nil?  
    response = @table.scan(scan_hash)  
    movies.concat(response.items) unless response.items.empty?  
    start_key = response.last_evaluated_key  
    done = start_key.nil?  
end  
rescue Aws::DynamoDB::Errors::ServiceError => e  
    puts("Couldn't scan for movies. Here's why:")  
    puts("\t#{e.code}: #{e.message}")  
    raise  
else  
    movies  
end
```

- For API details, see [Scan](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn list_items(client: &Client, table: &str, page_size: Option<i32>) ->  
Result<(), Error> {  
    let page_size = page_size.unwrap_or(10);  
    let items: Result<Vec<_>, _> = client  
        .scan()  
        .table_name(table)  
        .limit(page_size)  
        .into_paginator()  
        .items()  
        .send()
```

```
.collect()
.await;

println!("Items in table (up to {page_size}):");
for item in items? {
    println!("  {:?}", item);
}

Ok(())
}
```

- For API details, see [Scan](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
" Scan movies for rating greater than or equal to the rating specified
DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
    ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_rating }| ) ) .
DATA(lt_filter_conditions) = VALUE /aws1/
cl_dyncondition=>tt_filterconditionmap(
    ( VALUE /aws1/cl_dyncondition=>ts_filterconditionmap_maprow(
        key = 'rating'
        value = NEW /aws1/cl_dyncondition(
            it_attributevaluelist = lt_attributelist
            iv_comparisonoperator = |GE|
        ) ) ) .
oo_scan_result = lo_dyn->scan( iv_tablename = iv_table_name
    it_scanfilter = lt_filter_conditions ).
DATA(lt_items) = oo_scan_result->get_items( ).
LOOP AT lt_items INTO DATA(lo_item).
    " You can loop over to get individual attributes.
```

```
DATA(lo_title) = lo_item[ key = 'title' ]-value.  
DATA(lo_year) = lo_item[ key = 'year' ]-value.  
ENDLOOP.  
DATA(lv_count) = oo_scan_result->get_count( ).  
MESSAGE 'Found ' && lv_count && ' items' TYPE 'I'.  
CATCH /aws1/cx_dynresourcenotfoundex.  
MESSAGE 'The table or index does not exist' TYPE 'E'.  
ENDTRY.
```

- For API details, see [Scan](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Return an array of `Movie` objects released in the specified range of  
/// years.  
///  
/// - Parameters:  
///   - firstYear: The first year of movies to return.  
///   - lastYear: The last year of movies to return.  
///   - startKey: A starting point to resume processing; always use `nil`.  
///  
/// - Returns: An array of `Movie` objects describing the matching movies.  
///  
/// > Note: The `startKey` parameter is used by this function when  
///   recursively calling itself, and should always be `nil` when calling  
///   directly.
```

```
///  
func getMovies(firstYear: Int, lastYear: Int,  
               startKey: [Swift.String:DynamoDBClientTypes.AttributeValue]? =  
               nil)  
    async throws -> [Movie] {  
    var movieList: [Movie] = []  
  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = ScanInput(  
        consistentRead: true,  
        exclusiveStartKey: startKey,  
        expressionAttributeNames: [  
            "#y": "year"           // `year` is a reserved word, so use `#y`  
instead.  
        ],  
        expressionAttributeValues: [  
            ":y1": .n(String(firstYear)),  
            ":y2": .n(String(lastYear))  
        ],  
        filterExpression: "#y BETWEEN :y1 AND :y2",  
        tableName: self.tableName  
    )  
  
    let output = try await client.scan(input: input)  
  
    guard let items = output.items else {  
        return movieList  
    }  
  
    // Build an array of `Movie` objects for the returned items.  
  
    for item in items {  
        let movie = try Movie(withItem: item)  
        movieList.append(movie)  
    }  
  
    // Call this function recursively to continue collecting matching  
    // movies, if necessary.  
  
    if output.lastEvaluatedKey != nil {
```

```
        let movies = try await self.getMovies(firstYear: firstYear, lastYear:  
lastYear,  
                                         startKey: output.lastEvaluatedKey)  
        movieList += movies  
    }  
    return movieList  
}
```

- For API details, see [Scan](#) in [AWS SDK for Swift API reference](#).

For more DynamoDB examples, see [Code examples for DynamoDB using AWS SDKs](#).

Using DynamoDB with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples

SDK documentation	Code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

For examples specific to DynamoDB, see [Code examples for DynamoDB using AWS SDKs](#).

 **Example availability**

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Programming with DynamoDB and the AWS SDKs

This section covers developer-related topics. If you want to run code examples instead, see [Running the code examples in this Developer Guide](#).

Note

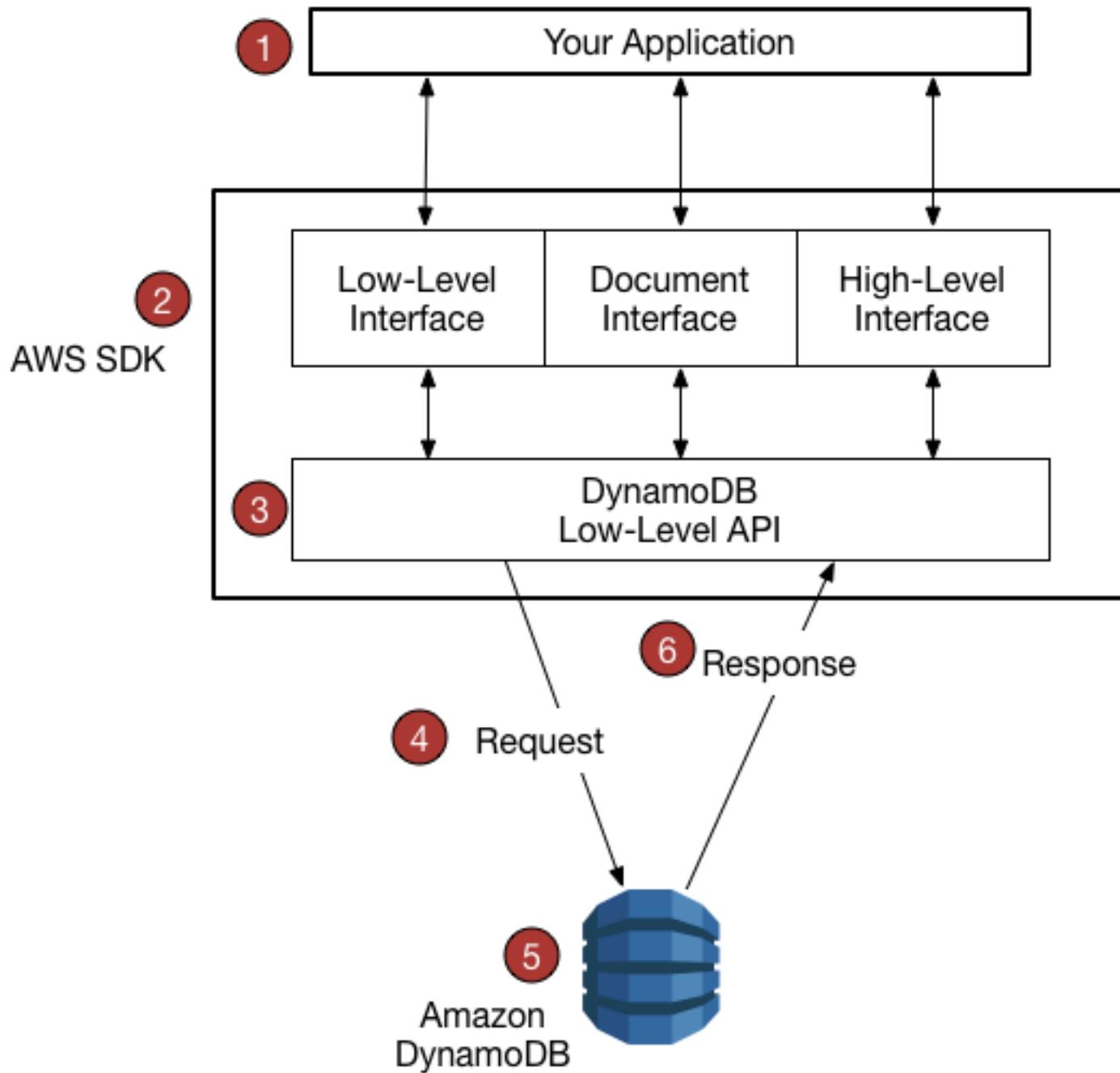
In December 2017, AWS began the process of migrating all Amazon DynamoDB endpoints to use secure certificates issued by Amazon Trust Services (ATS). For more information, see [Troubleshooting SSL/TLS connection establishment issues](#).

Topics

- [Overview of AWS SDK support for DynamoDB](#)
- [Higher-level programming interfaces for DynamoDB](#)
- [Running the code examples in this Developer Guide](#)
- [Programming Amazon DynamoDB with Python and Boto3](#)
- [Programming Amazon DynamoDB with JavaScript](#)
- [Programming Amazon DynamoDB with AWS SDK for Java 2.x](#)

Overview of AWS SDK support for DynamoDB

The following diagram provides a high-level overview of Amazon DynamoDB application programming using the AWS SDKs.



1. You write an application using an AWS SDK for your programming language.
2. Each AWS SDK provides one or more programmatic interfaces for working with DynamoDB. The specific interfaces available depend on which programming language and AWS SDK you use. Options include:
 - [Low-level interfaces](#)
 - [Document interfaces](#)
 - [Object persistence interface](#)

- [High Level Interfaces](#)

3. The AWS SDK constructs HTTP(S) requests for use with the low-level DynamoDB API.
4. The AWS SDK sends the request to the DynamoDB endpoint.
5. DynamoDB runs the request. If the request is successful, DynamoDB returns an HTTP 200 response code (OK). If the request is unsuccessful, DynamoDB returns an HTTP error code and an error message.
6. The AWS SDK processes the response and propagates it back to your application.

Each of the AWS SDKs provides important services to your application, including the following:

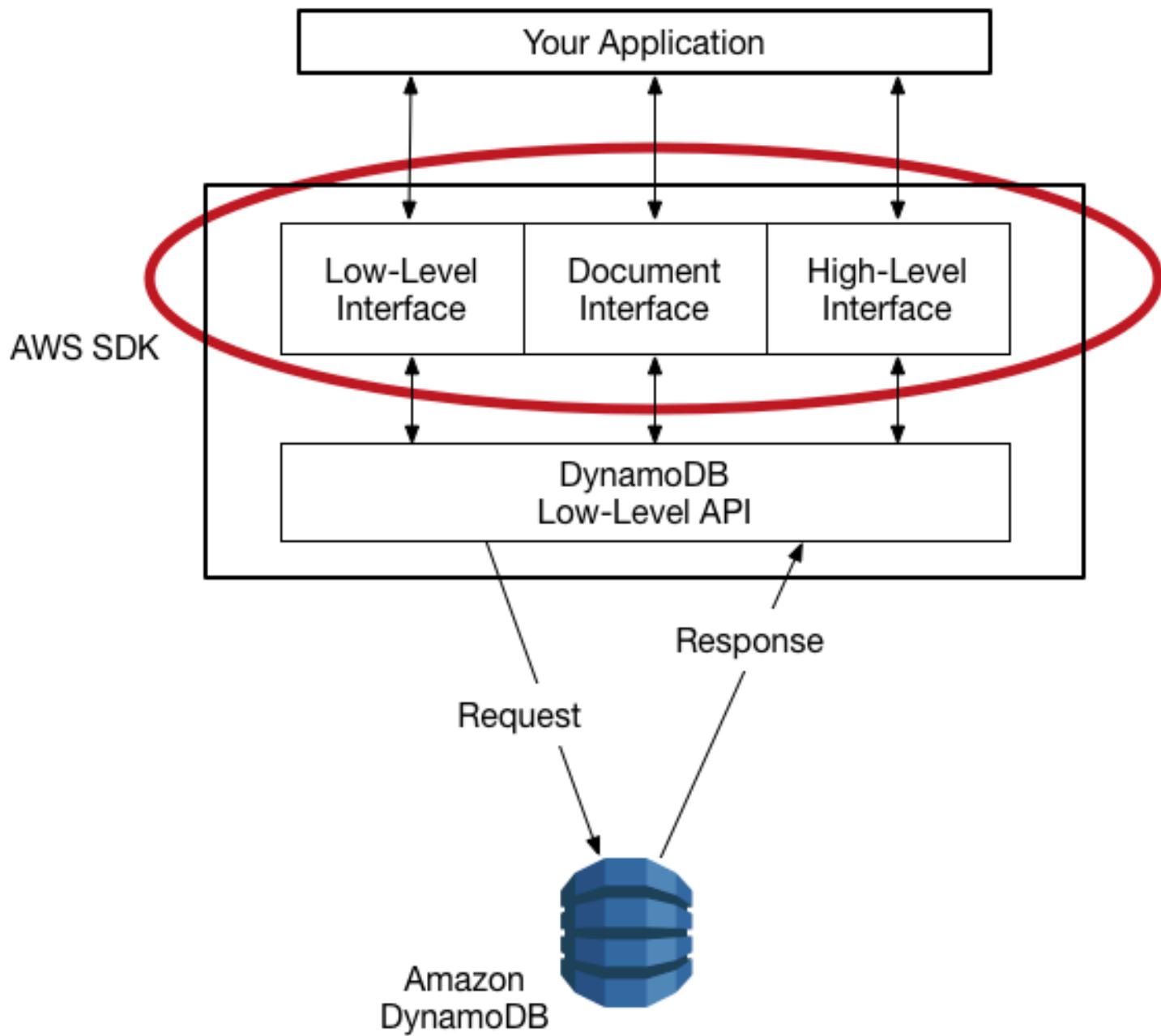
- Formatting HTTP(S) requests and serializing request parameters.
- Generating a cryptographic signature for each request.
- Forwarding requests to a DynamoDB endpoint and receiving responses from DynamoDB.
- Extracting the results from those responses.
- Implementing basic retry logic in case of errors.

You do not need to write code for any of these tasks.

 **Note**

For more information about AWS SDKs, including installation instructions and documentation, see [Tools for Amazon Web Services](#).

Programmatic interfaces



Every [AWS SDK](#) provides one or more programmatic interfaces for working with Amazon DynamoDB. These interfaces range from simple low-level DynamoDB wrappers to object-oriented persistence layers. The available interfaces vary depending on the AWS SDK and programming language that you use.

The following section highlights some of the interfaces available, using the AWS SDK for Java as an example. (Not all interfaces are available in all AWS SDKs.)

Topics

- [Low-level interfaces](#)
- [Document interfaces](#)
- [Object persistence interface](#)

Low-level interfaces

Every language-specific AWS SDK provides a low-level interface for Amazon DynamoDB, with methods that closely resemble low-level DynamoDB API requests.

In some cases, you will need to identify the data types of the attributes using [Data type descriptors](#), such as S for string or N for number.

 **Note**

A low-level interface is available in every language-specific AWS SDK.

The following Java program uses the low-level interface of the AWS SDK for Java.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
```

```
/*
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

            Usage:
            <tableName> <key> <keyVal>

            Where:
            tableName - The Amazon DynamoDB table from which an item is
            retrieved (for example, Music3).\s
            key - The key used in the Amazon DynamoDB table (for example,
            Artist).\s
            keyval - The key value that represents the item to get (for
            example, Famous Band).
            """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal, tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName, String
key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
            .s(keyVal)
            .build());

        GetItemRequest request = GetItemRequest.builder()
            .key(keyToGet)
```

```
.tableName(tableName)
.build();

try {
    // If there is no matching item, GetItem does not return any data.
    Map<String, AttributeValue> returnedItem = ddb.getItem(request).item();
    if (returnedItem.isEmpty())
        System.out.format("No item found with the key %s!\n", key);
    else {
        Set<String> keys = returnedItem.keySet();
        System.out.println("Amazon DynamoDB table attributes: \n");
        for (String key1 : keys) {
            System.out.format("%s: %s\n", key1,
returnedItem.get(key1).toString());
        }
    }
} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

Document interfaces

Many AWS SDKs provide a document interface, allowing you to perform data plane operations (create, read, update, delete) on tables and indexes. With a document interface, you do not need to specify [Data type descriptors](#). The data types are implied by the semantics of the data itself. These AWS SDKs also provide methods to easily convert JSON documents to and from native Amazon DynamoDB data types.

 **Note**

Document interfaces are available in the AWS SDKs for [Java](#), [.NET](#), [Node.js](#), and [JavaScript in the browser](#).

The following Java program uses the document interface of the AWS SDK for Java. The program creates a Table object that represents the Music table, and then asks that object to use GetItem to retrieve a song. The program then prints the year that the song was released.

The `com.amazonaws.services.dynamodbv2.document.DynamoDB` class implements the DynamoDB document interface. Note how DynamoDB acts as a wrapper around the low-level client (`AmazonDynamoDB`).

```
package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.GetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class MusicDocumentDemo {

    public static void main(String[] args) {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");
        GetItemOutcome outcome = table.getItemOutcome(
            "Artist", "No One You Know",
            "SongTitle", "Call Me Today");

        int year = outcome.getItem().getInt("Year");
        System.out.println("The song was released in " + year);

    }
}
```

Object persistence interface

Some AWS SDKs provide an object persistence interface where you do not directly perform data plane operations. Instead, you create objects that represent items in Amazon DynamoDB tables and indexes, and interact only with those objects. This allows you to write object-centric code, rather than database-centric code.

Note

Object persistence interfaces are available in the AWS SDKs for Java and .NET. For more information, see [Higher-level programming interfaces for DynamoDB](#) for DynamoDB.

```
import com.example.dynamodb.Customer;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.enhanced.dynamodb.model.GetItemEnhancedRequest;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
```

```
import com.example.dynamodb.Customer;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.enhanced.dynamodb.model.GetItemEnhancedRequest;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;

/*
 * Before running this code example, create an Amazon DynamoDB table named Customer
 * with these columns:
 *   - id - the id of the record that is the key. Be sure one of the id values is
 `id101`
 *   - custName - the customer name
 *   - email - the email value
 *   - registrationDate - an instant value when the item was added to the table. These
 values
 *           need to be in the form of `YYYY-MM-DDTHH:mm:ssZ`, such as
2022-07-11T00:00:00Z
 *
 * Also, ensure that you have set up your development environment, including your
credentials.
*
* For information, see this documentation topic:
*
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
*/
public class EnhancedGetItem {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
```

```
DynamoDbClient ddb = DynamoDbClient.builder()
    .region(region)
    .build();

DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
    .dynamoDbClient(ddb)
    .build();

getItem(enhancedClient);
ddb.close();
}

public static String getItem(DynamoDbEnhancedClient enhancedClient) {
    Customer result = null;
    try {
        DynamoDbTable<Customer> table = enhancedClient.table("Customer",
TableSchema.fromBean(Customer.class));
        Key key = Key.builder()
            .partitionValue("id101").sortValue("tred@noserver.com")
            .build();

        // Get the item by using the key.
        result = table.getItem(
            (GetItemEnhancedRequest.Builder requestBuilder) ->
requestBuilder.key(key));
        System.out.println("***** The description value is " +
result.getCustName());

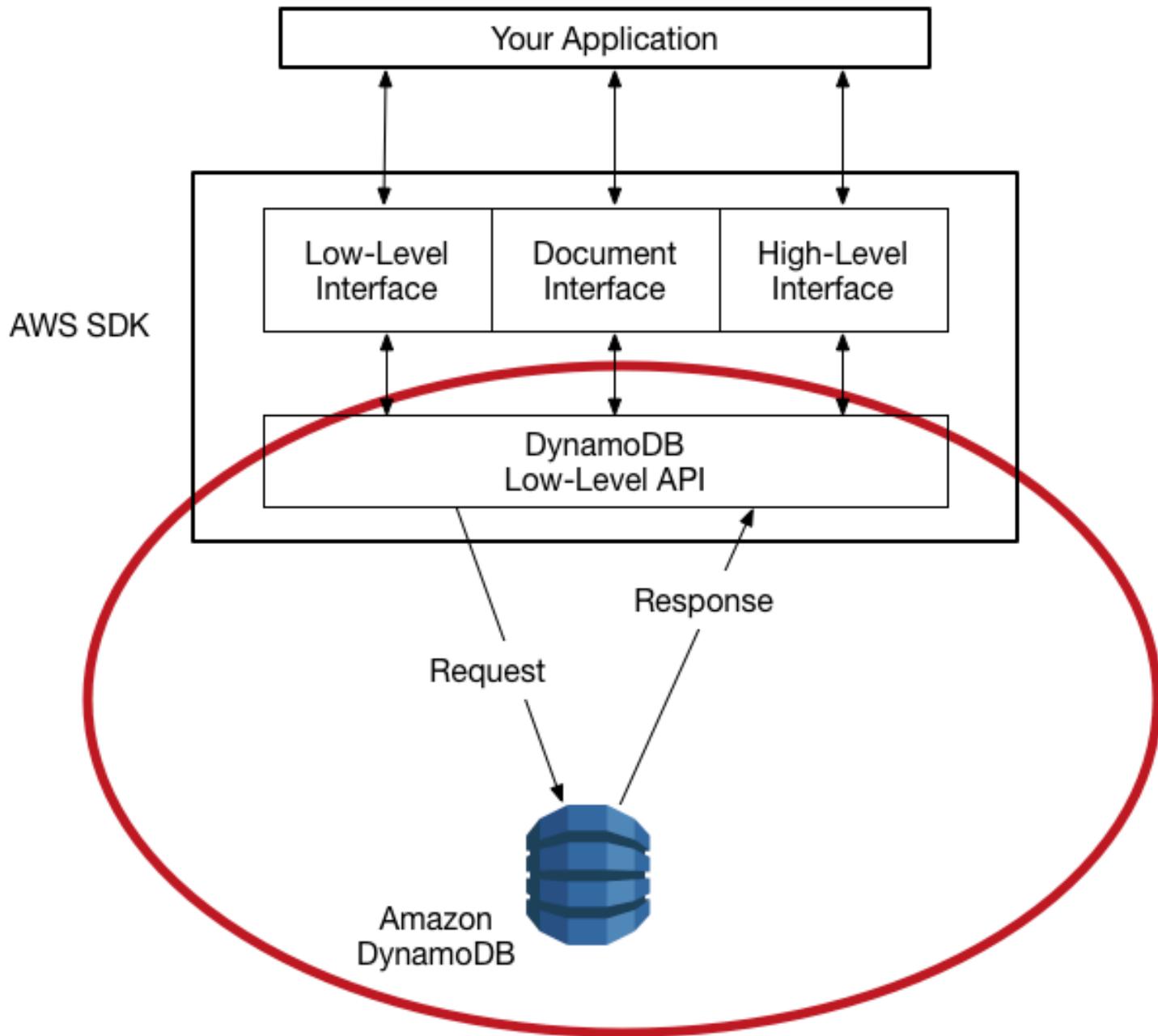
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    return result.getCustName();
}
}
```

DynamoDB low-level API

Topics

- [Request format](#)
- [Response format](#)
- [Data type descriptors](#)

- [Numeric data](#)
- [Binary data](#)



The Amazon DynamoDB *low-level API* is the protocol-level interface for DynamoDB. At this level, every HTTP(S) request must be correctly formatted and carry a valid digital signature.

The AWS SDKs construct low-level DynamoDB API requests on your behalf and process the responses from DynamoDB. This lets you focus on your application logic, instead of low-level

details. However, you can still benefit from a basic knowledge of how the low-level DynamoDB API works.

For more information about the low-level DynamoDB API, see [Amazon DynamoDB API Reference](#).

Note

DynamoDB Streams has its own low-level API, which is separate from that of DynamoDB and is fully supported by the AWS SDKs.

For more information, see [Change data capture for DynamoDB Streams](#). For the low-level DynamoDB Streams API, see the [Amazon DynamoDB Streams API Reference](#).

The low-level DynamoDB API uses JavaScript Object Notation (JSON) as a wire protocol format. JSON presents data in a hierarchy so that both data values and data structure are conveyed simultaneously. Name-value pairs are defined in the format name : value. The data hierarchy is defined by nested brackets of name-value pairs.

DynamoDB uses JSON only as a transport protocol, not as a storage format. The AWS SDKs use JSON to send data to DynamoDB, and DynamoDB responds with JSON. DynamoDB does not store data persistently in JSON format.

Note

For more information about JSON, see [Introducing JSON](#) on the [JSON.org](#) website.

Request format

The DynamoDB low-level API accepts HTTP(S) POST requests as input. The AWS SDKs construct these requests for you.

Suppose that you have a table named Pets, with a key schema consisting of AnimalType (partition key) and Name (sort key). Both of these attributes are of type string. To retrieve an item from Pets, the AWS SDK constructs the following request.

```
POST / HTTP/1.1
Host: dynamodb.<region>.amazonaws.com;
Accept-Encoding: identity
```

```
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.0
Authorization: AWS4-HMAC-SHA256 Credential=<Credential>, SignedHeaders=<Headers>,
Signature=<Signature>
X-Amz-Date: <Date>
X-Amz-Target: DynamoDB_20120810.GetItem

{
    "TableName": "Pets",
    "Key": {
        "AnimalType": {"S": "Dog"},
        "Name": {"S": "Fido"}
    }
}
```

Note the following about this request:

- The Authorization header contains information required for DynamoDB to authenticate the request. For more information, see [Signing AWS API requests](#) and [Signature Version 4 signing process](#) in the *Amazon Web Services General Reference*.
- The X-Amz-Target header contains the name of a DynamoDB operation: GetItem. (This is also accompanied by the low-level API version, in this case 20120810.)
- The payload (body) of the request contains the parameters for the operation, in JSON format. For the GetItem operation, the parameters are TableName and Key.

Response format

Upon receipt of the request, DynamoDB processes it and returns a response. For the request shown previously, the HTTP(S) response payload contains the results from the operation, as shown in the following example.

```
HTTP/1.1 200 OK
x-amzn-RequestId: <RequestId>
x-amz-crc32: <Checksum>
Content-Type: application/x-amz-json-1.0
Content-Length: <PayloadSizeBytes>
Date: <Date>
{
    "Item": {
```

```
"Age": {"N": "8"},  
"Colors": {  
    "L": [  
        {"S": "White"},  
        {"S": "Brown"},  
        {"S": "Black"}  
    ]  
},  
"Name": {"S": "Fido"},  
"Vaccinations": {  
    "M": {  
        "Rabies": {  
            "L": [  
                {"S": "2009-03-17"},  
                {"S": "2011-09-21"},  
                {"S": "2014-07-08"}  
            ]  
        },  
        "Distemper": {"S": "2015-10-13"}  
    }  
},  
"Breed": {"S": "Beagle"},  
"AnimalType": {"S": "Dog"}  
}  
}
```

At this point, the AWS SDK returns the response data to your application for further processing.

 **Note**

If DynamoDB can't process a request, it returns an HTTP error code and message. The AWS SDK propagates these to your application in the form of exceptions. For more information, see [Error handling with DynamoDB](#).

Data type descriptors

The low-level DynamoDB API protocol requires each attribute to be accompanied by a data type descriptor. *Data type descriptors* are tokens that tell DynamoDB how to interpret each attribute.

The examples in [Request format](#) and [Response format](#) show examples of how data type descriptors are used. The GetItem request specifies S for the Pets key schema attributes (AnimalType and

Name), which are of type **string**. The GetItem response contains a *Pets* item with attributes of type **string** (S), **number** (N), **map** (M), and **list** (L).

The following is a complete list of DynamoDB data type descriptors:

- **S** – String
- **N** – Number
- **B** – Binary
- **BOOL** – Boolean
- **NULL** – Null
- **M** – Map
- **L** – List
- **SS** – String Set
- **NS** – Number Set
- **BS** – Binary Set

 **Note**

For detailed descriptions of DynamoDB data types, see [Data types](#).

Numeric data

Different programming languages offer different levels of support for JSON. In some cases, you might decide to use a third-party library for validating and parsing JSON documents.

Some third-party libraries build upon the JSON number type, providing their own types such as `int`, `long`, or `double`. However, the native number data type in DynamoDB does not map exactly to these other data types, so these type distinctions can cause conflicts. In addition, many JSON libraries do not handle fixed-precision numeric values, and they automatically infer a double data type for digit sequences that contain a decimal point.

To solve these problems, DynamoDB provides a single numeric type with no data loss. To avoid unwanted implicit conversions to a double value, DynamoDB uses strings for the data transfer of numeric values. This approach provides flexibility for updating attribute values while maintaining proper sorting semantics, such as putting the values "01", "2", and "03" in the proper sequence.

If number precision is important to your application, you should convert numeric values to strings before you pass them to DynamoDB.

Binary data

DynamoDB supports binary attributes. However, JSON does not natively support encoding binary data. To send binary data in a request, you will need to encode it in base64 format. Upon receiving the request, DynamoDB decodes the base64 data back to binary.

The base64 encoding scheme used by DynamoDB is described at [RFC 4648](#) on the Internet Engineering Task Force (IETF) website.

Error handling with DynamoDB

This section describes runtime errors and how to handle them. It also describes error messages and codes that are specific to Amazon DynamoDB. For a list of common errors that apply to all AWS services, see [Access Management](#)

Topics

- [Error components](#)
- [Transactional errors](#)
- [Error messages and codes](#)
- [Error handling in your application](#)
- [Error retries and exponential backoff](#)
- [Batch operations and error handling](#)

Error components

When your program sends a request, DynamoDB attempts to process it. If the request is successful, DynamoDB returns an HTTP success status code (200 OK), along with the results from the requested operation.

If the request is unsuccessful, DynamoDB returns an error. Each error has three components:

- An HTTP status code (such as 400).
- An exception name (such as `ResourceNotFoundException`).
- An error message (such as `Requested resource not found: Table: tablename not found`).

The AWS SDKs take care of propagating errors to your application so that you can take appropriate action. For example, in a Java program, you can write try-catch logic to handle a `ResourceNotFoundException`.

If you are not using an AWS SDK, you need to parse the content of the low-level response from DynamoDB. The following is an example of such a response.

```
HTTP/1.1 400 Bad Request
x-amzn-RequestId: LDM6CJP8RMQ1FHKSC1RBVJFPNVV4KQNS05AEMF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 240
Date: Thu, 15 Mar 2012 23:56:23 GMT

{"__type":"com.amazonaws.dynamodb.v20120810#ResourceNotFoundException",
"message":"Requested resource not found: Table: tablename not found"}
```

Transactional errors

For information on transactional errors, please see [Transaction conflict handling in DynamoDB](#)

Error messages and codes

The following is a list of exceptions returned by DynamoDB, grouped by HTTP status code. If *OK to retry?* is *Yes*, you can submit the same request again. If *OK to retry?* is *No*, you need to fix the problem on the client side before you submit a new request.

HTTP status code 400

An HTTP 400 status code indicates a problem with your request, such as authentication failure, missing required parameters, or exceeding a table's provisioned throughput. You have to fix the issue in your application before submitting the request again.

AccessDeniedException

Message: *Access denied.*

The client did not correctly sign the request. If you are using an AWS SDK, requests are signed for you automatically; otherwise, go to the [Signature version 4 signing process](#) in the AWS *General Reference*.

OK to retry? No

ConditionalCheckFailedException

Message: *The conditional request failed.*

You specified a condition that evaluated to false. For example, you might have tried to perform a conditional update on an item, but the actual value of the attribute did not match the expected value in the condition.

OK to retry? No

IncompleteSignatureException

Message: *The request signature does not conform to AWS standards.*

The request signature did not include all of the required components. If you are using an AWS SDK, requests are signed for you automatically; otherwise, go to the [Signature Version 4 signing process](#) in the *AWS General Reference*.

OK to retry? No

ItemCollectionSizeLimitExceededException

Message: *Collection size exceeded.*

For a table with a local secondary index, a group of items with the same partition key value has exceeded the maximum size limit of 10 GB. For more information on item collections, see [Item collections in Local Secondary Indexes](#).

OK to retry? Yes

LimitExceededException

Message: *Too many operations for a given subscriber.*

There are too many concurrent control plane operations. The cumulative number of tables and indexes in the CREATING, DELETING, or UPDATING state cannot exceed 500.

OK to retry? Yes

MissingAuthenticationTokenException

Message: Request must contain a valid (registered) AWS Access Key ID.

The request did not include the required authorization header, or it was malformed. See [DynamoDB low-level API](#).

OK to retry? No

ProvisionedThroughputExceededException

Message: You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. To view performance metrics for provisioned throughput vs. consumed throughput, open the [Amazon CloudWatch console](#).

Example: Your request rate is too high. The AWS SDKs for DynamoDB automatically retry requests that receive this exception. Your request is eventually successful, unless your retry queue is too large to finish. Reduce the frequency of requests using [Error retries and exponential backoff](#).

OK to retry? Yes

RequestLimitExceeded

Message: Throughput exceeds the current throughput limit for your account. To request a limit increase, contact AWS Support at <https://aws.amazon.com/support>.

Example: Rate of on-demand requests exceeds the allowed account throughput and the table cannot be scaled further.

OK to retry? Yes

ResourceInUseException

Message: The resource which you are attempting to change is in use.

Example: You tried to re-create an existing table, or delete a table currently in the CREATING state.

OK to retry? No

ResourceNotFoundException

Message: *Requested resource not found.*

Example: The table that is being requested does not exist, or is too early in the CREATING state.

OK to retry? No

ThrottlingException

Message: *Rate of requests exceeds the allowed throughput.*

This exception is returned as an AmazonServiceException response with a THROTTLING_EXCEPTION status code. This exception might be returned if you perform [control plane](#) API operations too rapidly.

For tables using on-demand mode, this exception might be returned for any [data plane](#) API operation if your request rate is too high. To learn more about on-demand scaling, see [Peak traffic and scaling properties](#)

OK to retry? Yes

UnrecognizedClientException

Message: *The Access Key ID or security token is invalid.*

The request signature is incorrect. The most likely cause is an invalid AWS access key ID or secret key.

OK to retry? Yes

ValidationException

Message: Varies, depending upon the specific error(s) encountered

This error can occur for several reasons, such as a required parameter that is missing, a value that is out of range, or mismatched data types. The error message contains details about the specific part of the request that caused the error.

OK to retry? No

HTTP status code 5xx

An HTTP 5xx status code indicates a problem that must be resolved by AWS. This might be a transient error, in which case you can retry your request until it succeeds. Otherwise, go to the [AWS Service Health Dashboard](#) to see if there are any operational issues with the service.

For more information, see [How do I resolve HTTP 5xx errors in Amazon DynamoDB?](#)

InternalServerError (HTTP 500)

DynamoDB could not process your request.

OK to retry? Yes

Note

You might encounter internal server errors while working with items. These are expected during the lifetime of a table. Any failed requests can be retried immediately. When you receive a status code 500 on a write operation, the operation may have succeeded or failed. If the write operation is a `TransactWriteItem` request, then it is OK to retry the operation. If the write operation is a single-item write request such as `PutItem`, `UpdateItem`, or `DeleteItem`, then your application should read the state of the item before retrying the operation, and/or use [Condition expressions](#) to ensure that the item remains in a correct state after retrying regardless of whether the prior operation succeeded or failed. If idempotency is a requirement for the write operation, please use [TransactWriteItem](#), which supports idempotent requests by automatically specifying a `ClientRequestToken` to disambiguate multiple attempts to perform the same action.

ServiceUnavailable (HTTP 503)

DynamoDB is currently unavailable. (This should be a temporary state.)

OK to retry? Yes

Error handling in your application

For your application to run smoothly, you need to add logic to catch and respond to errors. Typical approaches include using try-catch blocks or if-then statements.

The AWS SDKs perform their own retries and error checking. If you encounter an error while using one of the AWS SDKs, the error code and description can help you troubleshoot it.

You should also see a Request ID in the response. The Request ID can be helpful if you need to work with AWS Support to diagnose an issue.

Error retries and exponential backoff

Numerous components on a network, such as DNS servers, switches, load balancers, and others, can generate errors anywhere in the life of a given request. The usual technique for dealing with these error responses in a networked environment is to implement retries in the client application. This technique increases the reliability of the application.

Each AWS SDK implements retry logic automatically. You can modify the retry parameters to your needs. For example, consider a Java application that requires a fail-fast strategy, with no retries allowed in case of an error. With the AWS SDK for Java, you could use the `ClientConfiguration` class and provide a `maxErrorRetry` value of `0` to turn off the retries. For more information, see the AWS SDK documentation for your programming language.

If you're not using an AWS SDK, you should retry original requests that receive server errors (5xx). However, client errors (4xx, other than a `ThrottlingException` or a `ProvisionedThroughputExceededException`) indicate that you need to revise the request itself to correct the problem before trying again.

In addition to simple retries, each AWS SDK implements an exponential backoff algorithm for better flow control. The concept behind exponential backoff is to use progressively longer waits between retries for consecutive error responses. For example, up to 50 milliseconds before the first retry, up to 100 milliseconds before the second, up to 200 milliseconds before third, and so on. However, after a minute, if the request has not succeeded, the problem might be the request size exceeding your provisioned throughput, and not the request rate. Set the maximum number of retries to stop around one minute. If the request is not successful, investigate your provisioned throughput options.

Note

The AWS SDKs implement automatic retry logic and exponential backoff.

Most exponential backoff algorithms use jitter (randomized delay) to prevent successive collisions. Because you aren't trying to avoid such collisions in these cases, you do not need to use this

random number. However, if you use concurrent clients, jitter can help your requests succeed faster. For more information, see the blog post about [Exponential backoff and jitter](#).

Batch operations and error handling

The DynamoDB low-level API supports batch operations for reads and writes. `BatchGetItem` reads items from one or more tables, and `BatchWriteItem` puts or deletes items in one or more tables. These batch operations are implemented as wrappers around other non-batch DynamoDB operations. In other words, `BatchGetItem` invokes `GetItem` once for each item in the batch. Similarly, `BatchWriteItem` invokes `DeleteItem` or `PutItem`, as appropriate, for each item in the batch.

A batch operation can tolerate the failure of individual requests in the batch. For example, consider a `BatchGetItem` request to read five items. Even if some of the underlying `GetItem` requests fail, this does not cause the entire `BatchGetItem` operation to fail. However, if all five read operations fail, then the entire `BatchGetItem` fails.

The batch operations return information about individual requests that fail so that you can diagnose the problem and retry the operation. For `BatchGetItem`, the tables and primary keys in question are returned in the `UnprocessedKeys` value of the response. For `BatchWriteItem`, similar information is returned in `UnprocessedItems`.

The most likely cause of a failed read or a failed write is *throttling*. For `BatchGetItem`, one or more of the tables in the batch request does not have enough provisioned read capacity to support the operation. For `BatchWriteItem`, one or more of the tables does not have enough provisioned write capacity.

If DynamoDB returns any unprocessed items, you should retry the batch operation on those items. However, *we strongly recommend that you use an exponential backoff algorithm*. If you retry the batch operation immediately, the underlying read or write requests can still fail due to throttling on the individual tables. If you delay the batch operation using exponential backoff, the individual requests in the batch are much more likely to succeed.

Higher-level programming interfaces for DynamoDB

The AWS SDKs provide applications with low-level interfaces for working with Amazon DynamoDB. These client-side classes and methods correspond directly to the low-level DynamoDB API. However, many developers experience a sense of disconnect, or *impedance mismatch*, when they need to map complex data types to items in a database table. With a low-level database interface,

developers must write methods for reading or writing object data to database tables, and vice versa. The amount of extra code required for each combination of object type and database table can seem overwhelming.

To simplify development, the AWS SDKs for Java and .NET provide additional interfaces with higher levels of abstraction. The higher-level interfaces for DynamoDB let you define the relationships between objects in your program and the database tables that store those objects' data. After you define this mapping, you call simple object methods such as save, load, or delete, and the underlying low-level DynamoDB operations are automatically invoked on your behalf. This allows you to write object-centric code, rather than database-centric code.

The higher-level programming interfaces for DynamoDB are available in the AWS SDKs for Java and .NET.

Java

- [Java 1.x: DynamoDBMapper](#)
- [Java 2.x: DynamoDB Enhanced Client](#)

.NET

- [.NET: Document model](#)
- [.NET: Object persistence model](#)

Java 1.x: DynamoDBMapper

The AWS SDK for Java provides a `DynamoDBMapper` class, allowing you to map your client-side classes to Amazon DynamoDB tables. To use `DynamoDBMapper`, you define the relationship between items in a DynamoDB table and their corresponding object instances in your code. The `DynamoDBMapper` class enables you to perform various create, read, update, and delete (CRUD) operations on items, and run queries and scans against tables.

Topics

- [Supported data types for DynamoDB Mapper for Java](#)
- [Java Annotations for DynamoDB](#)
- [DynamoDBMapper Class](#)
- [Optional configuration settings for DynamoDBMapper](#)

- [Optimistic locking with version number](#)
- [Mapping arbitrary data](#)
- [DynamoDBMapper Examples](#)

Note

The `DynamoDBMapper` class does not allow you to create, update, or delete tables.

To perform those tasks, use the low-level SDK for Java interface instead. For more information, see [Working with DynamoDB tables in Java](#).

The SDK for Java provides a set of annotation types so that you can map your classes to tables. For example, consider a `ProductCatalog` table that has `Id` as the partition key.

```
ProductCatalog(Id, ...)
```

You can map a class in your client application to the `ProductCatalog` table as shown in the following Java code. This code defines a plain old Java object (POJO) named `CatalogItem`, which uses annotations to map object fields to DynamoDB attribute names.

Example

```
package com.amazonaws.codesamples;

import java.util.Set;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIgnore;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;
```

```
@DynamoDBHashKey(attributeName="Id")
public Integer getId() { return id; }
public void setId(Integer id) {this.id = id; }

@dynamoDBAttribute(attributeName="Title")
public String getTitle() {return title; }
public void setTitle(String title) { this.title = title; }

@dynamoDBAttribute(attributeName="ISBN")
public String getISBN() { return ISBN; }
public void setISBN(String ISBN) { this.ISBN = ISBN; }

@dynamoDBAttribute(attributeName="Authors")
public Set<String> getBookAuthors() { return bookAuthors; }
public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors =
bookAuthors; }

@DynamoDBIgnore
public String getSomeProp() { return someProp; }
public void setSomeProp(String someProp) { this.someProp = someProp; }
}
```

In the preceding code, the `@DynamoDBTable` annotation maps the `CatalogItem` class to the `ProductCatalog` table. You can store individual class instances as items in the table. In the class definition, the `@DynamoDBHashKey` annotation maps the `Id` property to the primary key.

By default, the class properties map to the same name attributes in the table. The properties `Title` and `ISBN` map to the same name attributes in the table.

The `@DynamoDBAttribute` annotation is optional when the name of the DynamoDB attribute matches the name of the property declared in the class. When they differ, use this annotation with the `attributeName` parameter to specify which DynamoDB attribute this property corresponds to.

In the preceding example, the `@DynamoDBAttribute` annotation is added to each property to ensure that the property names match exactly with the tables created in [Creating tables and loading data for code examples in DynamoDB](#), and to be consistent with the attribute names used in other code examples in this guide.

Your class definition can have properties that don't map to any attributes in the table. You identify these properties by adding the `@DynamoDBIgnore` annotation. In the preceding example, the

The SomeProp property is marked with the @DynamoDBIgnore annotation. When you upload a CatalogItem instance to the table, your DynamoDBMapper instance does not include the SomeProp property. In addition, the mapper does not return this attribute when you retrieve an item from the table.

After you define your mapping class, you can use DynamoDBMapper methods to write an instance of that class to a corresponding item in the Catalog table. The following code example demonstrates this technique.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapper mapper = new DynamoDBMapper(client);

CatalogItem item = new CatalogItem();
item.setId(102);
item.setTitle("Book 102 Title");
item.setISBN("222-222222222");
item.setBookAuthors(new HashSet<String>(Arrays.asList("Author 1", "Author 2")));
item.setSomeProp("Test");

mapper.save(item);
```

The following code example shows how to retrieve the item and access some of its attributes.

```
CatalogItem partitionKey = new CatalogItem();

partitionKey.setId(102);
DynamoDBQueryExpression<CatalogItem> queryExpression = new
DynamoDBQueryExpression<CatalogItem>()
.withHashKeyValues(partitionKey);

List<CatalogItem> itemList = mapper.query(CatalogItem.class, queryExpression);

for (int i = 0; i < itemList.size(); i++) {
    System.out.println(itemList.get(i).getTitle());
    System.out.println(itemList.get(i).getBookAuthors());
}
```

DynamoDBMapper offers an intuitive, natural way of working with DynamoDB data within Java. It also provides several built-in features, such as optimistic locking, ACID transactions, autogenerated partition key and sort key values, and object versioning.

Supported data types for DynamoDB Mapper for Java

This section describes the supported primitive Java data types, collections, and arbitrary data types in Amazon DynamoDB.

Amazon DynamoDB supports the following primitive Java data types and primitive wrapper classes.

- `String`
- `Boolean`, `boolean`
- `Byte`, `byte`
- `Date` (as [ISO_8601](#) millisecond-precision string, shifted to UTC)
- `Calendar` (as [ISO_8601](#) millisecond-precision string, shifted to UTC)
- `Long`, `long`
- `Integer`, `int`
- `Double`, `double`
- `Float`, `float`
- `BigDecimal`
- `BigInteger`

Note

- For more information about DynamoDB naming rules and the various supported data types, see [Supported data types and naming rules in Amazon DynamoDB](#).
- Empty Binary values are supported by the DynamoDBMapper.
- Empty String values are supported by AWS SDK for Java 2.x.

In AWS SDK for Java 1.x, DynamoDBMapper supports reading of empty String attribute values, however, it will not write empty String attribute values because these attributes are dropped from the request.

DynamoDB supports the Java [Set](#), [List](#), and [Map](#) collection types. The following table summarizes how these Java types map to the DynamoDB types.

Java type	DynamoDB type
All number types	N (number type)
Strings	S (string type)
Boolean	BOOL (Boolean type), 0 or 1.
ByteBuffer	B (binary type)
Date	S (string type). The Date values are stored as ISO-8601 formatted strings.
Set collection types	SS (string set) type, NS (number set) type, or BS (binary set) type.

The `DynamoDBTypeConverter` interface lets you map your own arbitrary data types to a data type that is natively supported by DynamoDB. For more information, see [Mapping arbitrary data](#).

Java Annotations for DynamoDB

This section describes the annotations that are available for mapping your classes and properties to tables and attributes in Amazon DynamoDB.

For the corresponding Javadoc documentation, see [Annotation Types Summary](#) in the [AWS SDK for Java API Reference](#).

 **Note**

In the following annotations, only `DynamoDBTable` and the `DynamoDBHashKey` are required.

Topics

- [DynamoDBAttribute](#)
- [DynamoDBAutoGeneratedKey](#)
- [DynamoDBAutoGeneratedTimestamp](#)

- [DynamoDBDocument](#)
- [DynamoDBHashKey](#)
- [DynamoDBIgnore](#)
- [DynamoDBIndexHashKey](#)
- [DynamoDBIndexRangeKey](#)
- [DynamoDBRangeKey](#)
- [DynamoDBTable](#)
- [DynamoDBTypeConverted](#)
- [DynamoDBTyped](#)
- [DynamoDBVersionAttribute](#)

DynamoDBAttribute

Maps a property to a table attribute. By default, each class property maps to an item attribute with the same name. However, if the names are not the same, you can use this annotation to map a property to the attribute. In the following Java snippet, the `DynamoDBAttribute` maps the `BookAuthors` property to the `Authors` attribute name in the table.

```
@DynamoDBAttribute(attributeName = "Authors")
public List<String> getBookAuthors() { return BookAuthors; }
public void setBookAuthors(List<String> BookAuthors) { this.BookAuthors =
    BookAuthors; }
```

The `DynamoDBMapper` uses `Authors` as the attribute name when saving the object to the table.

DynamoDBAutoGeneratedKey

Marks a partition key or sort key property as being autogenerated. `DynamoDBMapper` generates a random [UUID](#) when saving these attributes. Only String properties can be marked as autogenerated keys.

The following example demonstrates using autogenerated keys.

```
@DynamoDBTable(tableName="AutoGeneratedKeysExample")
public class AutoGeneratedKeys {
    private String id;
```

```
private String payload;

@DynamoDBHashKey(attributeName = "Id")
@DynamoDBAutoGeneratedKey
public String getId() { return id; }
public void setId(String id) { this.id = id; }

@DynamoDBAttribute(attributeName="payload")
public String getPayload() { return this.payload; }
public void setPayload(String payload) { this.payload = payload; }

public static void saveItem() {
    AutoGeneratedKeys obj = new AutoGeneratedKeys();
    obj.setPayload("abc123");

    // id field is null at this point
    DynamoDBMapper mapper = new DynamoDBMapper(dynamoDBClient);
    mapper.save(obj);

    System.out.println("Object was saved with id " + obj.getId());
}
}
```

DynamoDBAutoGeneratedTimestamp

Automatically generates a timestamp.

```
@DynamoDBAutoGeneratedTimestamp(strategy=DynamoDBAutoGenerateStrategy.ALWAYS)
public Date getLastUpdatedDate() { return lastUpdatedDate; }
public void setLastUpdatedDate(Date lastUpdatedDate) { this.lastUpdatedDate =
lastUpdatedDate; }
```

Optionally, the auto-generation strategy can be defined by providing a strategy attribute. The default is ALWAYS.

DynamoDBDocument

Indicates that a class can be serialized as an Amazon DynamoDB document.

For example, suppose that you wanted to map a JSON document to a DynamoDB attribute of type Map (M). The following code example defines an item containing a nested attribute (Pictures) of type Map.

```
public class ProductCatalogItem {  
  
    private Integer id; //partition key  
    private Pictures pictures;  
    /* ...other attributes omitted... */  
  
    @DynamoDBHashKey(attributeName="Id")  
    public Integer getId() { return id; }  
    public void setId(Integer id) {this.id = id;}  
  
    @DynamoDBAttribute(attributeName="Pictures")  
    public Pictures getPictures() { return pictures; }  
    public void setPictures(Pictures pictures) {this.pictures = pictures;}  
  
    // Additional properties go here.  
  
    @DynamoDBDocument  
    public static class Pictures {  
        private String frontView;  
        private String rearView;  
        private String sideView;  
  
        @DynamoDBAttribute(attributeName = "FrontView")  
        public String getFrontView() { return frontView; }  
        public void setFrontView(String frontView) { this.frontView = frontView; }  
  
        @DynamoDBAttribute(attributeName = "RearView")  
        public String getRearView() { return rearView; }  
        public void setRearView(String rearView) { this.rearView = rearView; }  
  
        @DynamoDBAttribute(attributeName = "SideView")  
        public String getSideView() { return sideView; }  
        public void setSideView(String sideView) { this.sideView = sideView; }  
    }  
}
```

You could then save a new `ProductCatalog` item, with `Pictures`, as shown in the following example.

```
ProductCatalogItem item = new ProductCatalogItem();  
  
Pictures pix = new Pictures();
```

```
pix.setFrontView("http://example.com/products/123_front.jpg");
pix.setRearView("http://example.com/products/123_rear.jpg");
pix.setSideView("http://example.com/products/123_left_side.jpg");
item.setPictures(pic);

item.setId(123);

mapper.save(item);
```

The resulting ProductCatalog item would look like the following (in JSON format).

```
{
    "Id" : 123
    "Pictures" : {
        "SideView" : "http://example.com/products/123_left_side.jpg",
        "RearView" : "http://example.com/products/123_rear.jpg",
        "FrontView" : "http://example.com/products/123_front.jpg"
    }
}
```

DynamoDBHashKey

Maps a class property to the partition key of the table. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Assume that you have a table, ProductCatalog, that has Id as the primary key. The following Java code defines a CatalogItem class and maps its Id property to the primary key of the ProductCatalog table using the @DynamoDBHashKey tag.

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {
    private Integer Id;
    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() {
        return Id;
    }
    public void setId(Integer Id) {
        this.Id = Id;
    }
    // Additional properties go here.
}
```

DynamoDBIgnore

Indicates to the `DynamoDBMapper` instance that the associated property should be ignored. When saving data to the table, the `DynamoDBMapper` does not save this property to the table.

Applied to the getter method or the class field for a non-modeled property. If the annotation is applied directly to the class field, the corresponding getter and setter must be declared in the same class.

DynamoDBIndexHashKey

Maps a class property to the partition key of a global secondary index. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Use this annotation if you need to Query a global secondary index. You must specify the index name (`globalSecondaryIndexName`). If the name of the class property is different from the index partition key, you also must specify the name of that index attribute (`attributeName`).

DynamoDBIndexRangeKey

Maps a class property to the sort key of a global secondary index or a local secondary index. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Use this annotation if you need to Query a local secondary index or a global secondary index and want to refine your results using the index sort key. You must specify the index name (either `globalSecondaryIndexName` or `localSecondaryIndexName`). If the name of the class property is different from the index sort key, you must also specify the name of that index attribute (`attributeName`).

DynamoDBRangeKey

Maps a class property to the sort key of the table. The property must be one of the scalar string, number, or binary types. It cannot be a collection type.

If the primary key is composite (partition key and sort key), you can use this tag to map your class field to the sort key. For example, assume that you have a `Reply` table that stores replies for forum threads. Each thread can have many replies. So the primary key of this table is both the `ThreadId` and `ReplyDateTime`. The `ThreadId` is the partition key, and `ReplyDateTime` is the sort key.

The following Java code defines a `Reply` class and maps it to the `Reply` table. It uses both the `@DynamoDBHashKey` and `@DynamoDBRangeKey` tags to identify class properties that map to the primary key.

```
@DynamoDBTable(tableName="Reply")
public class Reply {
    private Integer id;
    private String replyDateTime;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    @DynamoDBRangeKey(attributeName="ReplyDateTime")
    public String getReplyDateTime() { return replyDateTime; }
    public void setReplyDateTime(String replyDateTime) { this.replyDateTime =
        replyDateTime; }

    // Additional properties go here.
}
```

DynamoDBTable

Identifies the target table in DynamoDB. For example, the following Java code defines a class `Developer` and maps it to the `People` table in DynamoDB.

```
@DynamoDBTable(tableName="People")
public class Developer { ... }
```

The `@DynamoDBTable` annotation can be inherited. Any new class that inherits from the `Developer` class also maps to the `People` table. For example, assume that you create a `Lead` class that inherits from the `Developer` class. Because you mapped the `Developer` class to the `People` table, the `Lead` class objects are also stored in the same table.

The `@DynamoDBTable` can also be overridden. Any new class that inherits from the `Developer` class by default maps to the same `People` table. However, you can override this default mapping. For example, if you create a class that inherits from the `Developer` class, you can explicitly map it to another table by adding the `@DynamoDBTable` annotation as shown in the following Java code example.

```
@DynamoDBTable(tableName="Managers")
```

```
public class Manager extends Developer { ...}
```

DynamoDBTypeConverted

An annotation to mark a property as using a custom type converter. Can be annotated on a user-defined annotation to pass additional properties to the `DynamoDBTypeConverter`.

The `DynamoDBTypeConverter` interface lets you map your own arbitrary data types to a data type that is natively supported by DynamoDB. For more information, see [Mapping arbitrary data](#).

DynamoDBTyped

An annotation to override the standard attribute type binding. Standard types do not require the annotation if applying the default attribute binding for that type.

DynamoDBVersionAttribute

Identifies a class property for storing an optimistic locking version number. `DynamoDBMapper` assigns a version number to this property when it saves a new item, and increments it each time you update the item. Only number scalar types are supported. For more information about data types, see [Data types](#). For more information about versioning, see [Optimistic locking with version number](#).

DynamoDBMapper Class

The `DynamoDBMapper` class is the entry point to Amazon DynamoDB. It provides access to a DynamoDB endpoint and enables you to access your data in various tables. It also enables you to perform various create, read, update, and delete (CRUD) operations on items, and run queries and scans against tables. This class provides the following methods for working with DynamoDB.

For the corresponding Javadoc documentation, see [DynamoDBMapper in the AWS SDK for Java API Reference](#).

Topics

- [save](#)
- [load](#)
- [delete](#)
- [query](#)
- [queryPage](#)

- [scan](#)
- [scanPage](#)
- [parallelScan](#)
- [batchSave](#)
- [batchLoad](#)
- [batchDelete](#)
- [batchWrite](#)
- [transactionWrite](#)
- [transactionLoad](#)
- [count](#)
- [generateCreateTableRequest](#)
- [createS3Link](#)
- [getS3ClientCache](#)

save

Saves the specified object to the table. The object that you want to save is the only required parameter for this method. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object.

If an item that has the same primary key does not exist, this method creates a new item in the table. If an item that has the same primary key exists, it updates the existing item. If the partition key and sort key are of type `String` and are annotated with `@DynamoDBAutoGeneratedKey`, they are given a random universally unique identifier (UUID) if left uninitialized. Version fields that are annotated with `@DynamoDBVersionAttribute` are incremented by one. Additionally, if a version field is updated or a key generated, the object passed in is updated as a result of the operation.

By default, only attributes corresponding to mapped class properties are updated. Any additional existing attributes on an item are unaffected. However, if you specify `SaveBehavior.CLOBBER`, you can force the item to be completely overwritten.

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER).build();

mapper.save(item, config);
```

If you have versioning enabled, the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see [Optimistic locking with version number](#).

load

Retrieves an item from a table. You must provide the primary key of the item that you want to retrieve. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object. For example, you can optionally request strongly consistent reads to ensure that this method retrieves only the latest item values as shown in the following Java statement.

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT).build();

CatalogItem item = mapper.load(CatalogItem.class, item.getId(), config);
```

By default, DynamoDB returns the item that has values that are eventually consistent. For information about the eventual consistency model of DynamoDB, see [Read consistency](#).

delete

Deletes an item from the table. You must pass in an object instance of the mapped class.

If you have versioning enabled, the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see [Optimistic locking with version number](#).

query

Queries a table or a secondary index. You can query a table or an index only if it has a composite primary key (partition key and sort key). This method requires you to provide a partition key value and a query filter that is applied on the sort key. A filter expression includes a condition and a value.

Assume that you have a table, `Reply`, that stores forum thread replies. Each thread subject can have zero or more replies. The primary key of the `Reply` table consists of the `Id` and `ReplyDateTime` fields, where `Id` is the partition key and `ReplyDateTime` is the sort key of the primary key.

```
Reply ( Id, ReplyDateTime, ... )
```

Assume that you created a mapping between a Reply class and the corresponding Reply table in DynamoDB. The following Java code uses DynamoDBMapper to find all replies in the past two weeks for a specific thread subject.

Example

```
String forumName = "&DDB;";
String forumSubject = "&DDB; Thread 1";
String partitionKey = forumName + "#" + forumSubject;

long twoWeeksAgoMilli = (new Date()).getTime() - (14L*24L*60L*60L*1000L);
Date twoWeeksAgo = new Date();
twoWeeksAgo.setTime(twoWeeksAgoMilli);
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
String twoWeeksAgoStr = df.format(twoWeeksAgo);

Map<String,AttributeValue> eav = new HashMap<String,AttributeValue>();
eav.put(":v1", new AttributeValue().withS(partitionKey));
eav.put(":v2", new AttributeValue().withS(twoWeeksAgoStr.toString()));

DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryExpression<Reply>()
    .withKeyConditionExpression("Id = :v1 and ReplyDateTime > :v2")
    .withExpressionAttributeValues(eav);

List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);
```

The query returns a collection of Reply objects.

By default, the query method returns a "lazy-loaded" collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, iterate over the latestReplies collection.

Note that calling the size() method on the collection will load every result in order to provide an accurate count. This can result in a lot of provisioned throughput being consumed, and on a very large table could even exhaust all the memory in your JVM.

To query an index, you must first model the index as a mapper class. Suppose that the Reply table has a global secondary index named *PostedBy-Message-Index*. The partition key for this index is PostedBy, and the sort key is Message. The class definition for an item in the index would look like the following.

```
@DynamoDBTable(tableName="Reply")
```

```
public class PostedByMessage {  
    private String postedBy;  
    private String message;  
  
    @DynamoDBIndexHashKey(globalSecondaryIndexName = "PostedBy-Message-Index",  
    attributeName = "PostedBy")  
    public String getPostedBy() { return postedBy; }  
    public void setPostedBy(String postedBy) { this.postedBy = postedBy; }  
  
    @DynamoDBIndexRangeKey(globalSecondaryIndexName = "PostedBy-Message-Index",  
    attributeName = "Message")  
    public String getMessage() { return message; }  
    public void setMessage(String message) { this.message = message; }  
  
    // Additional properties go here.  
}
```

The `@DynamoDBTable` annotation indicates that this index is associated with the `Reply` table. The `@DynamoDBIndexHashKey` annotation denotes the partition key (`PostedBy`) of the index, and `@DynamoDBIndexRangeKey` denotes the sort key (`Message`) of the index.

Now you can use `DynamoDBMapper` to query the index, retrieving a subset of messages that were posted by a particular user. You do not need to specify the index name if you do not have conflicting mappings across tables and indexes and the mappings are already made in the mapper. The mapper will infer based on the primary key and sort key. The following code queries a global secondary index. Because global secondary indexes support eventually consistent reads but not strongly consistent reads, you must specify `withConsistentRead(false)`.

```
HashMap<String,AttributeValue> eav = new HashMap<String,AttributeValue>();  
eav.put(":v1", new AttributeValue().withS("User A"));  
eav.put(":v2", new AttributeValue().withS("DynamoDB"));  
  
DynamoDBQueryExpression<PostedByMessage> queryExpression = new  
DynamoDBQueryExpression<PostedByMessage>()  
    .withIndexName("PostedBy-Message-Index")  
    .withConsistentRead(false)  
    .withKeyConditionExpression("PostedBy = :v1 and begins_with(Message, :v2)")  
    .withExpressionAttributeValues(eav);  
  
List<PostedByMessage> iList = mapper.query(PostedByMessage.class, queryExpression);
```

The query returns a collection of `PostedByMessage` objects.

queryPage

Queries a table or secondary index and returns a single page of matching results. As with the query method, you must specify a partition key value and a query filter that is applied on the sort key attribute. However, queryPage returns only the first "page" of data, that is, the amount of data that fits in 1 MB

scan

Scans an entire table or a secondary index. You can optionally specify a FilterExpression to filter the result set.

Assume that you have a table, Reply, that stores forum thread replies. Each thread subject can have zero or more replies. The primary key of the Reply table consists of the Id and ReplyDateTime fields, where Id is the partition key and ReplyDateTime is the sort key of the primary key.

```
Reply ( Id, ReplyDateTime, ... )
```

If you mapped a Java class to the Reply table, you can use the DynamoDBMapper to scan the table. For example, the following Java code scans the entire Reply table, returning only the replies for a particular year.

Example

```
HashMap<String,AttributeValue> eav = new HashMap<String,AttributeValue>();
eav.put(":v1", new AttributeValue().withS("2015"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("begins_with(ReplyDateTime,:v1)")
    .withExpressionAttributeValues(eav);

List<Reply> replies = mapper.scan(Reply.class, scanExpression);
```

By default, the scan method returns a "lazy-loaded" collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, iterate over the replies collection.

Note that calling the size() method on the collection will load every result in order to provide an accurate count. This can result in a lot of provisioned throughput being consumed, and on a very large table could even exhaust all the memory in your JVM.

To scan an index, you must first model the index as a mapper class. Suppose that the Reply table has a global secondary index named PostedBy-Message-Index. The partition key for this index is PostedBy, and the sort key is Message. A mapper class for this index is shown in the [query](#) section. It uses the `@DynamoDBIndexHashKey` and `@DynamoDBIndexRangeKey` annotations to specify the index partition key and sort key.

The following code example scans PostedBy-Message-Index. It does not use a scan filter, so all of the items in the index are returned to you.

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false);

List<PostedByMessage> iList = mapper.scan(PostedByMessage.class, scanExpression);
Iterator<PostedByMessage> indexItems = iList.iterator();
```

scanPage

Scans a table or secondary index and returns a single page of matching results. As with the `scan` method, you can optionally specify a `FilterExpression` to filter the result set. However, `scanPage` only returns the first "page" of data, that is, the amount of data that fits within 1 MB.

parallelScan

Performs a parallel scan of an entire table or secondary index. You specify a number of logical segments for the table, along with a scan expression to filter the results. The `parallelScan` divides the scan task among multiple workers, one for each logical segment; the workers process the data in parallel and return the results.

The following Java code example performs a parallel scan on the Product table.

```
int numberOfThreads = 4;

Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":n", new AttributeValue().withN("100"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("Price <= :n")
    .withExpressionAttributeValues(eav);
```

```
List<Product> scanResult = mapper.parallelScan(Product.class, scanExpression, numberofThreads);
```

For a Java code example illustrating the usage of `parallelScan`, see [DynamoDBMapper Query and scan operations](#).

batchSave

Saves objects to one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code saves two items (books) to the `ProductCatalog` table.

```
Book book1 = new Book();
book1.setId(901);
book1.setProductCategory("Book");
book1.setTitle("Book 901 Title");

Book book2 = new Book();
book2.setId(902);
book2.setProductCategory("Book");
book2.setTitle("Book 902 Title");

mapper.batchSave(Arrays.asList(book1, book2));
```

batchLoad

Retrieves multiple items from one or more tables using their primary keys.

The following Java code retrieves two items from two different tables.

```
ArrayList<Object> itemsToGet = new ArrayList<Object>();

ForumItem forumItem = new ForumItem();
forumItem.setForumName("Amazon DynamoDB");
itemsToGet.add(forumItem);

ThreadItem threadItem = new ThreadItem();
threadItem.setForumName("Amazon DynamoDB");
threadItem.setSubject("Amazon DynamoDB thread 1 message text");
```

```
itemsToGet.add(threadItem);

Map<String, List<Object>> items = mapper.batchLoad(itemsToGet);
```

batchDelete

Deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code deletes two items (books) from the `ProductCatalog` table.

```
Book book1 = mapper.load(Book.class, 901);
Book book2 = mapper.load(Book.class, 902);
mapper.batchDelete(Arrays.asList(book1, book2));
```

batchWrite

Saves objects to and deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees or support versioning (conditional puts or deletes).

The following Java code writes a new item to the `Forum` table, writes a new item to the `Thread` table, and deletes an item from the `ProductCatalog` table.

```
// Create a Forum item to save
Forum forumItem = new Forum();
forumItem.setName("Test BatchWrite Forum");

// Create a Thread item to save
Thread threadItem = new Thread();
threadItem.setForumName("AmazonDynamoDB");
threadItem.setSubject("My sample question");

// Load a ProductCatalog item to delete
Book book3 = mapper.load(Book.class, 903);

List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
List<Book> objectsToDelete = Arrays.asList(book3);

mapper.batchWrite(objectsToWrite, objectsToDelete);
```

transactionWrite

Saves objects to and deletes objects from one or more tables using one call to the `AmazonDynamoDB.transactWriteItems` method.

For a list of transaction-specific exceptions, see [TransactWriteItems errors](#).

For more information about DynamoDB transactions and the provided atomicity, consistency, isolation, and durability (ACID) guarantees see [Amazon DynamoDB Transactions](#).

 **Note**

This method does not support the following:

- [DynamoDBMapperConfig.SaveBehavior](#).

The following Java code writes a new item to each of the `Forum` and `Thread` tables, transactionally.

```
Thread s3ForumThread = new Thread();
s3ForumThread.setForumName("S3 Forum");
s3ForumThread.setSubject("Sample Subject 1");
s3ForumThread.setMessage("Sample Question 1");

Forum s3Forum = new Forum();
s3Forum.setName("S3 Forum");
s3Forum.setCategory("Amazon Web Services");
s3Forum.setThreads(1);

TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
transactionWriteRequest.addPut(s3Forum);
transactionWriteRequest.addPut(s3ForumThread);
mapper.transactionWrite(transactionWriteRequest);
```

transactionLoad

Loads objects from one or more tables using one call to the `AmazonDynamoDB.transactGetItems` method.

For a list of transaction-specific exceptions, see [TransactGetItems errors](#).

For more information about DynamoDB transactions and the provided atomicity, consistency, isolation, and durability (ACID) guarantees see [Amazon DynamoDB Transactions](#).

The following Java code loads one item from each of the Forum and Thread tables, transactionally.

```
Forum dynamodbForum = new Forum();
dynamodbForum.setName("DynamoDB Forum");
Thread dynamodbForumThread = new Thread();
dynamodbForumThread.setForumName("DynamoDB Forum");

TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();
transactionLoadRequest.addLoad(dynamodbForum);
transactionLoadRequest.addLoad(dynamodbForumThread);
mapper.transactionLoad(transactionLoadRequest);
```

count

Evaluates the specified scan expression and returns the count of matching items. No item data is returned.

generateCreateTableRequest

Parses a POJO class that represents a DynamoDB table, and returns a CreateTableRequest for that table.

createS3Link

Creates a link to an object in Amazon S3. You must specify a bucket name and a key name, which uniquely identifies the object in the bucket.

To use `createS3Link`, your mapper class must define getter and setter methods. The following code example illustrates this by adding a new attribute and getter/setter methods to the `CatalogItem` class.

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    ...

    public S3Link productImage;

    ...
}
```

```
@DynamoDBAttribute(attributeName = "ProductImage")
public S3Link getProductImage() {
    return productImage;
}

public void setProductImage(S3Link productImage) {
    this.productImage = productImage;
}

...
}
```

The following Java code defines a new item to be written to the Product table. The item includes a link to a product image; the image data is uploaded to Amazon S3.

```
CatalogItem item = new CatalogItem();

item.setId(150);
item.setTitle("Book 150 Title");

String myS3Bucket = "myS3bucket";
String myS3Key = "productImages/book_150_cover.jpg";
item.setProductImage(mapper.createS3Link(myS3Bucket, myS3Key));

item.getProductImage().uploadFrom(new File("/file/path/book_150_cover.jpg"));

mapper.save(item);
```

The S3Link class provides many other methods for manipulating objects in Amazon S3. For more information, see the [Javadocs for S3Link](#).

getS3ClientCache

Returns the underlying S3ClientCache for accessing Amazon S3. An S3ClientCache is a smart Map for AmazonS3Client objects. If you have multiple clients, an S3ClientCache can help you keep the clients organized by AWS Region, and can create new Amazon S3 clients on demand.

Optional configuration settings for DynamoDBMapper

When you create an instance of DynamoDBMapper, it has certain default behaviors; you can override these defaults by using the DynamoDBMapperConfig class.

The following code snippet creates a `DynamoDBMapper` with custom settings:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapperConfig mapperConfig = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER)
    .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSTANT)
    .withTableNameOverride(null)

    .withPaginationLoadingStrategy(DynamoDBMapperConfig.PaginationLoadingStrategy.EAGER_LOADING)
    .build();

DynamoDBMapper mapper = new DynamoDBMapper(client, mapperConfig);
```

For more information, see [DynamoDBMapperConfig](#) in the [AWS SDK for Java API Reference](#).

You can use the following arguments for an instance of `DynamoDBMapperConfig`:

- A `DynamoDBMapperConfig.ConsistentReads` enumeration value:
 - `EVENTUAL`—the mapper instance uses an eventually consistent read request.
 - `CONSISTENT`—the mapper instance uses a strongly consistent read request. You can use this optional setting with load, query, or scan operations. Strongly consistent reads have implications for performance and billing; see the [DynamoDB product detail page](#) for more information.

If you do not specify a read consistency setting for your mapper instance, the default is `EVENTUAL`.

 **Note**

This value is applied in the `query`, `querypage`, `load`, and `batch load` operations of the `DynamoDBMapper`.

- A `DynamoDBMapperConfig.PaginationLoadingStrategy` enumeration value—Controls how the mapper instance processes a paginated list of data, such as the results from a `query` or `scan`:
 - `LAZY_LOADING`—the mapper instance loads data when possible, and keeps all loaded results in memory.
 - `EAGER_LOADING`—the mapper instance loads the data as soon as the list is initialized.

- **ITERATION_ONLY**—you can only use an Iterator to read from the list. During the iteration, the list will clear all the previous results before loading the next page, so that the list will keep at most one page of the loaded results in memory. This also means the list can only be iterated once. This strategy is recommended when handling large items, in order to reduce memory overhead.

If you do not specify a pagination loading strategy for your mapper instance, the default is **LAZY_LOADING**.

- A `DynamoDBMapperConfig.SaveBehavior` enumeration value - Specifies how the mapper instance should deal with attributes during save operations:
 - **UPDATE**—during a save operation, all modeled attributes are updated, and unmodeled attributes are unaffected. Primitive number types (byte, int, long) are set to 0. Object types are set to null.
 - **CL0BBER**—clears and replaces all attributes, included unmodeled ones, during a save operation. This is done by deleting the item and re-creating it. Versioned field constraints are also disregarded.

If you do not specify the save behavior for your mapper instance, the default is **UPDATE**.

 **Note**

DynamoDBMapper transactional operations do not support `DynamoDBMapperConfig.SaveBehavior` enumeration.

- A `DynamoDBMapperConfig.TableNameOverride` object—Instructs the mapper instance to ignore the table name specified by a class's `DynamoDBTable` annotation, and instead use a different table name that you supply. This is useful when partitioning your data into multiple tables at runtime.

You can override the default configuration object for `DynamoDBMapper` per operation, as needed.

Optimistic locking with version number

Optimistic locking is a strategy to ensure that the client-side item that you are updating (or deleting) is the same as the item in Amazon DynamoDB. If you use this strategy, your database writes are protected from being overwritten by the writes of others, and vice versa.

With optimistic locking, each item has an attribute that acts as a version number. If you retrieve an item from a table, the application records the version number of that item. You can update the item, but only if the version number on the server side has not changed. If there is a version mismatch, it means that someone else has modified the item before you did. The update attempt fails, because you have a stale version of the item. If this happens, try again by retrieving the item and then trying to update it. Optimistic locking prevents you from accidentally overwriting changes that were made by others. It also prevents others from accidentally overwriting your changes.

While you can implement your own optimistic locking strategy, the AWS SDK for Java provides the `@DynamoDBVersionAttribute` annotation. In the mapping class for your table, you designate one property to store the version number, and mark it using this annotation. When you save an object, the corresponding item in the DynamoDB table will have an attribute that stores the version number. The `DynamoDBMapper` assigns a version number when you first save the object, and it automatically increments the version number each time you update the item. Your update or delete requests succeed only if the client-side object version matches the corresponding version number of the item in the DynamoDB table.

`ConditionalCheckFailedException` is thrown if:

- You use optimistic locking with `@DynamoDBVersionAttribute` and the version value on the server is different from the value on the client side.
- You specify your own conditional constraints while saving data by using `DynamoDBMapper` with `DynamoDBSaveExpression` and these constraints failed.

 **Note**

- DynamoDB global tables use a “last writer wins” reconciliation between concurrent updates. If you use global tables, last writer policy wins. So in this case, the locking strategy does not work as expected.
- `DynamoDBMapper` transactional write operations do not support `@DynamoDBVersionAttribute` annotation and condition expressions on the same object. If an object within a transactional write is annotated with `@DynamoDBVersionAttribute` and also has a condition expression, then an `SdkClientException` will be thrown.

For example, the following Java code defines a `CatalogItem` class that has several properties. The `Version` property is tagged with the `@DynamoDBVersionAttribute` annotation.

Example

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;
    private Long version;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer Id) { this.id = Id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN; }

    @DynamoDBAttribute(attributeName = "Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors =
bookAuthors; }

    @DynamoDBIgnore
    public String getSomeProp() { return someProp; }
    public void setSomeProp(String someProp) {this.someProp = someProp; }

    @DynamoDBVersionAttribute
    public Long getVersion() { return version; }
    public void setVersion(Long version) { this.version = version; }
}
```

You can apply the `@DynamoDBVersionAttribute` annotation to nullable types provided by the primitive wrappers classes that provide a nullable type, such as `Long` and `Integer`.

Optimistic locking has the following impact on these `DynamoDBMapper` methods:

- `save` — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the `save` operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- `delete` — The `delete` method takes an object as a parameter, and the `DynamoDBMapper` performs a version check before deleting the item. The version check can be disabled if `DynamoDBMapperConfig.SaveBehavior.CLOBBER` is specified in the request.

The internal implementation of optimistic locking within `DynamoDBMapper` uses conditional update and conditional delete support provided by DynamoDB.

- `transactionWrite` —

- `Put` — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the `put` operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- `Update` — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the `update` operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- `Delete` — The `DynamoDBMapper` performs a version check before deleting the item. The `delete` operation succeeds only if the version number on the client side and the server side match.
- `ConditionCheck` — The `@DynamoDBVersionAttribute` annotation is not supported for `ConditionCheck` operations. An `SdkClientException` will be thrown when a `ConditionCheck` item is annotated with `@DynamoDBVersionAttribute`.

Disabling optimistic locking

To disable optimistic locking, you can change the `DynamoDBMapperConfig.SaveBehavior` enumeration value from `UPDATE` to `CLOBBER`. You can do this by creating a `DynamoDBMapperConfig` instance that skips version checking and use this instance for all your requests. For information about `DynamoDBMapperConfig.SaveBehavior` and other optional `DynamoDBMapper` parameters, see [Optional configuration settings for `DynamoDBMapper`](#).

You can also set locking behavior for a specific operation only. For example, the following Java snippet uses the `DynamoDBMapper` to save a catalog item. It specifies `DynamoDBMapperConfig.SaveBehavior` by adding the optional `DynamoDBMapperConfig` parameter to the `save` method.

 **Note**

The `transactionWrite` method does not support `DynamoDBMapperConfig.SaveBehavior` configuration. Disabling optimistic locking for `transactionWrite` is not supported.

Example

```
DynamoDBMapper mapper = new DynamoDBMapper(client);

// Load a catalog item.
CatalogItem item = mapper.load(CatalogItem.class, 101);
item.setTitle("This is a new title for the item");
...
// Save the item.
mapper.save(item,
    new DynamoDBMapperConfig(
        DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

Mapping arbitrary data

In addition to the supported Java types (see [Supported data types for DynamoDB Mapper for Java](#)), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. To map these types, you must provide an implementation that converts your complex type to a DynamoDB supported type and vice versa, and annotate the complex type accessor method using the `@DynamoDBTypeConverted` annotation. The converter code transforms data when objects are saved or loaded. It is also used for all operations that consume complex types. Note that when comparing data during query and scan operations, the comparisons are made against the data stored in DynamoDB.

For example, consider the following `CatalogItem` class that defines a property, `Dimension`, that is of `DimensionType`. This property stores the item dimensions as height, width, and thickness. Assume that you decide to store these item dimensions as a string (such as `8.5x11x.05`) in DynamoDB. The following example provides converter code that converts the `DimensionType` object to a string and a string to the `DimensionType`.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

Example

```
public class DynamoDBMapperExample {  
  
    static AmazonDynamoDB client;  
  
    public static void main(String[] args) throws IOException {  
  
        // Set the AWS region you want to access.  
        Regions usWest2 = Regions.US_WEST_2;  
        client = AmazonDynamoDBClientBuilder.standard().withRegion(usWest2).build();  
  
        DimensionType dimType = new DimensionType();  
        dimType.setHeight("8.00");  
        dimType.setLength("11.0");  
        dimType.setThickness("1.0");  
  
        Book book = new Book();  
        book.setId(502);  
        book.setTitle("Book 502");  
        book.setISBN("555-5555555555");  
        book.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Author2")));  
        book.setDimensions(dimType);  
  
        DynamoDBMapper mapper = new DynamoDBMapper(client);  
        mapper.save(book);  
  
        Book bookRetrieved = mapper.load(Book.class, 502);  
        System.out.println("Book info: " + "\n" + bookRetrieved);  
  
        bookRetrieved.getDimensions().setHeight("9.0");  
        bookRetrieved.getDimensions().setLength("12.0");  
        bookRetrieved.getDimensions().setThickness("2.0");
```

```
    mapper.save(bookRetrieved);

    bookRetrieved = mapper.load(Book.class, 502);
    System.out.println("Updated book info: " + "\n" + bookRetrieved);
}

@DynamoDBTable(tableName = "ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private DimensionType dimensionType;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @DynamoDBAttribute(attributeName = "ISBN")
    public String getISBN() {
        return ISBN;
    }

    public void setISBN(String ISBN) {
        this.ISBN = ISBN;
    }

    @DynamoDBAttribute(attributeName = "Authors")
```

```
public Set<String> getBookAuthors() {
    return bookAuthors;
}

public void setBookAuthors(Set<String> bookAuthors) {
    this.bookAuthors = bookAuthors;
}

@DynamoDBTypeConverted(converter = DimensionTypeConverter.class)
@DynamoDBAttribute(attributeName = "Dimensions")
public DimensionType getDimensions() {
    return dimensionType;
}

@DynamoDBAttribute(attributeName = "Dimensions")
public void setDimensions(DimensionType dimensionType) {
    this.dimensionType = dimensionType;
}

@Override
public String toString() {
    return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors + ",
dimensionType= "
        + dimensionType.getHeight() + " X " + dimensionType.getLength() + "
X "
        + dimensionType.getThickness()
        + ", Id=" + id + ", Title=" + title + "]";
}
}

static public class DimensionType {

    private String length;
    private String height;
    private String thickness;

    public String getLength() {
        return length;
    }

    public void setLength(String length) {
        this.length = length;
    }
}
```

```
public String getHeight() {
    return height;
}

public void setHeight(String height) {
    this.height = height;
}

public String getThickness() {
    return thickness;
}

public void setThickness(String thickness) {
    this.thickness = thickness;
}

// Converts the complex type DimensionType to a string and vice-versa.
static public class DimensionTypeConverter implements DynamoDBTypeConverter<String,
DimensionType> {

    @Override
    public String convert(DimensionType object) {
        DimensionType itemDimensions = (DimensionType) object;
        String dimension = null;
        try {
            if (itemDimensions != null) {
                dimension = String.format("%s x %s x %s",
itemDimensions.getLength(), itemDimensions.getHeight(),
                    itemDimensions.getThickness());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return dimension;
    }

    @Override
    public DimensionType unconvert(String s) {

        DimensionType itemDimension = new DimensionType();
        try {
            if (s != null && s.length() != 0) {
                String[] data = s.split("x");
                itemDimension.setLength(Integer.parseInt(data[0]));
                itemDimension.setHeight(Integer.parseInt(data[1]));
                itemDimension.setThickness(Integer.parseInt(data[2]));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return itemDimension;
    }
}
```

```
        itemDimension.setLength(data[0].trim());
        itemDimension.setHeight(data[1].trim());
        itemDimension.setThickness(data[2].trim());
    }
} catch (Exception e) {
    e.printStackTrace();
}

return itemDimension;
}
}
```

DynamoDBMapper Examples

The following Java code examples demonstrate how to perform a variety of operations with the DynamoDBMapper class. You can use these examples to perform CRUD, query, scan, batch, and transaction operations.

Topics

- [DynamoDBMapper CRUD operations](#)
- [DynamoDBMapper Query and scan operations](#)
- [DynamoDBMapper batch operations](#)
- [DynamoDBMapper Transaction operations](#)

DynamoDBMapper CRUD operations

The following Java code example declares a CatalogItem class that has Id, Title, ISBN, and Authors properties. It uses the annotations to map these properties to the ProductCatalog table in DynamoDB. The example then uses the DynamoDBMapper to save a book object, retrieve it, update it, and then delete the book item.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

Imports

```
import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapperConfig;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
```

Code

```
public class DynamoDBMapperCRUDExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

    public static void main(String[] args) throws IOException {
        testCRUDOperations();
        System.out.println("Example complete!");
    }

    @DynamoDBTable(tableName = "ProductCatalog")
    public static class CatalogItem {
        private Integer id;
        private String title;
        private String ISBN;
        private Set<String> bookAuthors;

        // Partition key
        @DynamoDBHashKey(attributeName = "Id")
        public Integer getId() {
            return id;
        }

        public void setId(Integer id) {
            this.id = id;
        }
    }
}
```

```
@DynamoDBAttribute(attributeName = "Title")
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

@dynamoDBAttribute(attributeName = "ISBN")
public String getISBN() {
    return ISBN;
}

public void setISBN(String ISBN) {
    this.ISBN = ISBN;
}

@dynamoDBAttribute(attributeName = "Authors")
public Set<String> getBookAuthors() {
    return bookAuthors;
}

public void setBookAuthors(Set<String> bookAuthors) {
    this.bookAuthors = bookAuthors;
}

@Override
public String toString() {
    return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors + ", id=" + id
+ ", title=" + title + "]";
}

private static void testCRUDOperations() {

    CatalogItem item = new CatalogItem();
    item.setId(601);
    item.setTitle("Book 601");
    item.setISBN("611-1111111111");
    item.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Author2")));

    // Save the item (book).
    DynamoDBMapper mapper = new DynamoDBMapper(client);
```

```
mapper.save(item);

// Retrieve the item.
CatalogItem itemRetrieved = mapper.load(CatalogItem.class, 601);
System.out.println("Item retrieved:");
System.out.println(itemRetrieved);

// Update the item.
itemRetrieved.setISBN("622-222222222");
itemRetrieved.setBookAuthors(new HashSet<String>(Arrays.asList("Author1",
"Author3")));
mapper.save(itemRetrieved);
System.out.println("Item updated:");
System.out.println(itemRetrieved);

// Retrieve the updated item.
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT)
    .build();
CatalogItem updatedItem = mapper.load(CatalogItem.class, 601, config);
System.out.println("Retrieved the previously updated item:");
System.out.println(updatedItem);

// Delete the item.
mapper.delete(updatedItem);

// Try to retrieve deleted item.
CatalogItem deletedItem = mapper.load(CatalogItem.class, updatedItem.getId(),
config);
if (deletedItem == null) {
    System.out.println("Done - Sample item is deleted.");
}
}
```

DynamoDBMapper Query and scan operations

The Java example in this section defines the following classes and maps them to the tables in Amazon DynamoDB. For more information about creating sample tables, see [Creating tables and loading data for code examples in DynamoDB](#).

- The Book class maps to ProductCatalog table

- The Forum, Thread, and Reply classes map to tables of the same name.

The example then runs the following query and scan operations using a DynamoDBMapper instance.

- Get a book by Id.

The ProductCatalog table has Id as its primary key. It does not have a sort key as part of its primary key. Therefore, you cannot query the table. You can get an item using its Id value.

- Run the following queries against the Reply table.

The Reply table's primary key is composed of Id and ReplyDateTime attributes.

ReplyDateTime is a sort key. Therefore, you can query this table.

- Find replies to a forum thread posted in the last 15 days.
- Find replies to a forum thread posted in a specific date range.
- Scan the ProductCatalog table to find books whose price is less than a specified value.

For performance reasons, you should use the query operation instead of the scan operation. However, there are times you might need to scan a table. Suppose that there was a data entry error and one of the book prices was set to less than 0. This example scans the ProductCategory table to find book items (ProductCategory is book) whose price is less than 0.

- Perform a parallel scan of the ProductCatalog table to find bicycles of a specific type.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

Imports

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TimeZone;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBQueryExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBScanExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
```

Code

```
public class DynamoDBMapperQueryScanExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

    public static void main(String[] args) throws Exception {
        try {

            DynamoDBMapper mapper = new DynamoDBMapper(client);

            // Get a book - Id=101
            GetBook(mapper, 101);
            // Sample forum and thread to test queries.
            String forumName = "Amazon DynamoDB";
            String threadSubject = "DynamoDB Thread 1";
            // Sample queries.
            FindRepliesInLast15Days(mapper, forumName, threadSubject);
            FindRepliesPostedWithinTimePeriod(mapper, forumName, threadSubject);

            // Scan a table and find book items priced less than specified
            // value.
            FindBooksPricedLessThanSpecifiedValue(mapper, "20");

            // Scan a table with multiple threads and find bicycle items with a
            // specified bicycle type
            int numberOfThreads = 16;
```

```
        FindBicyclesOfTypeWithMultipleThreads(mapper, numberOfThreads,
"Road");

        System.out.println("Example complete!");

    } catch (Throwable t) {
        System.err.println("Error running the DynamoDBMapperQueryScanExample: " +
t);
        t.printStackTrace();
    }
}

private static void GetBook(DynamoDBMapper mapper, int id) throws Exception {
    System.out.println("GetBook: Get book Id='101' ");
    System.out.println("Book table has no sort key. You can do GetItem, but not
Query.");
    Book book = mapper.load(Book.class, id);
    System.out.format("Id = %s Title = %s, ISBN = %s %n", book.getId(),
book.getTitle(), book.getISBN());
}

private static void FindRepliesInLast15Days(DynamoDBMapper mapper, String
forumName, String threadSubject)
    throws Exception {
    System.out.println("FindRepliesInLast15Days: Replies within last 15 days.");

    String partitionKey = forumName + "#" + threadSubject;

    long twoWeeksAgoMilli = (new Date()).getTime() - (15L * 24L * 60L * 60L *
1000L);
    Date twoWeeksAgo = new Date();
    twoWeeksAgo.setTime(twoWeeksAgoMilli);
    SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");
    dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
    String twoWeeksAgoStr = dateFormatter.format(twoWeeksAgo);

    Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
    eav.put(":val1", new AttributeValue().withS(partitionKey));
    eav.put(":val2", new AttributeValue().withS(twoWeeksAgoStr.toString()));

    DynamoDBQueryExpression<Reply> queryExpression = new
DynamoDBQueryExpression<Reply>()
```

```
.withKeyConditionExpression("Id = :val1 and ReplyDateTime
> :val2").withExpressionAttributeValues(eav);

List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);

for (Reply reply : latestReplies) {
    System.out.format("Id=%s, Message=%s, PostedBy=%s %n, ReplyDateTime=%s %n",
reply.getId(),
            reply.getMessage(), reply.getPostedBy(), reply.getReplyDateTime());
}
}

private static void FindRepliesPostedWithinTimePeriod(DynamoDBMapper mapper, String
forumName, String threadSubject)
    throws Exception {
String partitionKey = forumName + "#" + threadSubject;

System.out.println(
        "FindRepliesPostedWithinTimePeriod: Find replies for thread Message =
'DynamoDB Thread 2' posted within a period.");
    long startDateMilli = (new Date()).getTime() - (14L * 24L * 60L * 60L *
1000L); // Two

// weeks

// ago.
    long endDateMilli = (new Date()).getTime() - (7L * 24L * 60L * 60L * 1000L); // One
                                                                //
week
                                                                //
ago.

    SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");
    dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
    String startDate = dateFormatter.format(startDateMilli);
    String endDate = dateFormatter.format(endDateMilli);

    Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
    eav.put(":val1", new AttributeValue().withS(partitionKey));
    eav.put(":val2", new AttributeValue().withS(startDate));
    eav.put(":val3", new AttributeValue().withS(endDate));
```

```
DynamoDBQueryExpression<Reply> queryExpression = new
DynamoDBQueryExpression<Reply>()
    .withKeyConditionExpression("Id = :val1 and ReplyDateTime between :val2
and :val3")
    .withExpressionAttributeValues(eav);

List<Reply> betweenReplies = mapper.query(Reply.class, queryExpression);

for (Reply reply : betweenReplies) {
    System.out.format("Id=%s, Message=%s, PostedBy=%s %n, PostedDateTime=%s
%n", reply.getId(),
        reply.getMessage(), reply.getPostedBy(), reply.getReplyDateTime());
}

}

private static void FindBooksPricedLessThanSpecifiedValue(DynamoDBMapper mapper,
String value) throws Exception {

    System.out.println("FindBooksPricedLessThanSpecifiedValue: Scan
ProductCatalog.");

    Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
    eav.put(":val1", new AttributeValue().withN(value));
    eav.put(":val2", new AttributeValue().withS("Book"));

    DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
        .withFilterExpression("Price < :val1 and ProductCategory
= :val2").withExpressionAttributeValues(eav);

    List<Book> scanResult = mapper.scan(Book.class, scanExpression);

    for (Book book : scanResult) {
        System.out.println(book);
    }
}

private static void FindBicyclesOfTypeWithMultipleThreads(DynamoDBMapper
mapper, int numberOfThreads,
String bicycleType) throws Exception {

    System.out.println("FindBicyclesOfTypeWithMultipleThreads: Scan
ProductCatalog With Multiple Threads.");
    Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
```

```
eav.put(":val1", new AttributeValue().withS("Bicycle"));
eav.put(":val2", new AttributeValue().withS(bicycleType));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("ProductCategory = :val1 and BicycleType
= :val2")
    .withExpressionAttributeValues(eav);

List<Bicycle> scanResult = mapper.parallelScan(Bicycle.class, scanExpression,
numberOfThreads);
for (Bicycle bicycle : scanResult) {
    System.out.println(bicycle);
}
}

{@DynamoDBTable(tableName = "ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private int price;
    private int pageCount;
    private String productCategory;
    private boolean inPublication;

    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @DynamoDBAttribute(attributeName = "ISBN")}
```

```
public String getISBN() {
    return ISBN;
}

public void setISBN(String ISBN) {
    this.ISBN = ISBN;
}

@DynamoDBAttribute(attributeName = "Price")
public int getPrice() {
    return price;
}

public void setPrice(int price) {
    this.price = price;
}

@DynamoDBAttribute(attributeName = "PageCount")
public int getPageCount() {
    return pageCount;
}

public void setPageCount(int pageCount) {
    this.pageCount = pageCount;
}

@DynamoDBAttribute(attributeName = "ProductCategory")
public String getProductCategory() {
    return productCategory;
}

public void setProductCategory(String productCategory) {
    this.productCategory = productCategory;
}

@DynamoDBAttribute(attributeName = "InPublication")
public boolean getInPublication() {
    return inPublication;
}

public void setInPublication(boolean inPublication) {
    this.inPublication = inPublication;
}
```

```
    @Override
    public String toString() {
        return "Book [ISBN=" + ISBN + ", price=" + price + ", product category=" +
productCategory + ", id=" + id
        + ", title=" + title + "]";
    }

}

@DynamoDBTable(tableName = "ProductCatalog")
public static class Bicycle {
    private int id;
    private String title;
    private String description;
    private String bicycleType;
    private String brand;
    private int price;
    private List<String> color;
    private String productCategory;

    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @DynamoDBAttribute(attributeName = "Description")
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
```

```
        this.description = description;
    }

    @DynamoDBAttribute(attributeName = "BicycleType")
    public String getBicycleType() {
        return bicycleType;
    }

    public void setBicycleType(String bicycleType) {
        this.bicycleType = bicycleType;
    }

    @DynamoDBAttribute(attributeName = "Brand")
    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    @DynamoDBAttribute(attributeName = "Price")
    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @DynamoDBAttribute(attributeName = "Color")
    public List<String> getColor() {
        return color;
    }

    public void setColor(List<String> color) {
        this.color = color;
    }

    @DynamoDBAttribute(attributeName = "ProductCategory")
    public String getProductCategory() {
        return productCategory;
    }
```

```
public void setProductCategory(String productCategory) {
    this.productCategory = productCategory;
}

@Override
public String toString() {
    return "Bicycle [Type=" + bicycleType + ", color=" + color + ", price=" +
price + ", product category="
        + productCategory + ", id=" + id + ", title=" + title + "]";
}

}

@DynamoDBTable(tableName = "Reply")
public static class Reply {
    private String id;
    private String replyDateTime;
    private String message;
    private String postedBy;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    // Range key
    @DynamoDBRangeKey(attributeName = "ReplyDateTime")
    public String getReplyDateTime() {
        return replyDateTime;
    }

    public void setReplyDateTime(String replyDateTime) {
        this.replyDateTime = replyDateTime;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }
}
```

```
public void setMessage(String message) {
    this.message = message;
}

@DynamoDBAttribute(attributeName = "PostedBy")
public String getPostedBy() {
    return postedBy;
}

public void setPostedBy(String postedBy) {
    this.postedBy = postedBy;
}
}

@dynamoDBTable(tableName = "Thread")
public static class Thread {
    private String forumName;
    private String subject;
    private String message;
    private String lastPostedDateTime;
    private String lastPostedBy;
    private Set<String> tags;
    private int answered;
    private int views;
    private int replies;

    // Partition key
    @DynamoDBHashKey(attributeName = "ForumName")
    public String getForumName() {
        return forumName;
    }

    public void setForumName(String forumName) {
        this.forumName = forumName;
    }

    // Range key
    @DynamoDBRangeKey(attributeName = "Subject")
    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
```

```
        this.subject = subject;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @DynamoDBAttribute(attributeName = "LastPostedDateTime")
    public String getLastPostedDateTime() {
        return lastPostedDateTime;
    }

    public void setLastPostedDateTime(String lastPostedDateTime) {
        this.lastPostedDateTime = lastPostedDateTime;
    }

    @DynamoDBAttribute(attributeName = "LastPostedBy")
    public String getLastPostedBy() {
        return lastPostedBy;
    }

    public void setLastPostedBy(String lastPostedBy) {
        this.lastPostedBy = lastPostedBy;
    }

    @DynamoDBAttribute(attributeName = "Tags")
    public Set<String> getTags() {
        return tags;
    }

    public void setTags(Set<String> tags) {
        this.tags = tags;
    }

    @DynamoDBAttribute(attributeName = "Answered")
    public int getAnswered() {
        return answered;
    }
```

```
public void setAnswered(int answered) {
    this.answered = answered;
}

@DynamoDBAttribute(attributeName = "Views")
public int getViews() {
    return views;
}

public void setViews(int views) {
    this.views = views;
}

@DynamoDBAttribute(attributeName = "Replies")
public int getReplies() {
    return replies;
}

public void setReplies(int replies) {
    this.replies = replies;
}

}

@dynamoDBTable(tableName = "Forum")
public static class Forum {
    private String name;
    private String category;
    private int threads;

    @DynamoDBHashKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Category")
    public String getCategory() {
        return category;
    }
}
```

```
public void setCategory(String category) {
    this.category = category;
}

@DynamoDBAttribute(attributeName = "Threads")
public int getThreads() {
    return threads;
}

public void setThreads(int threads) {
    this.threads = threads;
}
}

}
```

DynamoDBMapper batch operations

The following Java code example declares Book, Forum, Thread, and Reply classes and maps them to the Amazon DynamoDB tables using the DynamoDBMapper class.

The code illustrates the following batch write operations:

- `batchSave` to put book items in the `ProductCatalog` table.
- `batchDelete` to delete items from the `ProductCatalog` table.
- `batchWrite` to put and delete items from the `Forum` and the `Thread` tables.

For more information about the tables used in this example, see [Creating tables and loading data for code examples in DynamoDB](#). For step-by-step instructions for testing the following example, see [Java code examples](#).

Imports

```
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapperConfig;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
```

Code

```
public class DynamoDBMapperBatchWriteExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

    public static void main(String[] args) throws Exception {
        try {

            DynamoDBMapper mapper = new DynamoDBMapper(client);

            testBatchSave(mapper);
            testBatchDelete(mapper);
            testBatchWrite(mapper);

            System.out.println("Example complete!");

        } catch (Throwable t) {
            System.err.println("Error running the DynamoDBMapperBatchWriteExample: " +
t);
            t.printStackTrace();
        }
    }

    private static void testBatchSave(DynamoDBMapper mapper) {

        Book book1 = new Book();
        book1.setId(901);
        book1.setInPublication(true);
        book1.setISBN("902-11-11-1111");
        book1.setPageCount(100);
        book1.setPrice(10);
        book1.setProductCategory("Book");
        book1.setTitle("My book created in batch write");
    }
}
```

```
Book book2 = new Book();
book2.setId(902);
book2.setInPublication(true);
book2.setISBN("902-11-12-1111");
book2.setPageCount(200);
book2.setPrice(20);
book2.setProductCategory("Book");
book2.setTitle("My second book created in batch write");

Book book3 = new Book();
book3.setId(903);
book3.setInPublication(false);
book3.setISBN("902-11-13-1111");
book3.setPageCount(300);
book3.setPrice(25);
book3.setProductCategory("Book");
book3.setTitle("My third book created in batch write");

System.out.println("Adding three books to ProductCatalog table.");
mapper.batchSave(Arrays.asList(book1, book2, book3));
}

private static void testBatchDelete(DynamoDBMapper mapper) {

    Book book1 = mapper.load(Book.class, 901);
    Book book2 = mapper.load(Book.class, 902);
    System.out.println("Deleting two books from the ProductCatalog table.");
    mapper.batchDelete(Arrays.asList(book1, book2));
}

private static void testBatchWrite(DynamoDBMapper mapper) {

    // Create Forum item to save
    Forum forumItem = new Forum();
    forumItem.setName("Test BatchWrite Forum");
    forumItem.setThreads(0);
    forumItem.setCategory("Amazon Web Services");

    // Create Thread item to save
    Thread threadItem = new Thread();
    threadItem.setForumName("AmazonDynamoDB");
    threadItem.setSubject("My sample question");
    threadItem.setMessage("BatchWrite message");
    List<String> tags = new ArrayList<String>();
```

```
tags.add("batch operations");
tags.add("write");
threadItem.setTags(new HashSet<String>(tags));

// Load ProductCatalog item to delete
Book book3 = mapper.load(Book.class, 903);

List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
List<Book> objectsToDelete = Arrays.asList(book3);

DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER)
    .build();

mapper.batchWrite(objectsToWrite, objectsToDelete, config);
}

@DynamoDBTable(tableName = "ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private int price;
    private int pageCount;
    private String productCategory;
    private boolean inPublication;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

```
}

@DynamoDBAttribute(attributeName = "ISBN")
public String getISBN() {
    return ISBN;
}

public void setISBN(String ISBN) {
    this.ISBN = ISBN;
}

@DynamoDBAttribute(attributeName = "Price")
public int getPrice() {
    return price;
}

public void setPrice(int price) {
    this.price = price;
}

@DynamoDBAttribute(attributeName = "PageCount")
public int getPageCount() {
    return pageCount;
}

public void setPageCount(int pageCount) {
    this.pageCount = pageCount;
}

@DynamoDBAttribute(attributeName = "ProductCategory")
public String getProductCategory() {
    return productCategory;
}

public void setProductCategory(String productCategory) {
    this.productCategory = productCategory;
}

@DynamoDBAttribute(attributeName = "InPublication")
public boolean getInPublication() {
    return inPublication;
}

public void setInPublication(boolean inPublication) {
```

```
        this.inPublication = inPublication;
    }

    @Override
    public String toString() {
        return "Book [ISBN=" + ISBN + ", price=" + price + ", product category=" +
productCategory + ", id=" + id
                + ", title=" + title + "]";
    }

}

@DynamoDBTable(tableName = "Reply")
public static class Reply {
    private String id;
    private String replyDateTime;
    private String message;
    private String postedBy;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    // Sort key
    @DynamoDBRangeKey(attributeName = "ReplyDateTime")
    public String getReplyDateTime() {
        return replyDateTime;
    }

    public void setReplyDateTime(String replyDateTime) {
        this.replyDateTime = replyDateTime;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }
}
```

```
public void setMessage(String message) {
    this.message = message;
}

@DynamoDBAttribute(attributeName = "PostedBy")
public String getPostedBy() {
    return postedBy;
}

public void setPostedBy(String postedBy) {
    this.postedBy = postedBy;
}
}

{@DynamoDBTable(tableName = "Thread")
public static class Thread {
    private String forumName;
    private String subject;
    private String message;
    private String lastPostedDateTime;
    private String lastPostedBy;
    private Set<String> tags;
    private int answered;
    private int views;
    private int replies;

    // Partition key
    @DynamoDBHashKey(attributeName = "ForumName")
    public String getForumName() {
        return forumName;
    }

    public void setForumName(String forumName) {
        this.forumName = forumName;
    }

    // Sort key
    @DynamoDBRangeKey(attributeName = "Subject")
    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }
}
```

```
}

@DynamoDBAttribute(attributeName = "Message")
public String getMessage() {
    return message;
}

public void setMessage(String message) {
    this.message = message;
}

@DynamoDBAttribute(attributeName = "LastPostedDateTime")
public String getLastPostedDateTime() {
    return lastPostedDateTime;
}

public void setLastPostedDateTime(String lastPostedDateTime) {
    this.lastPostedDateTime = lastPostedDateTime;
}

@DynamoDBAttribute(attributeName = "LastPostedBy")
public String getLastPostedBy() {
    return lastPostedBy;
}

public void setLastPostedBy(String lastPostedBy) {
    this.lastPostedBy = lastPostedBy;
}

@DynamoDBAttribute(attributeName = "Tags")
public Set<String> getTags() {
    return tags;
}

public void setTags(Set<String> tags) {
    this.tags = tags;
}

@DynamoDBAttribute(attributeName = "Answered")
public int getAnswered() {
    return answered;
}

public void setAnswered(int answered) {
```

```
        this.answered = answered;
    }

    @DynamoDBAttribute(attributeName = "Views")
    public int getViews() {
        return views;
    }

    public void setViews(int views) {
        this.views = views;
    }

    @DynamoDBAttribute(attributeName = "Replies")
    public int getReplies() {
        return replies;
    }

    public void setReplies(int replies) {
        this.replies = replies;
    }

}

{@DynamoDBTable(tableName = "Forum")
public static class Forum {
    private String name;
    private String category;
    private int threads;

    // Partition key
    @DynamoDBHashKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Category")
    public String getCategory() {
        return category;
    }
}
```

```
public void setCategory(String category) {
    this.category = category;
}

@DynamoDBAttribute(attributeName = "Threads")
public int getThreads() {
    return threads;
}

public void setThreads(int threads) {
    this.threads = threads;
}
}
```

DynamoDBMapper Transaction operations

The following Java code example declares a `Forum` and a `Thread` class and maps them to the DynamoDB tables using the `DynamoDBMapper` class.

The code illustrates the following transactional operations:

- `transactionWrite` to add, update, and delete multiple items from one or more tables in one transaction.
- `transactionLoad` to retrieve multiple items from one or more tables in one transaction.

Imports

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMappingException;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
```

```
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTransactionLoadExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTransactionWriteExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.TransactionLoadRequest;
import com.amazonaws.services.dynamodbv2.datamodeling.TransactionWriteRequest;
import com.amazonaws.services.dynamodbv2.model.InternalServerErrorException;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import com.amazonaws.services.dynamodbv2.model.TransactionCanceledException;
```

Code

```
public class DynamoDBMapperTransactionExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDBMapper mapper;

    public static void main(String[] args) throws Exception {
        try {

            mapper = new DynamoDBMapper(client);

            testPutAndUpdateInTransactionWrite();
            testPutWithConditionalUpdateInTransactionWrite();
            testPutWithConditionCheckInTransactionWrite();
            testMixedOperationsInTransactionWrite();
            testTransactionLoadWithSave();
            testTransactionLoadWithTransactionWrite();
            System.out.println("Example complete");

        } catch (Throwable t) {
            System.err.println("Error running the
DynamoDBMapperTransactionWriteExample: " + t);
            t.printStackTrace();
        }
    }

    private static void testTransactionLoadWithSave() {
        // Create new Forum item for DynamoDB using save
        Forum dynamodbForum = new Forum();
        dynamodbForum.setName("DynamoDB Forum");
        dynamodbForum.setCategory("Amazon Web Services");
        dynamodbForum.setThreads(0);
    }
}
```

```
mapper.save(dynamodbForum);

// Add a thread to DynamoDB Forum
Thread dynamodbForumThread = new Thread();
dynamodbForumThread.setForumName("DynamoDB Forum");
dynamodbForumThread.setSubject("Sample Subject 1");
dynamodbForumThread.setMessage("Sample Question 1");
mapper.save(dynamodbForumThread);

// Update DynamoDB Forum to reflect updated thread count
dynamodbForum.setThreads(1);
mapper.save(dynamodbForum);

// Read DynamoDB Forum item and Thread item at the same time in a serializable
// manner
TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();

// Read entire item for DynamoDB Forum
transactionLoadRequest.addLoad(dynamodbForum);

// Only read subject and message attributes from Thread item
DynamoDBTransactionLoadExpression loadExpressionForThread = new
DynamoDBTransactionLoadExpression()
    .withProjectionExpression("Subject, Message");
transactionLoadRequest.addLoad(dynamodbForumThread, loadExpressionForThread);

// Loaded objects are guaranteed to be in same order as the order in which they
// are
// added to TransactionLoadRequest
List<Object> loadedObjects = executeTransactionLoad(transactionLoadRequest);
Forum loadedDynamoDBForum = (Forum) loadedObjects.get(0);
System.out.println("Forum: " + loadedDynamoDBForum.getName());
System.out.println("Threads: " + loadedDynamoDBForum.getThreads());
Thread loadedDynamodbForumThread = (Thread) loadedObjects.get(1);
System.out.println("Subject: " + loadedDynamodbForumThread.getSubject());
System.out.println("Message: " + loadedDynamodbForumThread.getMessage());
}

private static void testTransactionLoadWithTransactionWrite() {
    // Create new Forum item for DynamoDB using save
    Forum dynamodbForum = new Forum();
    dynamodbForum.setName("DynamoDB New Forum");
    dynamodbForum.setCategory("Amazon Web Services");
    dynamodbForum.setThreads(0);
```

```
mapper.save(dynamodbForum);

// Update Forum item for DynamoDB and add a thread to DynamoDB Forum, in
// an ACID manner using transactionWrite

dynamodbForum.setThreads(1);
Thread dynamodbForumThread = new Thread();
dynamodbForumThread.setForumName("DynamoDB New Forum");
dynamodbForumThread.setSubject("Sample Subject 2");
dynamodbForumThread.setMessage("Sample Question 2");
TransactionWriteRequest transactionWriteRequest = new
TransactionWriteRequest();
transactionWriteRequest.addPut(dynamodbForumThread);
transactionWriteRequest.addUpdate(dynamodbForum);
executeTransactionWrite(transactionWriteRequest);

// Read DynamoDB Forum item and Thread item at the same time in a serializable
// manner
TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();

// Read entire item for DynamoDB Forum
transactionLoadRequest.addLoad(dynamodbForum);

// Only read subject and message attributes from Thread item
DynamoDBTransactionLoadExpression loadExpressionForThread = new
DynamoDBTransactionLoadExpression()
    .withProjectionExpression("Subject, Message");
transactionLoadRequest.addLoad(dynamodbForumThread, loadExpressionForThread);

// Loaded objects are guaranteed to be in same order as the order in which they
// are
// added to TransactionLoadRequest
List<Object> loadedObjects = executeTransactionLoad(transactionLoadRequest);
Forum loadedDynamoDBForum = (Forum) loadedObjects.get(0);
System.out.println("Forum: " + loadedDynamoDBForum.getName());
System.out.println("Threads: " + loadedDynamoDBForum.getThreads());
Thread loadedDynamodbForumThread = (Thread) loadedObjects.get(1);
System.out.println("Subject: " + loadedDynamodbForumThread.getSubject());
System.out.println("Message: " + loadedDynamodbForumThread.getMessage());
}

private static void testPutAndUpdateInTransactionWrite() {
// Create new Forum item for S3 using save
Forum s3Forum = new Forum();
```

```
s3Forum.setName("S3 Forum");
s3Forum.setCategory("Core Amazon Web Services");
s3Forum.setThreads(0);
mapper.save(s3Forum);

// Update Forum item for S3 and Create new Forum item for DynamoDB using
// transactionWrite
s3Forum.setCategory("Amazon Web Services");
Forum dynamodbForum = new Forum();
dynamodbForum.setName("DynamoDB Forum");
dynamodbForum.setCategory("Amazon Web Services");
dynamodbForum.setThreads(0);
TransactionWriteRequest transactionWriteRequest = new
TransactionWriteRequest();
transactionWriteRequest.addUpdate(s3Forum);
transactionWriteRequest.addPut(dynamodbForum);
executeTransactionWrite(transactionWriteRequest);
}

private static void testPutWithConditionalUpdateInTransactionWrite() {
    // Create new Thread item for DynamoDB forum and update thread count in
    DynamoDB
    // forum
    // if the DynamoDB Forum exists
    Thread dynamodbForumThread = new Thread();
    dynamodbForumThread.setForumName("DynamoDB Forum");
    dynamodbForumThread.setSubject("Sample Subject 1");
    dynamodbForumThread.setMessage("Sample Question 1");

    Forum dynamodbForum = new Forum();
    dynamodbForum.setName("DynamoDB Forum");
    dynamodbForum.setCategory("Amazon Web Services");
    dynamodbForum.setThreads(1);

    DynamoDBTransactionWriteExpression transactionWriteExpression = new
DynamoDBTransactionWriteExpression()
        .withConditionExpression("attribute_exists(Category)");

    TransactionWriteRequest transactionWriteRequest = new
TransactionWriteRequest();
    transactionWriteRequest.addPut(dynamodbForumThread);
    transactionWriteRequest.addUpdate(dynamodbForum, transactionWriteExpression);
    executeTransactionWrite(transactionWriteRequest);
}
```

```
private static void testPutWithConditionCheckInTransactionWrite() {
    // Create new Thread item for DynamoDB forum and update thread count in
DynamoDB
    // forum if a thread already exists
    Thread dynamodbForumThread2 = new Thread();
    dynamodbForumThread2.setForumName("DynamoDB Forum");
    dynamodbForumThread2.setSubject("Sample Subject 2");
    dynamodbForumThread2.setMessage("Sample Question 2");

    Thread dynamodbForumThread1 = new Thread();
    dynamodbForumThread1.setForumName("DynamoDB Forum");
    dynamodbForumThread1.setSubject("Sample Subject 1");
    DynamoDBTransactionWriteExpression conditionExpressionForConditionCheck = new
DynamoDBTransactionWriteExpression()
        .withConditionExpression("attribute_exists(Subject)");

    Forum dynamodbForum = new Forum();
    dynamodbForum.setName("DynamoDB Forum");
    dynamodbForum.setCategory("Amazon Web Services");
    dynamodbForum.setThreads(2);

    TransactionWriteRequest transactionWriteRequest = new
TransactionWriteRequest();
    transactionWriteRequest.addPut(dynamodbForumThread2);
    transactionWriteRequest.addConditionCheck(dynamodbForumThread1,
conditionExpressionForConditionCheck);
    transactionWriteRequest.addUpdate(dynamodbForum);
    executeTransactionWrite(transactionWriteRequest);
}

private static void testMixedOperationsInTransactionWrite() {
    // Create new Thread item for S3 forum and delete "Sample Subject 1" Thread
from
    // DynamoDB forum if
    // "Sample Subject 2" Thread exists in DynamoDB forum
    Thread s3ForumThread = new Thread();
    s3ForumThread.setForumName("S3 Forum");
    s3ForumThread.setSubject("Sample Subject 1");
    s3ForumThread.setMessage("Sample Question 1");

    Forum s3Forum = new Forum();
    s3Forum.setName("S3 Forum");
    s3Forum.setCategory("Amazon Web Services");
```

```
s3Forum.setThreads(1);

Thread dynamodbForumThread1 = new Thread();
dynamodbForumThread1.setForumName("DynamoDB Forum");
dynamodbForumThread1.setSubject("Sample Subject 1");

Thread dynamodbForumThread2 = new Thread();
dynamodbForumThread2.setForumName("DynamoDB Forum");
dynamodbForumThread2.setSubject("Sample Subject 2");
DynamoDBTransactionWriteExpression conditionExpressionForConditionCheck = new
DynamoDBTransactionWriteExpression()
    .withConditionExpression("attribute_exists(Subject)");

Forum dynamodbForum = new Forum();
dynamodbForum.setName("DynamoDB Forum");
dynamodbForum.setCategory("Amazon Web Services");
dynamodbForum.setThreads(1);

TransactionWriteRequest transactionWriteRequest = new
TransactionWriteRequest();
transactionWriteRequest.addPut(s3ForumThread);
transactionWriteRequest.addUpdate(s3Forum);
transactionWriteRequest.addDelete(dynamodbForumThread1);
transactionWriteRequest.addConditionCheck(dynamodbForumThread2,
conditionExpressionForConditionCheck);
transactionWriteRequest.addUpdate(dynamodbForum);
executeTransactionWrite(transactionWriteRequest);
}

private static List<Object> executeTransactionLoad(TransactionLoadRequest
transactionLoadRequest) {
    List<Object> loadedObjects = new ArrayList<Object>();
    try {
        loadedObjects = mapper.transactionLoad(transactionLoadRequest);
    } catch (DynamoDBMappingException ddbme) {
        System.err.println("Client side error in Mapper, fix before retrying.
Error: " + ddbme.getMessage());
    } catch (ResourceNotFoundException rnfe) {
        System.err.println("One of the tables was not found, verify table exists
before retrying. Error: "
                + rnfe.getMessage());
    } catch (InternalServerException ise) {
        System.err.println(
```

```
        "Internal Server Error, generally safe to retry with back-off.
Error: " + ise.getMessage());
    } catch (TransactionCanceledException tce) {
        System.err.println(
            "Transaction Canceled, implies a client issue, fix before retrying.
Error: " + tce.getMessage());
    } catch (Exception ex) {
        System.err.println(
            "An exception occurred, investigate and configure retry strategy.
Error: " + ex.getMessage());
    }
    return loadedObjects;
}

private static void executeTransactionWrite(TransactionWriteRequest transactionWriteRequest) {
    try {
        mapper.transactionWrite(transactionWriteRequest);
    } catch (DynamoDBMappingException ddbme) {
        System.err.println("Client side error in Mapper, fix before retrying.
Error: " + ddbme.getMessage());
    } catch (ResourceNotFoundException rnfe) {
        System.err.println("One of the tables was not found, verify table exists
before retrying. Error: "
            + rnfe.getMessage());
    } catch (InternalServerErrorException ise) {
        System.err.println(
            "Internal Server Error, generally safe to retry with back-off.
Error: " + ise.getMessage());
    } catch (TransactionCanceledException tce) {
        System.err.println(
            "Transaction Canceled, implies a client issue, fix before retrying.
Error: " + tce.getMessage());
    } catch (Exception ex) {
        System.err.println(
            "An exception occurred, investigate and configure retry strategy.
Error: " + ex.getMessage());
    }
}

@dynamoDBTable(tableName = "Thread")
public static class Thread {
    private String forumName;
    private String subject;
```

```
private String message;
private String lastPostedDateTime;
private String lastPostedBy;
private Set<String> tags;
private int answered;
private int views;
private int replies;

// Partition key
@DynamoDBHashKey(attributeName = "ForumName")
public String getForumName() {
    return forumName;
}

public void setForumName(String forumName) {
    this.forumName = forumName;
}

// Sort key
@DynamoDBRangeKey(attributeName = "Subject")
public String getSubject() {
    return subject;
}

public void setSubject(String subject) {
    this.subject = subject;
}

@DynamoDBAttribute(attributeName = "Message")
public String getMessage() {
    return message;
}

public void setMessage(String message) {
    this.message = message;
}

@DynamoDBAttribute(attributeName = "LastPostedDateTime")
public String getLastPostedDateTime() {
    return lastPostedDateTime;
}

public void setLastPostedDateTime(String lastPostedDateTime) {
    this.lastPostedDateTime = lastPostedDateTime;
}
```

```
}

@DynamoDBAttribute(attributeName = "LastPostedBy")
public String getLastPostedBy() {
    return lastPostedBy;
}

public void setLastPostedBy(String lastPostedBy) {
    this.lastPostedBy = lastPostedBy;
}

@DynamoDBAttribute(attributeName = "Tags")
public Set<String> getTags() {
    return tags;
}

public void setTags(Set<String> tags) {
    this.tags = tags;
}

@DynamoDBAttribute(attributeName = "Answered")
public int getAnswered() {
    return answered;
}

public void setAnswered(int answered) {
    this.answered = answered;
}

@DynamoDBAttribute(attributeName = "Views")
public int getViews() {
    return views;
}

public void setViews(int views) {
    this.views = views;
}

@DynamoDBAttribute(attributeName = "Replies")
public int getReplies() {
    return replies;
}

public void setReplies(int replies) {
```

```
        this.replies = replies;
    }

}

@DynamoDBTable(tableName = "Forum")
public static class Forum {
    private String name;
    private String category;
    private int threads;

    // Partition key
    @DynamoDBHashKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Category")
    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    @DynamoDBAttribute(attributeName = "Threads")
    public int getThreads() {
        return threads;
    }

    public void setThreads(int threads) {
        this.threads = threads;
    }
}
```

Java 2.x: DynamoDB Enhanced Client

The DynamoDB enhanced client is a high-level library that is part of the AWS SDK for Java version 2 (v2). It offers a straightforward way to map client-side classes to DynamoDB tables. You define the relationships between tables and their corresponding model classes in your code. After you define those relationships, you can intuitively perform various create, read, update, or delete (CRUD) operations on tables or items in DynamoDB.

For more information on how you can use the enhanced client with DynamoDB, see [Using the DynamoDB Enhanced Client in the AWS SDK for Java 2.x](#).

.NET: Document model

The AWS SDK for .NET provides document model classes that wrap some of the low-level Amazon DynamoDB operations, further simplifying your coding. In the document model, the primary classes are `Table` and `Document`. The `Table` class provides data operation methods such as `PutItem`, `GetItem`, and `DeleteItem`. It also provides the `Query` and the `Scan` methods. The `Document` class represents a single item in a table.

The preceding document model classes are available in the `Amazon.DynamoDBv2.DocumentModel` namespace.

 **Note**

You can't use the document model classes to create, update, and delete tables. However, the document model does support most common data operations.

Topics

- [Supported data types](#)
- [Working with items in DynamoDB using the AWS SDK for .NET document model](#)
- [Example: CRUD operations using the AWS SDK for .NET document model](#)
- [Example: Batch operations using the AWS SDK for .NET document model API](#)
- [Working with tables in DynamoDB using the AWS SDK for .NET document model](#)

Supported data types

The document model supports a set of primitive .NET data types and collections data types. The model supports the following primitive data types.

- bool
- byte
- char
- DateTime
- decimal
- double
- float
- Guid
- Int16
- Int32
- Int64
- SByte
- string
- UInt16
- UInt32
- UInt64

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

.NET primitive type	DynamoDB type
All number types	N (number type)
All string types	S (string type)
MemoryStream, byte[]	B (binary type)
bool	N (number type). 0 represents false and 1 represents true.

.NET primitive type	DynamoDB type
DateTime	S (string type). The DateTime values are stored as ISO-8601 formatted strings.
Guid	S (string type).
Collection types (List, HashSet, and array)	BS (binary set) type, SS (string set) type, and NS (number set) type

AWS SDK for .NET defines types for mapping DynamoDB's Boolean, null, list and map types to .NET document model API:

- Use `DynamoDBBool` for Boolean type.
- Use `DynamoDBNull` for null type.
- Use `DynamoDBList` for list type.
- Use `Document` for map type.

 **Note**

- Empty binary values are supported.
- Reading of empty string values is supported. Empty string attribute values are supported within attribute values of string Set type while writing to DynamoDB. Empty string attribute values of string type and empty string values contained within List or Map type are dropped from write requests

Working with items in DynamoDB using the AWS SDK for .NET document model

The following code examples demonstrate how to perform a variety of operations with the AWS SDK for .NET document model. You can use these examples to perform CRUD, batch, and transaction operations.

Topics

- [Putting an item - Table.PutItem method](#)

- [Specifying optional parameters](#)
- [Getting an item - Table.GetItem](#)
- [Deleting an item - Table.DeleteItem](#)
- [Updating an item - Table.UpdateItem](#)
- [Batch write - putting and deleting multiple items](#)

To perform data operations using the document model, you must first call the `Table.LoadTable` method, which creates an instance of the `Table` class that represents a specific table. The following C# example creates a `Table` object that represents the `ProductCatalog` table in Amazon DynamoDB.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
```

Note

In general, you use the `LoadTable` method once at the beginning of your application because it makes a `DescribeTable` call that adds to the round trip to DynamoDB.

You can then use the `Table` object to perform various data operations. Each data operation has two types of overloads: One takes the minimum required parameters and the other takes optional, operation-specific configuration information. For example, to retrieve an item, you must provide the table's primary key value, in which case you can use the following `GetItem` overload.

Example

```
// Get the item from a table that has a primary key that is composed of only a
// partition key.
Table.GetItem(Primitive partitionKey);
// Get the item from a table whose primary key is composed of both a partition key and
// sort key.
Table.GetItem(Primitive partitionKey, Primitive sortKey);
```

You also can pass optional parameters to these methods. For example, the preceding `GetItem` returns the entire item including all its attributes. You can optionally specify a list of attributes to

retrieve. In this case, you use the following `GetItem` overload that takes in the operation-specific configuration object parameter.

Example

```
// Configuration object that specifies optional parameters.  
GetItemOperationConfig config = new GetItemOperationConfig()  
{  
    AttributesToGet = new List<string>() { "Id", "Title" },  
};  
// Pass in the configuration to the GetItem method.  
// 1. Table that has only a partition key as primary key.  
Table.GetItem(Primitive partitionKey, GetItemOperationConfig config);  
// 2. Table that has both a partition key and a sort key.  
Table.GetItem(Primitive partitionKey, Primitive sortKey, GetItemOperationConfig  
config);
```

You can use the configuration object to specify several optional parameters such as request a specific list of attributes or specify the page size (number of items per page). Each data operation method has its own configuration class. For example, you can use the `GetItemOperationConfig` class to provide options for the `GetItem` operation. You can use the `PutItemOperationConfig` class to provide optional parameters for the `PutItem` operation.

The following sections discuss each of the data operations that are supported by the `Table` class.

Putting an item - `Table.PutItem` method

The `PutItem` method uploads the input `Document` instance to the table. If an item that has a primary key that is specified in the input `Document` exists in the table, the `PutItem` operation replaces the entire existing item. The new item is identical to the `Document` object that you provided to the `PutItem` method. If your original item had any extra attributes, they are no longer present in the new item.

The following are the steps to put a new item into a table using the AWS SDK for .NET document model.

1. Run the `Table.LoadTable` method that provides the table name in which you want to put an item.
2. Create a `Document` object that has a list of attribute names and their values.
3. Run `Table.PutItem` by providing the `Document` instance as a parameter.

The following C# code example demonstrates the preceding tasks. The example uploads an item to the ProductCatalog table.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 101;
book["Title"] = "Book 101 Title";
book["ISBN"] = "11-11-11-11";
book["Authors"] = new List<string> { "Author 1", "Author 2" };
book["InStock"] = new DynamoDBBool(true);
book["QuantityOnHand"] = new DynamoDBNull();

table.PutItem(book);
```

In the preceding example, the Document instance creates an item that has Number, String, String Set, Boolean, and Null attributes. (Null is used to indicate that the *QuantityOnHand* for this product is unknown.) For Boolean and Null, use the constructor methods DynamoDBBool and DynamoDBNull.

In DynamoDB, the List and Map data types can contain elements composed of other data types. Here is how to map these data types to the document model API:

- List — use the DynamoDBList constructor.
- Map — use the Document constructor.

You can modify the preceding example to add a List attribute to the item. To do this, use a DynamoDBList constructor, as shown in the following code example.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 101;

/*other attributes omitted for brevity...*/

var relatedItems = new DynamoDBList();
```

```
relatedItems.Add(341);
relatedItems.Add(472);
relatedItems.Add(649);
book.Add("RelatedItems", relatedItems);

table.PutItem(book);
```

To add a Map attribute to the book, you define another Document. The following code example illustrates how to do this.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 101;

/*other attributes omitted for brevity...*/

var pictures = new Document();
pictures.Add("FrontView", "http://example.com/products/101_front.jpg" );
pictures.Add("RearView", "http://example.com/products/101_rear.jpg" );

book.Add("Pictures", pictures);

table.PutItem(book);
```

These examples are based on the item shown in [Specifying item attributes when using expressions](#). The document model lets you create complex nested attributes, such as the ProductReviews attribute shown in the case study.

Specifying optional parameters

You can configure optional parameters for the PutItem operation by adding the PutItemOperationConfig parameter. For a complete list of optional parameters, see [PutItem](#). The following C# code example puts an item in the ProductCatalog table. It specifies the following optional parameter:

- The ConditionalExpression parameter to make this a conditional put request. The example creates an expression that specifies the ISBN attribute must have a specific value that has to be present in the item that you are replacing.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 555;
book["Title"] = "Book 555 Title";
book["Price"] = "25.00";
book["ISBN"] = "55-55-55-55";
book["Name"] = "Item 1 updated";
book["Authors"] = new List<string> { "Author x", "Author y" };
book["InStock"] = new DynamoDBBool(true);
book["QuantityOnHand"] = new DynamoDBNull();

// Create a condition expression for the optional conditional put operation.
Expression expr = new Expression();
expr.ExpressionStatement = "ISBN = :val";
expr.ExpressionAttributeValues[":val"] = "55-55-55-55";

PutItemOperationConfig config = new PutItemOperationConfig()
{
    // Optional parameter.
    ConditionalExpression = expr
};

table.PutItem(book, config);
```

Getting an item - Table.GetItem

The GetItem operation retrieves an item as a Document instance. You must provide the primary key of the item that you want to retrieve as shown in the following C# code example.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
Document document = table.GetItem(101); // Primary key 101.
```

The GetItem operation returns all the attributes of the item and performs an eventually consistent read (see [Read consistency](#)) by default.

Specifying optional parameters

You can configure additional options for the `GetItem` operation by adding the `GetItemOperationConfig` parameter. For a complete list of optional parameters, see [GetItem](#). The following C# code example retrieves an item from the `ProductCatalog` table. It specifies the `GetItemOperationConfig` to provide the following optional parameters:

- The `AttributesToGet` parameter to retrieve only the specified attributes.
- The `ConsistentRead` parameter to request the latest values for all the specified attributes. To learn more about data consistency, see [Read consistency](#).

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

GetItemOperationConfig config = new GetItemOperationConfig()
{
    AttributesToGet = new List<string>() { "Id", "Title", "Authors", "InStock",
    "QuantityOnHand" },
    ConsistentRead = true
};
Document doc = table.GetItem(101, config);
```

When you retrieve an item using the document model API, you can access individual elements within the `Document` object is returned, as shown in the following example.

Example

```
int id = doc["Id"].AsInt();
string title = doc["Title"].AsString();
List<string> authors = doc["Authors"].AsListOfString();
bool inStock = doc["InStock"].AsBoolean();
DynamoDBNull quantityOnHand = doc["QuantityOnHand"].AsDynamoDBNull();
```

For attributes that are of type `List` or `Map`, here is how to map these attributes to the document model API:

- `List` — Use the `AsDynamoDBList` method.
- `Map` — Use the `AsDocument` method.

The following code example shows how to retrieve a List (RelatedItems) and a Map (Pictures) from the Document object:

Example

```
DynamoDBList relatedItems = doc["RelatedItems"].AsDynamoDBList();  
  
Document pictures = doc["Pictures"].AsDocument();
```

Deleting an item - Table.DeleteItem

The DeleteItem operation deletes an item from a table. You can pass the item's primary key as a parameter. Or, if you've already read an item and have the corresponding Document object, you can pass it as a parameter to the DeleteItem method, as shown in the following C# code example.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");  
  
// Retrieve a book (a Document instance)  
Document document = table.GetItem(111);  
  
// 1) Delete using the Document instance.  
table.DeleteItem(document);  
  
// 2) Delete using the primary key.  
int partitionKey = 222;  
table.DeleteItem(partitionKey)
```

Specifying optional parameters

You can configure additional options for the Delete operation by adding the DeleteItemOperationConfig parameter. For a complete list of optional parameters, see [DeleteTable](#). The following C# code example specifies the two following optional parameters:

- The ConditionalExpression parameter to ensure that the book item being deleted has a specific value for the ISBN attribute.
- The ReturnValue parameter to request that the Delete method return the item that it deleted.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
int partitionKey = 111;

Expression expr = new Expression();
expr.ExpressionStatement = "ISBN = :val";
expr.ExpressionAttributeValues[":val"] = "11-11-11-11";

// Specify optional parameters for Delete operation.
DeleteItemOperationConfig config = new DeleteItemOperationConfig
{
    ConditionalExpression = expr,
    ReturnValues = ReturnValues.AllOldAttributes // This is the only supported value
    when using the document model.
};

// Delete the book.
Document d = table.DeleteItem(partitionKey, config);
```

Updating an item - Table.UpdateItem

The `UpdateItem` operation updates an existing item if it is present. If the item that has the specified primary key is not found, the `UpdateItem` operation adds a new item.

You can use the `UpdateItem` operation to update existing attribute values, add new attributes to the existing collection, or delete attributes from the existing collection. You provide these updates by creating a `Document` instance that describes the updates that you want to perform.

The `UpdateItem` action uses the following guidelines:

- If the item does not exist, `UpdateItem` adds a new item using the primary key that is specified in the input.
- If the item exists, `UpdateItem` applies the updates as follows:
 - Replaces the existing attribute values with the values in the update.
 - If an attribute that you provide in the input does not exist, it adds a new attribute to the item.
 - If the input attribute value is null, it deletes the attributes, if it is present.

Note

This midlevel `UpdateItem` operation does not support the Add action (see [UpdateItem](#)) that is supported by the underlying DynamoDB operation.

Note

The `PutItem` operation ([Putting an item - Table.PutItem method](#)) can also perform an update. If you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. If there are attributes in the existing item and those attributes are not specified on the Document that is being put, the `PutItem` operation deletes those attributes. However, `UpdateItem` only updates the specified input attributes. Any other existing attributes of that item remain unchanged.

The following are the steps to update an item using the AWS SDK for .NET document model:

1. Run the `Table.LoadTable` method by providing the name of the table in which you want to perform the update operation.
2. Create a `Document` instance by providing all the updates that you want to perform.

To delete an existing attribute, specify the attribute value as null.

3. Call the `Table.UpdateItem` method and provide the `Document` instance as an input parameter.

You must provide the primary key either in the `Document` instance or explicitly as a parameter.

The following C# code example demonstrates the preceding tasks. The code example updates an item in the Book table. The `UpdateItem` operation updates the existing `Authors` attribute, deletes the `PageCount` attribute, and adds a new `XYZ` attribute. The `Document` instance includes the primary key of the book to update.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
```

```
var book = new Document();

// Set the attributes that you wish to update.
book["Id"] = 111; // Primary key.
// Replace the authors attribute.
book["Authors"] = new List<string> { "Author x", "Author y" };
// Add a new attribute.
book["XYZ"] = 12345;
// Delete the existing PageCount attribute.
book["PageCount"] = null;

table.Update(book);
```

Specifying optional parameters

You can configure additional options for the `UpdateItem` operation by adding the `UpdateItemOperationConfig` parameter. For a complete list of optional parameters, see [UpdateItem](#).

The following C# code example updates a book item price to 25. It specifies the two following optional parameters:

- The `ConditionalExpression` parameter that identifies the `Price` attribute with value 20 that you expect to be present.
- The `ReturnValues` parameter to request the `UpdateItem` operation to return the item that is updated.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
string partitionKey = "111";

var book = new Document();
book["Id"] = partitionKey;
book["Price"] = 25;

Expression expr = new Expression();
expr.ExpressionStatement = "Price = :val";
expr.ExpressionAttributeValues[":val"] = "20";

UpdateItemOperationConfig config = new UpdateItemOperationConfig()
```

```
{  
    ConditionalExpression = expr,  
    ReturnValues = ReturnValues.AllOldAttributes  
};  
  
Document d1 = table.Update(book, config);
```

Batch write - putting and deleting multiple items

Batch write refers to putting and deleting multiple items in a batch. The operation enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to put or delete multiple items from a table using the AWS SDK for .NET document model API.

1. Create a `Table` object by executing the `Table.LoadTable` method by providing the name of the table in which you want to perform the batch operation.
2. Run the `createBatchWrite` method on the table instance you created in the preceding step and create a `DocumentBatchWrite` object.
3. Use the `DocumentBatchWrite` object methods to specify the documents that you want to upload or delete.
4. Call the `DocumentBatchWrite.Execute` method to run the batch operation.

When using the document model API, you can specify any number of operations in a batch. However, DynamoDB limits the number of operations in a batch and the total size of the batch in a batch operation. For more information about the specific limits, see [BatchWriteItem](#). If the document model API detects that your batch write request exceeded the number of allowed write requests, or the HTTP payload size of a batch exceeded the limit allowed by `BatchWriteItem`, it breaks the batch into several smaller batches. Additionally, if a response to a batch write returns unprocessed items, the document model API automatically sends another batch request with those unprocessed items.

The following C# code example demonstrates the preceding steps. The example uses batch write operation to perform two writes; upload a book item and delete another book item.

```
Table productCatalog = Table.LoadTable(client, "ProductCatalog");  
var batchWrite = productCatalog.CreateBatchWrite();  
  
var book1 = new Document();  
book1["Id"] = 902;
```

```
book1["Title"] = "My book1 in batch write using .NET document model";
book1["Price"] = 10;
book1["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };
book1["InStock"] = new DynamoDBBool(true);
book1["QuantityOnHand"] = 5;

batchWrite.AddDocumentToPut(book1);
// specify delete item using overload that takes PK.
batchWrite.AddKeyToDelete(12345);

batchWrite.Execute();
```

For a working example, see [Example: Batch operations using the AWS SDK for .NET document model API](#).

You can use the `batchWrite` operation to perform put and delete operations on multiple tables. The following are the steps to put or delete multiple items from multiple tables using the AWS SDK for .NET document model.

1. You create a `DocumentBatchWrite` instance for each table in which you want to put or delete multiple items, as described in the preceding procedure.
2. Create an instance of the `MultiTableDocumentBatchWrite` and add the individual `DocumentBatchWrite` objects to it.
3. Run the `MultiTableDocumentBatchWrite.Execute` method.

The following C# code example demonstrates the preceding steps. The example uses the batch write operation to perform the following write operations:

- Put a new item in the `Forum` table item.
- Put an item in the `Thread` table and delete an item from the same table.

```
// 1. Specify item to add in the Forum table.
Table forum = Table.LoadTable(client, "Forum");
var forumBatchWrite = forum.CreateBatchWrite();

var forum1 = new Document();
forum1["Name"] = "Test BatchWrite Forum";
forum1["Threads"] = 0;
```

```
forumBatchWrite.AddDocumentToPut(forum1);

// 2a. Specify item to add in the Thread table.
Table thread = Table.LoadTable(client, "Thread");
var threadBatchWrite = thread.CreateBatchWrite();

var thread1 = new Document();
thread1["ForumName"] = "Amazon S3 forum";
thread1["Subject"] = "My sample question";
thread1["Message"] = "Message text";
thread1["KeywordTags"] = new List<string>{ "Amazon S3", "Bucket" };
threadBatchWrite.AddDocumentToPut(thread1);

// 2b. Specify item to delete from the Thread table.
threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

// 3. Create multi-table batch.
var superBatch = new MultiTableDocumentBatchWrite();
superBatch.AddBatch(forumBatchWrite);
superBatch.AddBatch(threadBatchWrite);

superBatch.Execute();
```

Example: CRUD operations using the AWS SDK for .NET document model

The following C# code example performs the following actions:

- Creates a book item in the ProductCatalog table.
- Retrieves the book item.
- Updates the book item. The code example shows a normal update that adds new attributes and updates existing attributes. It also shows a conditional update that updates the book price only if the existing price value is as specified in the code.
- Deletes the book item.

For step-by-step instructions to test the following example, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class MidlevelItemCRUD
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "ProductCatalog";
        // The sample uses the following id PK value to add book item.
        private static int sampleBookId = 555;

        static void Main(string[] args)
        {
            try
            {
                Table productCatalog = Table.LoadTable(client, tableName);
                CreateBookItem(productCatalog);
                RetrieveBook(productCatalog);
                // Couple of sample updates.
                UpdateMultipleAttributes(productCatalog);
                UpdateBookPriceConditionally(productCatalog);

                // Delete.
                DeleteBook(productCatalog);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        // Creates a sample book item.
        private static void CreateBookItem(Table productCatalog)
        {
            Console.WriteLine("\n*** Executing CreateBookItem() ***");
            var book = new Document();
            book["Id"] = sampleBookId;
            book["Title"] = "Book " + sampleBookId;
            book["Price"] = 19.99;
            book["ISBN"] = "111-1111111111";
        }
    }
}
```

```
book["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };
book["PageCount"] = 500;
book["Dimensions"] = "8.5x11x.5";
book["InPublication"] = new DynamoDBBool(true);
book["InStock"] = new DynamoDBBool(false);
book["QuantityOnHand"] = 0;

productCatalog.PutItem(book);
}

private static void RetrieveBook(Table productCatalog)
{
    Console.WriteLine("\n*** Executing RetrieveBook() ***");
    // Optional configuration.
    GetItemOperationConfig config = new GetItemOperationConfig
    {
        AttributesToGet = new List<string> { "Id", "ISBN", "Title", "Authors",
"Price" },
        ConsistentRead = true
    };
    Document document = productCatalog.GetItem(sampleBookId, config);
    Console.WriteLine("RetrieveBook: Printing book retrieved...");
    PrintDocument(document);
}

private static void UpdateMultipleAttributes(Table productCatalog)
{
    Console.WriteLine("\n*** Executing UpdateMultipleAttributes() ***");
    Console.WriteLine("\nUpdating multiple attributes....");
    int partitionKey = sampleBookId;

    var book = new Document();
    book["Id"] = partitionKey;
    // List of attribute updates.
    // The following replaces the existing authors list.
    book["Authors"] = new List<string> { "Author x", "Author y" };
    book["newAttribute"] = "New Value";
    book["ISBN"] = null; // Remove it.

    // Optional parameters.
    UpdateItemOperationConfig config = new UpdateItemOperationConfig
    {
        // Get updated item in response.
        ReturnValue = ReturnValue.AllNewAttributes
```

```
};

Document updatedBook = productCatalog.UpdateItem(book, config);
Console.WriteLine("UpdateMultipleAttributes: Printing item after
updates ...");
PrintDocument(updatedBook);
}

private static void UpdateBookPriceConditionally(Table productCatalog)
{
    Console.WriteLine("\n*** Executing UpdateBookPriceConditionally() ***");

    int partitionKey = sampleBookId;

    var book = new Document();
    book["Id"] = partitionKey;
    book["Price"] = 29.99;

    // For conditional price update, creating a condition expression.
    Expression expr = new Expression();
    expr.ExpressionStatement = "Price = :val";
    expr.ExpressionAttributeValues[":val"] = 19.00;

    // Optional parameters.
    UpdateItemOperationConfig config = new UpdateItemOperationConfig
    {
        ConditionalExpression = expr,
        ReturnValues = ReturnValues.AllNewAttributes
    };
    Document updatedBook = productCatalog.UpdateItem(book, config);
    Console.WriteLine("UpdateBookPriceConditionally: Printing item whose price
was conditionally updated");
    PrintDocument(updatedBook);
}

private static void DeleteBook(Table productCatalog)
{
    Console.WriteLine("\n*** Executing DeleteBook() ***");
    // Optional configuration.
    DeleteItemOperationConfig config = new DeleteItemOperationConfig
    {
        // Return the deleted item.
        ReturnValues = ReturnValues.AllOldAttributes
    };
    Document document = productCatalog.DeleteItem(sampleBookId, config);
```

```
        Console.WriteLine("DeleteBook: Printing deleted just deleted...");  
        PrintDocument(document);  
    }  
  
    private static void PrintDocument(Document updatedDocument)  
{  
        foreach (var attribute in updatedDocument.GetAttributeNames())  
        {  
            string stringValue = null;  
            var value = updatedDocument[attribute];  
            if (value is Primitive)  
                stringValue = value.AsPrimitive().Value.ToString();  
            else if (value is PrimitiveList)  
                stringValue = string.Join(", ", (from primitive  
                                                in value.AsPrimitiveList().Entries  
                                                select primitive.Value).ToArray());  
            Console.WriteLine("{0} - {1}", attribute, stringValue);  
        }  
    }  
}
```

Example: Batch operations using the AWS SDK for .NET document model API

Topics

- [Example: Batch write using the AWS SDK for .NET document model](#)

Example: Batch write using the AWS SDK for .NET document model

The following C# code example illustrates single table and multi-table batch write operations. The example performs the following tasks:

- Illustrates a single table batch write. It adds two items to the ProductCatalog table.
- Illustrates a multi-table batch write. It adds an item to both the Forum and Thread tables and deletes an item from the Thread table.

If you followed the steps in [Creating tables and loading data for code examples in DynamoDB](#), you already have the ProductCatalog, Forum, and Thread tables created. You can also create these sample tables programmatically. For more information, see [Creating example tables and uploading](#)

[data using the AWS SDK for .NET](#). For step-by-step instructions for testing the following example, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class MidLevelBatchWriteItem
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        static void Main(string[] args)
        {
            try
            {
                SingleTableBatchWrite();
                MultiTableBatchWrite();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void SingleTableBatchWrite()
        {
            Table productCatalog = Table.LoadTable(client, "ProductCatalog");
            var batchWrite = productCatalog.CreateBatchWrite();

            var book1 = new Document();
            book1["Id"] = 902;
            book1["Title"] = "My book1 in batch write using .NET helper classes";
            book1["ISBN"] = "902-11-11111";
            book1["Price"] = 10;
            book1["ProductCategory"] = "Book";
            book1["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };
            book1["Dimensions"] = "8.5x11x.5";
        }
    }
}
```

```
        book1["InStock"] = new DynamoDBBool(true);
        book1["QuantityOnHand"] = new DynamoDBNull(); //Quantity is unknown at this
time

        batchWrite.AddDocumentToPut(book1);
        // Specify delete item using overload that takes PK.
        batchWrite.AddKeyToDelete(12345);
        Console.WriteLine("Performing batch write in SingleTableBatchWrite()");
        batchWrite.Execute();
    }

private static void MultiTableBatchWrite()
{
    // 1. Specify item to add in the Forum table.
    Table forum = Table.LoadTable(client, "Forum");
    var forumBatchWrite = forum.CreateBatchWrite();

    var forum1 = new Document();
    forum1["Name"] = "Test BatchWrite Forum";
    forum1["Threads"] = 0;
    forumBatchWrite.AddDocumentToPut(forum1);

    // 2a. Specify item to add in the Thread table.
    Table thread = Table.LoadTable(client, "Thread");
    var threadBatchWrite = thread.CreateBatchWrite();

    var thread1 = new Document();
    thread1["ForumName"] = "S3 forum";
    thread1["Subject"] = "My sample question";
    thread1["Message"] = "Message text";
    thread1["KeywordTags"] = new List<string> { "S3", "Bucket" };
    threadBatchWrite.AddDocumentToPut(thread1);

    // 2b. Specify item to delete from the Thread table.
    threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

    // 3. Create multi-table batch.
    var superBatch = new MultiTableDocumentBatchWrite();
    superBatch.AddBatch(forumBatchWrite);
    superBatch.AddBatch(threadBatchWrite);
    Console.WriteLine("Performing batch write in MultiTableBatchWrite()");
    superBatch.Execute();
}
```

```
    }  
}
```

Working with tables in DynamoDB using the AWS SDK for .NET document model

Topics

- [Table.Query method in the AWS SDK for .NET](#)
- [Table.Scan method in the AWS SDK for .NET](#)

Table.Query method in the AWS SDK for .NET

The Query method enables you to query your tables. You can only query the tables that have a composite primary key (partition key and sort key). If your table's primary key is made of only a partition key, then the Query operation is not supported. By default, Query internally performs queries that are eventually consistent. To learn about the consistency model, see [Read consistency](#).

The Query method provides two overloads. The minimum required parameters to the Query method are a partition key value and a sort key filter. You can use the following overload to provide these minimum required parameters.

Example

```
Query(Primitive partitionKey, RangeFilter Filter);
```

For example, the following C# code queries for all forum replies that were posted in the last 15 days.

Example

```
string tableName = "Reply";  
Table table = Table.LoadTable(client, tableName);  
  
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);  
RangeFilter filter = new RangeFilter(QueryOperator.GreaterThan, twoWeeksAgoDate);  
Search search = table.Query("DynamoDB Thread 2", filter);
```

This creates a Search object. You can now call the Search.GetNextSet method iteratively to retrieve one page of results at a time, as shown in the following C# code example. The code prints the attribute values for each item that the query returns.

Example

```
List<Document> documentSet = new List<Document>();
do
{
    documentSet = search.GetNextSet();
    foreach (var document in documentSet)
        PrintDocument(document);
} while (!search.IsDone());

private static void PrintDocument(Document document)
{
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
        if (value is Primitive)
            stringValue = value.AsPrimitive().Value;
        else if (value is PrimitiveList)
            stringValue = string.Join(", ", (from primitive
                                              in value.AsPrimitiveList().Entries
                                              select primitive.Value).ToArray());
        Console.WriteLine("{0} - {1}", attribute, stringValue);
    }
}
```

Specifying optional parameters

You can also specify optional parameters for `Query`, such as specifying a list of attributes to retrieve, strongly consistent reads, page size, and the number of items returned per page. For a complete list of parameters, see [Query](#). To specify optional parameters, you must use the following overload in which you provide the `QueryOperationConfig` object.

Example

```
Query(QueryOperationConfig config);
```

Assume that you want to run the query in the preceding example (retrieve forum replies posted in the last 15 days). However, assume that you want to provide optional query parameters to retrieve

only specific attributes and also request a strongly consistent read. The following C# code example constructs the request using the `QueryOperationConfig` object.

Example

```
Table table = Table.LoadTable(client, "Reply");
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
QueryOperationConfig config = new QueryOperationConfig()
{
    HashKey = "DynamoDB Thread 2", //Partition key
    AttributesToGet = new List<string>
    {
        "Subject", "ReplyDateTime", "PostedBy"
    },
    ConsistentRead = true,
    Filter = new RangeFilter(QueryOperator.GreaterThan, twoWeeksAgoDate)
};

Search search = table.Query(config);
```

Example: Query using the Table.Query method

The following C# code example uses the `Table.Query` method to run the following sample queries.

- The following queries are run against the `Reply` table.
 - Find forum thread replies that were posted in the last 15 days.

This query is run twice. In the first `Table.Query` call, the example provides only the required query parameters. In the second `Table.Query` call, you provide optional query parameters to request a strongly consistent read and a list of attributes to retrieve.

- Find forum thread replies posted during a period of time.

This query uses the `Between` query operator to find replies posted in between two dates.

- Get a product from the `ProductCatalog` table.

Because the `ProductCatalog` table has a primary key that is only a partition key, you can only get items; you cannot query the table. The example retrieves a specific product item using the item Id.

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class MidLevelQueryAndScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                // Query examples.
                Table replyTable = Table.LoadTable(client, "Reply");
                string forumName = "Amazon DynamoDB";
                string threadSubject = "DynamoDB Thread 2";
                FindRepliesInLast15Days(replyTable, forumName, threadSubject);
                FindRepliesInLast15DaysWithConfig(replyTable, forumName,
threadSubject);
                FindRepliesPostedWithinTimePeriod(replyTable, forumName,
threadSubject);

                // Get Example.
                Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
                int productId = 101;
                GetProduct(productCatalogTable, productId);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void GetProduct(Table tableName, int productId)
```

```
{  
    Console.WriteLine("**** Executing GetProduct() ****");  
    Document productDocument = tableName.GetItem(productId);  
    if (productDocument != null)  
    {  
        PrintDocument(productDocument);  
    }  
    else  
    {  
        Console.WriteLine("Error: product " + productId + " does not exist");  
    }  
  
    private static void FindRepliesInLast15Days(Table table, string forumName,  
string threadSubject)  
    {  
        string Attribute = forumName + "#" + threadSubject;  
  
        DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);  
        QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal,  
partitionKey);  
        filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,  
twoWeeksAgoDate);  
  
        // Use Query overloads that takes the minimum required query parameters.  
        Search search = table.Query(filter);  
  
        List<Document> documentSet = new List<Document>();  
        do  
        {  
            documentSet = search.GetNextSet();  
            Console.WriteLine("\nFindRepliesInLast15Days: printing .....");  
            foreach (var document in documentSet)  
                PrintDocument(document);  
        } while (!search.IsDone);  
    }  
  
    private static void FindRepliesPostedWithinTimePeriod(Table table, string  
forumName, string threadSubject)  
    {  
        DateTime startDate = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0, 0));  
        DateTime endDate = DateTime.UtcNow.Subtract(new TimeSpan(1, 0, 0, 0));  
    }
```

```
        QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
"#" + threadSubject);
        filter.AddCondition("ReplyDateTime", QueryOperator.Between, startDate,
endDate);

        QueryOperationConfig config = new QueryOperationConfig()
{
    Limit = 2, // 2 items/page.
    Select = SelectValues.SpecificAttributes,
    AttributesToGet = new List<string> { "Message",
                                         "ReplyDateTime",
                                         "PostedBy" },
    ConsistentRead = true,
    Filter = filter
};

Search search = table.Query(config);

List<Document> documentList = new List<Document>();

do
{
    documentList = search.GetNextSet();
    Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: printing
replies posted within dates: {0} and {1} ....", startDate, endDate);
    foreach (var document in documentList)
    {
        PrintDocument(document);
    }
} while (!search.IsDone());
}

private static void FindRepliesInLast15DaysWithConfig(Table table, string
forumName, string threadName)
{
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
    QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
"#" + threadName);
    filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);
    // You are specifying optional parameters so use QueryOperationConfig.
    QueryOperationConfig config = new QueryOperationConfig()
{
    Filter = filter,
```

```
// Optional parameters.
Select = SelectValues.SpecificAttributes,
AttributesToGet = new List<string> { "Message", "ReplyDateTime",
                                         "PostedBy" },
ConsistentRead = true
};

Search search = table.Query(config);

List<Document> documentSet = new List<Document>();
do
{
    documentSet = search.GetNextSet();
    Console.WriteLine("\nFindRepliesInLast15DaysWithConfig:
printing .....");
    foreach (var document in documentSet)
        PrintDocument(document);
} while (!search.IsDone());
}

private static void PrintDocument(Document document)
{
    // count++;
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
        if (value is Primitive)
            stringValue = value.AsPrimitive().Value.ToString();
        else if (value is PrimitiveList)
            stringValue = string.Join(", ", (from primitive
                                              in value.AsPrimitiveList().Entries
                                              select primitive.Value).ToArray());
        Console.WriteLine("{0} - {1}", attribute, stringValue);
    }
}
}
```

Table.Scan method in the AWS SDK for .NET

The Scan method performs a full table scan. It provides two overloads. The only parameter required by the Scan method is the scan filter, which you can provide using the following overload.

Example

```
Scan(ScanFilter filter);
```

For example, assume that you maintain a table of forum threads tracking information such as thread subject (primary), the related message, forum Id to which the thread belongs, Tags, and other information. Assume that the subject is the primary key.

Example

```
Thread(Subject, Message, ForumId, Tags, LastPostedDateTime, .... )
```

This is a simplified version of forums and threads that you see on AWS forums (see [Discussion forums](#)). The following C# code example queries all threads in a specific forum (ForumId = 101) that are tagged "sortkey". Because the ForumId is not a primary key, the example scans the table. The ScanFilter includes two conditions. The query returns all the threads that satisfy both of the conditions.

Example

```
string tableName = "Thread";
Table ThreadTable = Table.LoadTable(client, tableName);

ScanFilter scanFilter = new ScanFilter();
scanFilter.AddCondition("ForumId", ScanOperator.Equal, 101);
scanFilter.AddCondition("Tags", ScanOperator.Contains, "sortkey");

Search search = ThreadTable.Scan(scanFilter);
```

Specifying optional parameters

You also can specify optional parameters to Scan, such as a specific list of attributes to retrieve or whether to perform a strongly consistent read. To specify optional parameters, you must create a ScanOperationConfig object that includes both the required and optional parameters and use the following overload.

Example

```
Scan(ScanOperationConfig config);
```

The following C# code example runs the same preceding query (find forum threads in which the ForumId is 101 and the Tag attribute contains the "sortkey" keyword). Assume that you want to add an optional parameter to retrieve only a specific attribute list. In this case, you must create a ScanOperationConfig object by providing all the parameters, required and optional parameters, as shown in the following code example.

Example

```
string tableName = "Thread";
Table ThreadTable = Table.LoadTable(client, tableName);

ScanFilter scanFilter = new ScanFilter();
scanFilter.AddCondition("ForumId", ScanOperator.Equal, forumId);
scanFilter.AddCondition("Tags", ScanOperator.Contains, "sortkey");

ScanOperationConfig config = new ScanOperationConfig()
{
    AttributesToGet = new List<string> { "Subject", "Message" } ,
    Filter = scanFilter
};

Search search = ThreadTable.Scan(config);
```

Example: Scan using the Table.Scan method

The Scan operation performs a full table scan making it a potentially expensive operation. You should use queries instead. However, there are times when you might need to run a scan against a table. For example, you might have a data entry error in the product pricing, and you must scan the table as shown in the following C# code example. The example scans the ProductCatalog table to find products for which the price value is less than 0. The example illustrates the use of the two Table.Scan overloads.

- Table.Scan that takes the ScanFilter object as a parameter.

You can pass the ScanFilter parameter when passing in only the required parameters.

- Table.Scan that takes the ScanOperationConfig object as a parameter.

You must use the `ScanOperationConfig` parameter if you want to pass any optional parameters to the `Scan` method.

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;

namespace com.amazonaws.codesamples
{
    class MidLevelScanOnly
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
            // Scan example.
            FindProductsWithNegativePrice(productCatalogTable);
            FindProductsWithNegativePriceWithConfig(productCatalogTable);

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void FindProductsWithNegativePrice(Table productCatalogTable)
        {
            // Assume there is a price error. So we scan to find items priced < 0.
            ScanFilter scanFilter = new ScanFilter();
            scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

            Search search = productCatalogTable.Scan(scanFilter);

            List<Document> documentList = new List<Document>();
            do
            {
                documentList = search.GetNextSet();
                Console.WriteLine("\nFindProductsWithNegativePrice:
printing .....");
```

```
        foreach (var document in documentList)
            PrintDocument(document);
    } while (!search.IsDone);
}

private static void FindProductsWithNegativePriceWithConfig(Table
productCatalogTable)
{
    // Assume there is a price error. So we scan to find items priced < 0.
    ScanFilter scanFilter = new ScanFilter();
    scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

    ScanOperationConfig config = new ScanOperationConfig()
    {
        Filter = scanFilter,
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string> { "Title", "Id" }
    };

    Search search = productCatalogTable.Scan(config);

    List<Document> documentList = new List<Document>();
    do
    {
        documentList = search.GetNextSet();
        Console.WriteLine("\nFindProductsWithNegativePriceWithConfig:
printing .....");
        foreach (var document in documentList)
            PrintDocument(document);
    } while (!search.IsDone);
}

private static void PrintDocument(Document document)
{
    // count++;
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
        if (value is Primitive)
            stringValue = value.AsPrimitive().Value.ToString();
        else if (value is PrimitiveList)
            stringValue = string.Join(", ", (from primitive
```

```
        in value.AsPrimitiveList().Entries
            select primitive.Value).ToArray());
    Console.WriteLine("{0} - {1}", attribute, stringValue);
}
}
}
```

.NET: Object persistence model

Topics

- [DynamoDB attributes](#)
- [DynamoDBContext class](#)
- [Supported data types](#)
- [Optimistic locking using a version number with DynamoDB using the AWS SDK for .NET object persistence model](#)
- [Mapping arbitrary data with DynamoDB using the AWS SDK for .NET object persistence model](#)
- [Batch operations using the AWS SDK for .NET object persistence model](#)
- [Example: CRUD operations using the AWS SDK for .NET object persistence model](#)
- [Example: Batch write operation using the AWS SDK for .NET object persistence model](#)
- [Example: Query and scan in DynamoDB using the AWS SDK for .NET object persistence model](#)

The AWS SDK for .NET provides an object persistence model that enables you to map your client-side classes to Amazon DynamoDB tables. Each object instance then maps to an item in the corresponding tables. To save your client-side objects to the tables, the object persistence model provides the `DynamoDBContext` class, an entry point to DynamoDB. This class provides you a connection to DynamoDB and enables you to access tables, perform various CRUD operations, and run queries.

The object persistence model provides a set of attributes to map client-side classes to tables, and properties/fields to table attributes.

Note

The object persistence model does not provide an API to create, update, or delete tables. It provides only data operations. You can use only the AWS SDK for .NET low-level API

to create, update, and delete tables. For more information, see [Working with DynamoDB tables in .NET](#).

The following example shows how the object persistence model works. It starts with the ProductCatalog table. It has Id as the primary key.

```
ProductCatalog(Id, ...)
```

Suppose that you have a Book class with Title, ISBN, and Authors properties. You can map the Book class to the ProductCatalog table by adding the attributes defined by the object persistence model, as shown in the following C# code example.

Example

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey]
    public int Id { get; set; }

    public string Title { get; set; }
    public int ISBN { get; set; }

    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }

    [DynamoDBIgnore]
    public string CoverPage { get; set; }
}
```

In the preceding example, the DynamoDBTable attribute maps the Book class to the ProductCatalog table.

The object persistence model supports both the explicit and default mapping between class properties and table attributes.

- **Explicit mapping**—To map a property to a primary key, you must use the DynamoDBHashKey and DynamoDBRangeKey object persistence model attributes. Additionally, for the nonprimary key attributes, if a property name in your class and the corresponding table attribute to which

you want to map it are not the same, you must define the mapping by explicitly adding the `DynamoDBProperty` attribute.

In the preceding example, the `Id` property maps to the primary key with the same name, and the `BookAuthors` property maps to the `Authors` attribute in the `ProductCatalog` table.

- **Default mapping**—By default, the object persistence model maps the class properties to the attributes with the same name in the table.

In the preceding example, the properties `Title` and `ISBN` map to the attributes with the same name in the `ProductCatalog` table.

You don't have to map every single class property. You identify these properties by adding the `DynamoDBIgnore` attribute. When you save a `Book` instance to the table, the `DynamoDBContext` does not include the `CoverPage` property. It also does not return this property when you retrieve the book instance.

You can map properties of .NET primitive types such as `int` and `string`. You also can map any arbitrary data types as long as you provide an appropriate converter to map the arbitrary data to one of the DynamoDB types. To learn about mapping arbitrary types, see [Mapping arbitrary data with DynamoDB using the AWS SDK for .NET object persistence model](#).

The object persistence model supports optimistic locking. During an update operation, this ensures that you have the latest copy of the item you are about to update. For more information, see [Optimistic locking using a version number with DynamoDB using the AWS SDK for .NET object persistence model](#).

DynamoDB attributes

This section describes the attributes that the object persistence model offers so that you can map your classes and properties to DynamoDB tables and attributes.

Note

In the following attributes, only `DynamoDBTable` and `DynamoDBHashKey` are required.

DynamoDBGlobalSecondaryIndexHashKey

Maps a class property to the partition key of a global secondary index. Use this attribute if you need to Query a global secondary index.

DynamoDBGlobalSecondaryIndexRangeKey

Maps a class property to the sort key of a global secondary index. Use this attribute if you need to Query a global secondary index and want to refine your results using the index sort key.

DynamoDBHashKey

Maps a class property to the partition key of the table's primary key. The primary key attributes cannot be a collection type.

The following C# code example maps the Book class to the ProductCatalog table, and the Id property to the table's primary key partition key.

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey]
    public int Id { get; set; }

    // Additional properties go here.
}
```

DynamoDBIgnore

Indicates that the associated property should be ignored. If you don't want to save any of your class properties, you can add this attribute to instruct DynamoDBContext not to include this property when saving objects to the table.

DynamoDBLocalSecondaryIndexRangeKey

Maps a class property to the sort key of a local secondary index. Use this attribute if you need to Query a local secondary index and want to refine your results using the index sort key.

DynamoDBProperty

Maps a class property to a table attribute. If the class property maps to a table attribute of the same name, you don't need to specify this attribute. However, if the names are not the same, you

can use this tag to provide the mapping. In the following C# statement, the `DynamoDBProperty` maps the `BookAuthors` property to the `Authors` attribute in the table.

```
[DynamoDBProperty("Authors")]
public List<string> BookAuthors { get; set; }
```

`DynamoDBContext` uses this mapping information to create the `Authors` attribute when saving object data to the corresponding table.

DynamoDBRenamable

Specifies an alternative name for a class property. This is useful if you are writing a custom converter for mapping arbitrary data to a DynamoDB table where the name of a class property is different from a table attribute.

DynamoDBRangeKey

Maps a class property to the sort key of the table's primary key. If the table has a composite primary key (partition key and sort key), you must specify both the `DynamoDBHashKey` and `DynamoDBRangeKey` attributes in your class mapping.

For example, the sample table `Reply` has a primary key made of the `Id` partition key and `Replenishment` sort key. The following C# code example maps the `Reply` class to the `Reply` table. The class definition also indicates that two of its properties map to the primary key.

For more information about sample tables, see [Creating tables and loading data for code examples in DynamoDB](#).

```
[DynamoDBTable("Reply")]
public class Reply
{
    [DynamoDBHashKey]
    public int ThreadId { get; set; }
    [DynamoDBRangeKey]
    public string Replenishment { get; set; }

    // Additional properties go here.
}
```

DynamoDBTable

Identifies the target table in DynamoDB to which the class maps. For example, the following C# code example maps the `Developer` class to the `People` table in DynamoDB.

```
[DynamoDBTable("People")]
public class Developer { ... }
```

This attribute can be inherited or overridden.

- The `DynamoDBTable` attribute can be inherited. In the preceding example, if you add a new class, `Lead`, that inherits from the `Developer` class, it also maps to the `People` table. Both the `Developer` and `Lead` objects are stored in the `People` table.
- The `DynamoDBTable` attribute can also be overridden. In the following C# code example, the `Manager` class inherits from the `Developer` class. However, the explicit addition of the `DynamoDBTable` attribute maps the class to another table (`Managers`).

```
[DynamoDBTable("Managers")]
public class Manager : Developer { ... }
```

You can add the optional parameter, `LowerCamelCaseProperties`, to request DynamoDB to make the first letter of the property name lowercase when storing the objects to a table, as shown in the following C# example.

```
[DynamoDBTable("People", LowerCamelCaseProperties=true)]
public class Developer
{
    string DeveloperName;
    ...
}
```

When saving instances of the `Developer` class, `DynamoDBContext` saves the `DeveloperName` property as the `developerName`.

DynamoDBVersion

Identifies a class property for storing the item version number. For more information about versioning, see [Optimistic locking using a version number with DynamoDB using the AWS SDK for .NET object persistence model](#).

DynamoDBContext class

The `DynamoDBContext` class is the entry point to the Amazon DynamoDB database. It provides a connection to DynamoDB and enables you to access your data in various tables, perform various CRUD operations, and run queries. The `DynamoDBContext` class provides the following methods.

CreateMultiTableBatchGet

Creates a `MultiTableBatchGet` object, composed of multiple individual `BatchGet` objects. Each of these `BatchGet` objects can be used for retrieving items from a single DynamoDB table.

To retrieve the items from tables, use the `ExecuteBatchGet` method, passing the `MultiTableBatchGet` object as a parameter.

CreateMultiTableBatchWrite

Creates a `MultiTableBatchWrite` object, composed of multiple individual `BatchWrite` objects. Each of these `BatchWrite` objects can be used for writing or deleting items in a single DynamoDB table.

To write to tables, use the `ExecuteBatchWrite` method, passing the `MultiTableBatchWrite` object as a parameter.

CreateBatchGet

Creates a `BatchGet` object that you can use to retrieve multiple items from a table. For more information, see [Batch get: Getting multiple items](#).

createBatchWrite

Creates a `BatchWrite` object that you can use to put multiple items into a table, or to delete multiple items from a table. For more information, see [Batch write: Putting and deleting multiple items](#).

Delete

Deletes an item from the table. The method requires the primary key of the item you want to delete. You can provide either the primary key value or a client-side object containing a primary key value as a parameter to this method.

- If you specify a client-side object as a parameter and you have enabled optimistic locking, the delete succeeds only if the client-side and the server-side versions of the object match.

- If you specify only the primary key value as a parameter, the delete succeeds regardless of whether you have enabled optimistic locking or not.

 **Note**

To perform this operation in the background, use the `DeleteAsync` method instead.

Dispose

Disposes of all managed and unmanaged resources.

Executebatchget

Reads data from one or more tables, processing all of the `BatchGet` objects in a `MultiTableBatchGet`.

 **Note**

To perform this operation in the background, use the `ExecuteBatchGetAsync` method instead.

Executebatchwrite

Writes or deletes data in one or more tables, processing all of the `BatchWrite` objects in a `MultiTableBatchWrite`.

 **Note**

To perform this operation in the background, use the `ExecuteBatchWriteAsync` method instead.

FromDocument

Given an instance of a `Document`, the `FromDocument` method returns an instance of a client-side class.

This is helpful if you want to use the document model classes along with the object persistence model to perform any data operations. For more information about the document model classes provided by the AWS SDK for .NET, see [.NET: Document model](#).

Suppose that you have a Document object named doc, that contains a representation of a Forum item. (To see how to construct this object, see the description for the ToDocument method later in this topic.) You can use FromDocument to retrieve the Forum item from the Document, as shown in the following C# code example.

Example

```
forum101 = context.FromDocument<Forum>(101);
```

Note

If your Document object implements the `IEnumerable` interface, you can use the `FromDocuments` method instead. This allows you to iterate over all of the class instances in the Document.

FromQuery

Runs a Query operation, with the query parameters defined in a `QueryOperationConfig` object.

Note

To perform this operation in the background, use the `FromQueryAsync` method instead.

FromScan

Runs a Scan operation, with the scan parameters defined in a `ScanOperationConfig` object.

Note

To perform this operation in the background, use the `FromScanAsync` method instead.

Gettargetable

Retrieves the target table for the specified type. This is useful if you are writing a custom converter for mapping arbitrary data to a DynamoDB table, and you need to determine which table is associated with a custom data type.

Load

Retrieves an item from a table. The method requires only the primary key of the item you want to retrieve.

By default, DynamoDB returns the item with values that are eventually consistent. For information about the eventual consistency model, see [Read consistency](#).

Note

To perform this operation in the background, use the LoadAsync method instead.

Query

Queries a table based on query parameters you provide.

You can query a table only if it has a composite primary key (partition key and sort key). When querying, you must specify a partition key and a condition that applies to the sort key.

Suppose that you have a client-side Reply class mapped to the Reply table in DynamoDB. The following C# code example queries the Reply table to find forum thread replies posted in the past 15 days. The Reply table has a primary key that has the Id partition key and the ReplyDateTime sort key. For more information about the Reply table, see [Creating tables and loading data for code examples in DynamoDB](#).

Example

```
DynamoDBContext context = new DynamoDBContext(client);

string replyId = "DynamoDB#DynamoDB Thread 1"; //Partition key
DateTime twoWeeksAgoDate = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0)); // Date
// to compare.
IEnumerable<Reply> latestReplies = context.Query<Reply>(replyId,
    QueryOperator.GreaterThan, twoWeeksAgoDate);
```

This returns a collection of Reply objects.

The Query method returns a "lazy-loaded" `IEnumerable` collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, you need to iterate only over the `IEnumerable`.

If your table has a simple primary key (partition key), you can't use the Query method. Instead, you can use the Load method and provide the partition key to retrieve the item.

 **Note**

To perform this operation in the background, use the `QueryAsync` method instead.

Save

Saves the specified object to the table. If the primary key specified in the input object doesn't exist in the table, the method adds a new item to the table. If the primary key exists, the method updates the existing item.

If you have optimistic locking configured, the update succeeds only if the client and the server-side versions of the item match. For more information, see [Optimistic locking using a version number with DynamoDB using the AWS SDK for .NET object persistence model](#).

 **Note**

To perform this operation in the background, use the `SaveAsync` method instead.

Scan

Performs an entire table scan.

You can filter scan results by specifying a scan condition. The condition can be evaluated on any attributes in the table. Suppose that you have a client-side class `Book` mapped to the `ProductCatalog` table in DynamoDB. The following C# example scans the table and returns only the book items priced less than 0.

Example

```
IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>()
```

```
        new ScanCondition("Price", ScanOperator.LessThan, price),
        new ScanCondition("ProductCategory", ScanOperator.Equal, "Book")
    );
```

The `Scan` method returns a "lazy-loaded" `IEnumerable` collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, you only need to iterate over the `IEnumerable`.

For performance reasons, you should query your tables and avoid a table scan.

 **Note**

To perform this operation in the background, use the `ScanAsync` method instead.

ToDocument

Returns an instance of the `Document` document model class from your class instance.

This is helpful if you want to use the document model classes along with the object persistence model to perform any data operations. For more information about the document model classes provided by the AWS SDK for .NET, see [.NET: Document model](#).

Suppose that you have a client-side class mapped to the sample `Forum` table. You can then use a `DynamoDBContext` to get an item as a `Document` object from the `Forum` table, as shown in the following C# code example.

Example

```
DynamoDBContext context = new DynamoDBContext(client);

Forum forum101 = context.Load<Forum>(101); // Retrieve a forum by primary key.
Document doc = context.ToDocument<Forum>(forum101);
```

Specifying optional parameters for `DynamoDBContext`

When using the object persistence model, you can specify the following optional parameters for the `DynamoDBContext`.

- **ConsistentRead**—When retrieving data using the `Load`, `Query`, or `Scan` operations, you can add this optional parameter to request the latest values for the data.

- **IgnoreNullValues**—This parameter informs DynamoDBContext to ignore null values on attributes during a Save operation. If this parameter is false (or if it is not set), then a null value is interpreted as a directive to delete the specific attribute.
- **SkipVersionCheck**— This parameter informs DynamoDBContext not to compare versions when saving or deleting an item. For more information about versioning, see [Optimistic locking using a version number with DynamoDB using the AWS SDK for .NET object persistence model](#).
- **TableNamePrefix**— Prefixes all table names with a specific string. If this parameter is null (or if it is not set), then no prefix is used.

The following C# example creates a new DynamoDBContext by specifying two of the preceding optional parameters.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context =
    new DynamoDBContext(client, new DynamoDBContextConfig { ConsistentRead = true,
    SkipVersionCheck = true});
```

DynamoDBContext includes these optional parameters with each request that you send using this context.

Instead of setting these parameters at the DynamoDBContext level, you can specify them for individual operations you run using DynamoDBContext, as shown in the following C# code example. The example loads a specific book item. The Load method of DynamoDBContext specifies the preceding optional parameters.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context = new DynamoDBContext(client);
Book bookItem = context.Load<Book>(productId,new DynamoDBContextConfig{ ConsistentRead
= true, SkipVersionCheck = true });
```

In this case, DynamoDBContext includes these parameters only when sending the Get request.

Supported data types

The object persistence model supports a set of primitive .NET data types, collections, and arbitrary data types. The model supports the following primitive data types.

- `bool`
- `byte`
- `char`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Int16`
- `Int32`
- `Int64`
- `SByte`
- `string`
- `UInt16`
- `UInt32`
- `UInt64`

The object persistence model also supports the .NET collection types. `DynamoDBContext` is able to convert concrete collection types and simple Plain Old CLR Objects (POCOs).

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

.NET primitive type	DynamoDB type
All number types	N (number type)
All string types	S (string type)
<code>MemoryStream</code> , <code>byte[]</code>	B (binary type)

.NET primitive type	DynamoDB type
bool	N (number type). 0 represents false and 1 represents true.
Collection types	BS (binary set) type, SS (string set) type, and NS (number set) type
DateTime	S (string type). The DateTime values are stored as ISO-8601 formatted strings.

The object persistence model also supports arbitrary data types. However, you must provide converter code to map the complex types to the DynamoDB types.

 **Note**

- Empty binary values are supported.
- Reading of empty string values is supported. Empty string attribute values are supported within attribute values of string Set type while writing to DynamoDB. Empty string attribute values of string type and empty string values contained within List or Map type are dropped from write requests

Optimistic locking using a version number with DynamoDB using the AWS SDK for .NET object persistence model

Optimistic locking support in the object persistence model ensures that the item version for your application is the same as the item version on the server side before updating or deleting the item. Suppose that you retrieve an item for update. However, before you send your updates back, some other application updates the same item. Now your application has a stale copy of the item. Without optimistic locking, any update you perform will overwrite the update made by the other application.

The optimistic locking feature of the object persistence model provides the `DynamoDBVersion` tag that you can use to enable optimistic locking. To use this feature, you add a property to your class for storing the version number. You add the `DynamoDBVersion` attribute to the property.

When you first save the object, the `DynamoDBContext` assigns a version number and increments this value each time you update the item.

Your update or delete request succeeds only if the client-side object version matches the corresponding version number of the item on the server side. If your application has a stale copy, it must get the latest version from the server before it can update or delete that item.

The following C# code example defines a `Book` class with object persistence attributes mapping it to the `ProductCatalog` table. The `VersionNumber` property in the class decorated with the `DynamoDBVersion` attribute stores the version number value.

Example

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id { get; set; }
    [DynamoDBProperty]
    public string Title { get; set; }
    [DynamoDBProperty]
    public string ISBN { get; set; }
    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }
    [DynamoDBVersion]
    public int? VersionNumber { get; set; }
}
```

Note

You can apply the `DynamoDBVersion` attribute only to a nullable numeric primitive type (such as `int?`).

Optimistic locking has the following impact on `DynamoDBContext` operations:

- For a new item, `DynamoDBContext` assigns initial version number 0. If you retrieve an existing item, update one or more of its properties, and try to save the changes, the save operation succeeds only if the version number on the client side and the server side match. `DynamoDBContext` increments the version number. You don't need to set the version number.

- The Delete method provides overloads that can take either a primary key value or an object as parameter, as shown in the following C# code example.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
...
// Load a book.
Book book = context.Load<ProductCatalog>(111);
// Do other operations.
// Delete 1 - Pass in the book object.
context.Delete<ProductCatalog>(book);

// Delete 2 - Pass in the Id (primary key)
context.Delete<ProductCatalog>(222);
```

If you provide an object as the parameter, the delete succeeds only if the object version matches the corresponding server-side item version. However, if you provide a primary key value as the parameter, DynamoDBContext is unaware of any version numbers, and it deletes the item without making the version check.

Note that the internal implementation of optimistic locking in the object persistence model code uses the conditional update and the conditional delete API actions in DynamoDB.

Disabling optimistic locking

To disable optimistic locking, you use the SkipVersionCheck configuration property. You can set this property when creating DynamoDBContext. In this case, optimistic locking is disabled for any requests that you make using the context. For more information, see [Specifying optional parameters for DynamoDBContext](#).

Instead of setting the property at the context level, you can disable optimistic locking for a specific operation, as shown in the following C# code example. The example uses the context to delete a book item. The Delete method sets the optional SkipVersionCheck property to true, disabling version checking.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
// Load a book.
```

```
Book book = context.Load<ProductCatalog>(111);
...
// Delete the book.
context.Delete<Book>(book, new DynamoDBContextConfig { SkipVersionCheck = true });
```

Mapping arbitrary data with DynamoDB using the AWS SDK for .NET object persistence model

In addition to the supported .NET types (see [Supported data types](#)), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. The object persistence model supports storing data of arbitrary types as long as you provide the converter to convert data from the arbitrary type to the DynamoDB type and vice versa. The converter code transforms data during both the saving and loading of the objects.

You can create any types on the client-side. However the data stored in the tables is one of the DynamoDB types, and during query and scan, any data comparisons made are against the data stored in DynamoDB.

The following C# code example defines a Book class with Id, Title, ISBN, and Dimension properties. The Dimension property is of the DimensionType that describes Height, Width, and Thickness properties. The example code provides the converter methods ToEntry and FromEntry to convert data between the DimensionType and the DynamoDB string types. For example, when saving a Book instance, the converter creates a book Dimension string such as "8.5x11x.05". When you retrieve a book, it converts the string to a DimensionType instance.

The example maps the Book type to the ProductCatalog table. It saves a sample Book instance, retrieves it, updates its dimensions, and saves the updated Book again.

For step-by-step instructions for testing the following example, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
```

```
{  
    class HighLevelMappingArbitraryData  
    {  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
  
        static void Main(string[] args)  
        {  
            try  
            {  
                DynamoDBContext context = new DynamoDBContext(client);  
  
                // 1. Create a book.  
                DimensionType myBookDimensions = new DimensionType()  
                {  
                    Length = 8M,  
                    Height = 11M,  
                    Thickness = 0.5M  
                };  
  
                Book myBook = new Book  
                {  
                    Id = 501,  
                    Title = "AWS SDK for .NET Object Persistence Model Handling  
Arbitrary Data",  
                    ISBN = "999-9999999999",  
                    BookAuthors = new List<string> { "Author 1", "Author 2" },  
                    Dimensions = myBookDimensions  
                };  
  
                context.Save(myBook);  
  
                // 2. Retrieve the book.  
                Book bookRetrieved = context.Load<Book>(501);  
  
                // 3. Update property (book dimensions).  
                bookRetrieved.Dimensions.Height += 1;  
                bookRetrieved.Dimensions.Length += 1;  
                bookRetrieved.Dimensions.Thickness += 0.2M;  
                // Update the book.  
                context.Save(bookRetrieved);  
  
                Console.WriteLine("To continue, press Enter");  
                Console.ReadLine();  
            }  
        }  
    }  
}
```

```
        catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }
}

[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id
    {
        get; set;
    }
    [DynamoDBProperty]
    public string Title
    {
        get; set;
    }
    [DynamoDBProperty]
    public string ISBN
    {
        get; set;
    }
    // Multi-valued (set type) attribute.
    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors
    {
        get; set;
    }
    // Arbitrary type, with a converter to map it to DynamoDB type.
    [DynamoDBProperty(typeof(DimensionTypeConverter))]
    public DimensionType Dimensions
    {
        get; set;
    }
}

public class DimensionType
{
    public decimal Length
    {
        get; set;
    }
    public decimal Height
```

```
{  
    get; set;  
}  
public decimal Thickness  
{  
    get; set;  
}  
}  
  
// Converts the complex type DimensionType to string and vice-versa.  
public class DimensionTypeConverter : IPropertyConverter  
{  
    public DynamoDBEntry ToEntry(object value)  
    {  
        DimensionType bookDimensions = value as DimensionType;  
        if (bookDimensions == null) throw new ArgumentOutOfRangeException();  
  
        string data = string.Format("{1}{0}{2}{0}{3}", " x ",  
                                     bookDimensions.Length, bookDimensions.Height,  
bookDimensions.Thickness);  
  
        DynamoDBEntry entry = new Primitive  
        {  
            Value = data  
        };  
        return entry;  
    }  
  
    public object FromEntry(DynamoDBEntry entry)  
    {  
        Primitive primitive = entry as Primitive;  
        if (primitive == null || !(primitive.Value is String) ||  
string.IsNullOrEmpty((string)primitive.Value))  
            throw new ArgumentOutOfRangeException();  
  
        string[] data = ((string)(primitive.Value)).Split(new string[] { " x " },  
StringSplitOptions.None);  
        if (data.Length != 3) throw new ArgumentOutOfRangeException();  
  
        DimensionType complexData = new DimensionType  
        {  
            Length = Convert.ToDecimal(data[0]),  
            Height = Convert.ToDecimal(data[1]),  
            Thickness = Convert.ToDecimal(data[2])  
        };  
    }  
}
```

```
    };
    return complexData;
}
}
```

Batch operations using the AWS SDK for .NET object persistence model

Batch write: Putting and deleting multiple items

To put or delete multiple objects from a table in a single request, do the following:

- Run the `createBatchWrite` method of the `DynamoDBContext`, and create an instance of the `BatchWrite` class.
- Specify the items that you want to put or delete.
 - To put one or more items, use either the `AddPutItem` or the `AddPutItems` method.
 - To delete one or more items, you can specify either the primary key of the item or a client-side object that maps to the item that you want to delete. Use the `AddDeleteItem`, `AddDeleteItems`, and the `AddDeleteKey` methods to specify the list of items to delete.
- Call the `BatchWrite.Execute` method to put and delete all the specified items from the table.

Note

When using the object persistence model, you can specify any number of operations in a batch. However, note that Amazon DynamoDB limits the number of operations in a batch and the total size of the batch in a batch operation. For more information about the specific limits, see [BatchWriteItem](#). If the API detects that your batch write request exceeded the allowed number of write requests or exceeded the maximum allowed HTTP payload size, it breaks the batch into several smaller batches. Additionally, if a response to a batch write returns unprocessed items, the API automatically sends another batch request with those unprocessed items.

Suppose that you defined a C# class `Book` class that maps to the `ProductCatalog` table in DynamoDB. The following C# code example uses the `BatchWrite` object to upload two items and delete one item from the `ProductCatalog` table.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
var bookBatch = context.CreateBatchWrite<Book>();

// 1. Specify two books to add.
Book book1 = new Book
{
    Id = 902,
    ISBN = "902-11-11-1111",
    ProductCategory = "Book",
    Title = "My book3 in batch write"
};

Book book2 = new Book
{
    Id = 903,
    ISBN = "903-11-11-1111",
    ProductCategory = "Book",
    Title = "My book4 in batch write"
};

bookBatch.AddPutItems(new List<Book> { book1, book2 });

// 2. Specify one book to delete.
bookBatch.AddDeleteKey(111);

bookBatch.Execute();
```

To put or delete objects from multiple tables, do the following:

- Create one instance of the BatchWrite class for each type and specify the items you want to put or delete as described in the preceding section.
- Create an instance of MultiTableBatchWrite using one of the following methods:
 - Run the Combine method on one of the BatchWrite objects that you created in the preceding step.
 - Create an instance of the MultiTableBatchWrite type by providing a list of BatchWrite objects.
 - Run the CreateMultiTableBatchWrite method of DynamoDBContext and pass in your list of BatchWrite objects.

- Call the Execute method of MultiTableBatchWrite, which performs the specified put and delete operations on various tables.

Suppose that you defined Forum and Thread C# classes that map to the Forum and Thread tables in DynamoDB. Also, suppose that the Thread class has versioning enabled. Because versioning is not supported when using batch operations, you must explicitly disable versioning as shown in the following C# code example. The example uses the MultiTableBatchWrite object to perform a multi-table update.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
// Create BatchWrite objects for each of the Forum and Thread classes.
var forumBatch = context.CreateBatchWrite<Forum>();

DynamoDBOperationConfig config = new DynamoDBOperationConfig();
config.SkipVersionCheck = true;
var threadBatch = context.CreateBatchWrite<Thread>(config);

// 1. New Forum item.
Forum newForum = new Forum
{
    Name = "Test BatchWrite Forum",
    Threads = 0
};
forumBatch.AddPutItem(newForum);

// 2. Specify a forum to delete by specifying its primary key.
forumBatch.AddDeleteKey("Some forum");

// 3. New Thread item.
Thread newThread = new Thread
{
    ForumName = "Amazon S3 forum",
    Subject = "My sample question",
    KeywordTags = new List<string> { "Amazon S3", "Bucket" },
    Message = "Message text"
};

threadBatch.AddPutItem(newThread);

// Now run multi-table batch write.
```

```
var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);
superBatch.Execute();
```

For a working example, see [Example: Batch write operation using the AWS SDK for .NET object persistence model](#).

Note

The DynamoDB batch API limits the number of writes in a batch and also limits the size of the batch. For more information, see [BatchWriteItem](#). When using the .NET object persistence model API, you can specify any number of operations. However, if either the number of operations in a batch or the size exceeds the limit, the .NET API breaks the batch write request into smaller batches and sends multiple batch write requests to DynamoDB.

Batch get: Getting multiple items

To retrieve multiple items from a table in a single request, do the following:

- Create an instance of the `CreateBatchGet` class.
- Specify a list of primary keys to retrieve.
- Call the `Execute` method. The response returns the items in the `Results` property.

The following C# code example retrieves three items from the `ProductCatalog` table. The items in the result are not necessarily in the same order in which you specified the primary keys.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
var bookBatch = context.CreateBatchGet<ProductCatalog>();
bookBatch.AddKey(101);
bookBatch.AddKey(102);
bookBatch.AddKey(103);
bookBatch.Execute();
// Process result.
Console.WriteLine(bookBatch.Results.Count);
Book book1 = bookBatch.Results[0];
Book book2 = bookBatch.Results[1];
Book book3 = bookBatch.Results[2];
```

To retrieve objects from multiple tables, do the following:

- For each type, create an instance of the `CreateBatchGet` type and provide the primary key values you want to retrieve from each table.
- Create an instance of the `MultiTableBatchGet` class using one of the following methods:
 - Run the `Combine` method on one of the `BatchGet` objects you created in the preceding step.
 - Create an instance of the `MultiBatchGet` type by providing a list of `BatchGet` objects.
 - Run the `CreateMultiTableBatchGet` method of `DynamoDBContext` and pass in your list of `BatchGet` objects.
- Call the `Execute` method of `MultiTableBatchGet`, which returns the typed results in the individual `BatchGet` objects.

The following C# code example retrieves multiple items from the `Order` and `OrderDetail` tables using the `CreateBatchGet` method.

Example

```
var orderBatch = context.CreateBatchGet<Order>();
orderBatch.AddKey(101);
orderBatch.AddKey(102);

var orderDetailBatch = context.CreateBatchGet<OrderDetail>();
orderDetailBatch.AddKey(101, "P1");
orderDetailBatch.AddKey(101, "P2");
orderDetailBatch.AddKey(102, "P3");
orderDetailBatch.AddKey(102, "P1");

var orderAndDetailSuperBatch = orderBatch.Combine(orderDetailBatch);
orderAndDetailSuperBatch.Execute();

Console.WriteLine(orderBatch.Results.Count);
Console.WriteLine(orderDetailBatch.Results.Count);

Order order1 = orderBatch.Results[0];
Order order2 = orderBatch.Results[1];
OrderDetail orderDetail1 = orderDetailBatch.Results[0];
```

Example: CRUD operations using the AWS SDK for .NET object persistence model

The following C# code example declares a Book class with Id, Title, ISBN, and Authors properties. The example uses object persistence attributes to map these properties to the ProductCatalog table in Amazon DynamoDB. The example then uses the DynamoDBContext to illustrate typical create, read, update, and delete (CRUD) operations. The example creates a sample Book instance and saves it to the ProductCatalog table. It then retrieves the book item and updates its ISBN and Authors properties. Note that the update replaces the existing authors list. Finally, the example deletes the book item.

For more information about the ProductCatalog table used in this example, see [Creating tables and loading data for code examples in DynamoDB](#). For step-by-step instructions to test the following example, see [.NET code examples](#).

Note

The following example doesn't work with .NET core because it doesn't support synchronous methods. For more information, see [AWS asynchronous APIs for .NET](#).

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class HighLevelItemCRUD
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                DynamoDBContext context = new DynamoDBContext(client);
                TestCRUDOperations(context);
                Console.WriteLine("To continue, press Enter");
            }
        }
    }
}
```

```
        Console.ReadLine();
    }
    catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
    catch (Exception e) { Console.WriteLine(e.Message); }
}

private static void TestCRUDOperations(DynamoDBContext context)
{
    int bookID = 1001; // Some unique value.
    Book myBook = new Book
    {
        Id = bookID,
        Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
        ISBN = "111-1111111001",
        BookAuthors = new List<string> { "Author 1", "Author 2" },
    };

    // Save the book.
    context.Save(myBook);
    // Retrieve the book.
    Book bookRetrieved = context.Load<Book>(bookID);

    // Update few properties.
    bookRetrieved.ISBN = "222-2222221001";
    bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author x" }; // Replace existing authors list with this.
    context.Save(bookRetrieved);

    // Retrieve the updated book. This time add the optional ConsistentRead
    parameter using DynamoDBContextConfig object.
    Book updatedBook = context.Load<Book>(bookID, new DynamoDBContextConfig
    {
        ConsistentRead = true
    });

    // Delete the book.
    context.Delete<Book>(bookID);
    // Try to retrieve deleted book. It should return null.
    Book deletedBook = context.Load<Book>(bookID, new DynamoDBContextConfig
    {
        ConsistentRead = true
    });
    if (deletedBook == null)
```

```
        Console.WriteLine("Book is deleted");
    }
}

[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id
    {
        get; set;
    }
    [DynamoDBProperty]
    public string Title
    {
        get; set;
    }
    [DynamoDBProperty]
    public string ISBN
    {
        get; set;
    }
    [DynamoDBProperty("Authors")] //String Set datatype
    public List<string> BookAuthors
    {
        get; set;
    }
}
}
```

Example: Batch write operation using the AWS SDK for .NET object persistence model

The following C# code example declares Book, Forum, Thread, and Reply classes and maps them to Amazon DynamoDB tables using the object persistence model attributes.

The example then uses the DynamoDBContext to illustrate the following batch write operations:

- `BatchWrite` object to put and delete book items from the `ProductCatalog` table.
- `MultiTableBatchWrite` object to put and delete items from the `Forum` and the `Thread` tables.

For more information about the tables used in this example, see [Creating tables and loading data for code examples in DynamoDB](#). For step-by-step instructions to test the following example, see [.NET code examples](#).

 **Note**

The following example doesn't work with .NET core because it doesn't support synchronous methods. For more information, see [AWS asynchronous APIs for .NET](#).

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class HighLevelBatchWriteItem
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                DynamoDBContext context = new DynamoDBContext(client);
                SingleTableBatchWrite(context);
                MultiTableBatchWrite(context);
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void SingleTableBatchWrite(DynamoDBContext context)
        {
```

```
Book book1 = new Book
{
    Id = 902,
    InPublication = true,
    ISBN = "902-11-11-1111",
    PageCount = "100",
    Price = 10,
    ProductCategory = "Book",
    Title = "My book3 in batch write"
};

Book book2 = new Book
{
    Id = 903,
    InPublication = true,
    ISBN = "903-11-11-1111",
    PageCount = "200",
    Price = 10,
    ProductCategory = "Book",
    Title = "My book4 in batch write"
};

var bookBatch = context.CreateBatchWrite<Book>();
bookBatch.AddPutItems(new List<Book> { book1, book2 });

Console.WriteLine("Performing batch write in SingleTableBatchWrite().");
bookBatch.Execute();
}

private static void MultiTableBatchWrite(DynamoDBContext context)
{
    // 1. New Forum item.
    Forum newForum = new Forum
    {
        Name = "Test BatchWrite Forum",
        Threads = 0
    };
    var forumBatch = context.CreateBatchWrite<Forum>();
    forumBatch.AddPutItem(newForum);

    // 2. New Thread item.
    Thread newThread = new Thread
    {
        ForumName = "S3 forum",
        Subject = "My sample question",
    
```

```
        KeywordTags = new List<string> { "S3", "Bucket" },
        Message = "Message text"
    };

    DynamoDBOperationConfig config = new DynamoDBOperationConfig();
    config.SkipVersionCheck = true;
    var threadBatch = context.CreateBatchWrite<Thread>(config);
    threadBatch.AddPutItem(newThread);
    threadBatch.AddDeleteKey("some partition key value", "some sort key
value");

    var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);
    Console.WriteLine("Performing batch write in MultiTableBatchWrite().");
    superBatch.Execute();
}
}

[DynamoDBTable("Reply")]
public class Reply
{
    [DynamoDBHashKey] //Partition key
    public string Id
    {
        get; set;
    }

    [DynamoDBRangeKey] //Sort key
    public DateTime ReplyDateTime
    {
        get; set;
    }

    // Properties included implicitly.
    public string Message
    {
        get; set;
    }
    // Explicit property mapping with object persistence model attributes.
    [DynamoDBProperty("LastPostedBy")]
    public string PostedBy
    {
        get; set;
    }
    // Property to store version number for optimistic locking.
```

```
[DynamoDBVersion]
public int? Version
{
    get; set;
}

[DynamoDBTable("Thread")]
public class Thread
{
    // PK mapping.
    [DynamoDBHashKey]      //Partition key
    public string ForumName
    {
        get; set;
    }
    [DynamoDBRangeKey]      //Sort key
    public String Subject
    {
        get; set;
    }
    // Implicit mapping.
    public string Message
    {
        get; set;
    }
    public string LastPostedBy
    {
        get; set;
    }
    public int Views
    {
        get; set;
    }
    public int Replies
    {
        get; set;
    }
    public bool Answered
    {
        get; set;
    }
    public DateTime LastPostedDateTime
    {
```

```
        get; set;
    }
    // Explicit mapping (property and table attribute names are different.
    [DynamoDBProperty("Tags")]
    public List<string> KeywordTags
    {
        get; set;
    }
    // Property to store version number for optimistic locking.
    [DynamoDBVersion]
    public int? Version
    {
        get; set;
    }
}

[DynamoDBTable("Forum")]
public class Forum
{
    [DynamoDBHashKey]          //Partition key
    public string Name
    {
        get; set;
    }
    // All the following properties are explicitly mapped,
    // only to show how to provide mapping.
    [DynamoDBProperty]
    public int Threads
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Views
    {
        get; set;
    }
    [DynamoDBProperty]
    public string LastPostBy
    {
        get; set;
    }
    [DynamoDBProperty]
    public DateTime LastPostDateTime
    {
```

```
        get; set;
    }
    [DynamoDBProperty]
    public int Messages
    {
        get; set;
    }
}

[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id
    {
        get; set;
    }
    public string Title
    {
        get; set;
    }
    public string ISBN
    {
        get; set;
    }
    public int Price
    {
        get; set;
    }
    public string PageCount
    {
        get; set;
    }
    public string ProductCategory
    {
        get; set;
    }
    public bool InPublication
    {
        get; set;
    }
}
```

Example: Query and scan in DynamoDB using the AWS SDK for .NET object persistence model

The C# example in this section defines the following classes and maps them to the tables in DynamoDB. For more information about creating the tables used in this example, see [Creating tables and loading data for code examples in DynamoDB](#).

- The Book class maps to the ProductCatalog table.
- The Forum, Thread, and Reply classes map to tables of the same name.

The example then runs the following query and scan operations using DynamoDBContext.

- Get a book by Id.

The ProductCatalog table has Id as its primary key. It does not have a sort key as part of its primary key. Therefore, you cannot query the table. You can get an item using its Id value.

- Run the following queries against the Reply table. (The Reply table's primary key is composed of Id and ReplyDateTime attributes. The ReplyDateTime is a sort key. Therefore, you can query this table.)
 - Find replies to a forum thread posted in the last 15 days.
 - Find replies to a forum thread posted in a specific date range.
- Scan the ProductCatalog table to find books whose price is less than zero.

For performance reasons, you should use a query operation instead of a scan operation.

However, there are times you might need to scan a table. Suppose that there was a data entry error and one of the book prices is set to less than 0. This example scans the ProductCategory table to find book items (the ProductCategory is book) at price of less than 0.

For instructions about creating a working sample, see [.NET code examples](#).

Note

The following example does not work with .NET core because it does not support synchronous methods. For more information, see [AWS asynchronous APIs for .NET](#).

Example

```
using System;
using System.Collections.Generic;
using System.Configuration;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class HighLevelQueryAndScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                DynamoDBContext context = new DynamoDBContext(client);
                // Get an item.
                GetBook(context, 101);

                // Sample forum and thread to test queries.
                string forumName = "Amazon DynamoDB";
                string threadSubject = "DynamoDB Thread 1";
                // Sample queries.
                FindRepliesInLast15Days(context, forumName, threadSubject);
                FindRepliesPostedWithinTimePeriod(context, forumName, threadSubject);

                // Scan table.
                FindProductsPricedLessThanZero(context);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void GetBook(DynamoDBContext context, int productId)
        {
            Book bookItem = context.Load<Book>(productId);
```

```
Console.WriteLine("\nGetBook: Printing result.....");
Console.WriteLine("Title: {0} \n No.Of threads:{1} \n No. of messages:
{2}",
bookItem.Title, bookItem.ISBN, bookItem.PageCount);
}

private static void FindRepliesInLast15Days(DynamoDBContext context,
                                             string forumName,
                                             string threadSubject)
{
    string replyId = forumName + "#" + threadSubject;
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
    IEnumerable<Reply> latestReplies =
        context.Query<Reply>(replyId, QueryOperator.GreaterThan,
twoWeeksAgoDate);
    Console.WriteLine("\nFindRepliesInLast15Days: Printing result.....");
    foreach (Reply r in latestReplies)
        Console.WriteLine("{0}\t{1}\t{2}\t{3}", r.Id, r.PostedBy, r.Message,
r.ReplyDateTime);
}

private static void FindRepliesPostedWithinTimePeriod(DynamoDBContext context,
                                                       string forumName,
                                                       string threadSubject)
{
    string forumId = forumName + "#" + threadSubject;
    Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: Printing
result.....");

    DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(30);
    DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(1);

    IEnumerable<Reply> repliesInAPeriod = context.Query<Reply>(forumId,
                                                               QueryOperator.Between, startDate, endDate);
    foreach (Reply r in repliesInAPeriod)
        Console.WriteLine("{0}\t{1}\t{2}\t{3}", r.Id, r.PostedBy, r.Message,
r.ReplyDateTime);
}

private static void FindProductsPricedLessThanZero(DynamoDBContext context)
{
    int price = 0;
    IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(

```

```
        new ScanCondition("Price", ScanOperator.LessThan, price),
        new ScanCondition("ProductCategory", ScanOperator.Equal, "Book")
    );
    Console.WriteLine("\nFindProductsPricedLessThanZero: Printing
result.....");
    foreach (Book r in itemsWithWrongPrice)
        Console.WriteLine("{0}\t{1}\t{2}\t{3}", r.Id, r.Title, r.Price,
r.ISBN);
}
}

[DynamoDBTable("Reply")]
public class Reply
{
    [DynamoDBHashKey] //Partition key
    public string Id
    {
        get; set;
    }

    [DynamoDBRangeKey] //Sort key
    public DateTime ReplyDateTime
    {
        get; set;
    }

    // Properties included implicitly.
    public string Message
    {
        get; set;
    }
    // Explicit property mapping with object persistence model attributes.
    [DynamoDBProperty("LastPostedBy")]
    public string PostedBy
    {
        get; set;
    }
    // Property to store version number for optimistic locking.
    [DynamoDBVersion]
    public int? Version
    {
        get; set;
    }
}
```

```
[DynamoDBTable("Thread")]
public class Thread
{
    // Partition key mapping.
    [DynamoDBHashKey] //Partition key
    public string ForumName
    {
        get; set;
    }
    [DynamoDBRangeKey] //Sort key
    public DateTime Subject
    {
        get; set;
    }
    // Implicit mapping.
    public string Message
    {
        get; set;
    }
    public string LastPostedBy
    {
        get; set;
    }
    public int Views
    {
        get; set;
    }
    public int Replies
    {
        get; set;
    }
    public bool Answered
    {
        get; set;
    }
    public DateTime LastPostedDateTime
    {
        get; set;
    }
    // Explicit mapping (property and table attribute names are different).
    [DynamoDBProperty("Tags")]
    public List<string> KeywordTags
    {
```

```
        get; set;
    }
    // Property to store version number for optimistic locking.
    [DynamoDBVersion]
    public int? Version
    {
        get; set;
    }
}

[DynamoDBTable("Forum")]
public class Forum
{
    [DynamoDBHashKey]
    public string Name
    {
        get; set;
    }
    // All the following properties are explicitly mapped
    // to show how to provide mapping.
    [DynamoDBProperty]
    public int Threads
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Views
    {
        get; set;
    }
    [DynamoDBProperty]
    public string LastPostBy
    {
        get; set;
    }
    [DynamoDBProperty]
    public DateTime LastPostDateTime
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Messages
    {
        get; set;
    }
}
```

```
        }  
    }  
  
    [DynamoDBTable("ProductCatalog")]  
    public class Book  
    {  
        [DynamoDBHashKey] //Partition key  
        public int Id  
        {  
            get; set;  
        }  
        public string Title  
        {  
            get; set;  
        }  
        public string ISBN  
        {  
            get; set;  
        }  
        public int Price  
        {  
            get; set;  
        }  
        public string PageCount  
        {  
            get; set;  
        }  
        public string ProductCategory  
        {  
            get; set;  
        }  
        public bool InPublication  
        {  
            get; set;  
        }  
    }  
}
```

Running the code examples in this Developer Guide

The AWS SDKs provide broad support for Amazon DynamoDB in the following languages:

- [Java](#)

- [JavaScript in the browser](#)
- [.NET](#)
- [Node.js](#)
- [PHP](#)
- [Python](#)
- [Ruby](#)
- [C++](#)
- [Go](#)
- [Android](#)
- [iOS](#)

To get started quickly with these languages, see [Getting started with DynamoDB and the AWS SDKs](#).

The code examples in this developer guide provide more in-depth coverage of DynamoDB operations, using the following programming languages:

- [Java code examples](#)
- [.NET code examples](#)

Before you can begin with this exercise, you need to create an AWS account, get your access key and secret key, and set up the AWS Command Line Interface (AWS CLI) on your computer. For more information, see [Setting up DynamoDB \(web service\)](#).

 **Note**

If you are using the downloadable version of DynamoDB, you need to use the AWS CLI to create the tables and sample data. You also need to specify the `--endpoint-url` parameter with each AWS CLI command. For more information, see [Setting the local endpoint](#).

Creating tables and loading data for code examples in DynamoDB

See below for the basics on creating tables in DynamoDB, loading in a sample dataset, querying the data, and updating the data.

- [Step 1: Create a table](#)
- [Step 2: Write data to a table using the console or AWS CLI](#)
- [Step 3: Read data from a table](#)
- [Step 4: Update data in a table](#)

Java code examples

Topics

- [Java: Setting your AWS credentials](#)
- [Java: Setting the AWS Region and endpoint](#)

This Developer Guide contains Java code snippets and ready-to-run programs. You can find these code examples in the following sections:

- [Working with items and attributes](#)
- [Working with tables and data in DynamoDB](#)
- [Query operations in DynamoDB](#)
- [Working with scans in DynamoDB](#)
- [Improving data access with secondary indexes](#)
- [Java 1.x: DynamoDBMapper](#)
- [Change data capture for DynamoDB Streams](#)

You can get started quickly by using Eclipse with the [AWS Toolkit for Eclipse](#). In addition to a full-featured IDE, you also get the AWS SDK for Java with automatic updates, and preconfigured templates for building AWS applications.

To run the Java code examples (using Eclipse)

1. Download and install the [Eclipse](#) IDE.

2. Download and install the [AWS Toolkit for Eclipse](#).
3. Start Eclipse, and on the **Eclipse** menu, choose **File**, **New**, and then **Other**.
4. In **Select a wizard**, choose **AWS**, choose **AWS Java Project**, and then choose **Next**.
5. In **Create an AWS Java**, do the following:
 - a. In **Project name**, enter a name for your project.
 - b. In **Select Account**, choose your credentials profile from the list.

If this is your first time using the [AWS Toolkit for Eclipse](#), choose **Configure AWS Accounts** to set up your AWS credentials.
6. Choose **Finish** to create the project.
7. From the **Eclipse** menu, choose **File**, **New**, and then **Class**.
8. In **Java Class**, enter a name for your class in **Name** (use the same name as the code example that you want to run), and then choose **Finish** to create the class.
9. Copy the code example from the documentation page into the Eclipse editor.
10. To run the code, choose **Run** on the Eclipse menu.

The SDK for Java provides thread-safe clients for working with DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information, see the [AWS SDK for Java](#).

 **Note**

The code examples in this guide are intended for use with the latest version of the AWS SDK for Java.

If you are using the AWS Toolkit for Eclipse, you can configure automatic updates for the SDK for Java. To do this in Eclipse, go to **Preferences** and choose **AWS Toolkit**, **AWS SDK for Java**, **Download new SDKs automatically**.

Java: Setting your AWS credentials

The SDK for Java requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using an AWS credentials file, as described in [Set up your AWS credentials](#) in the *AWS SDK for Java Developer Guide*.

The following is an example of an AWS credentials file named `~/.aws/credentials`, where the tilde character (~) represents your home directory.

```
[default]
aws_access_key_id = AWS access key ID goes here
aws_secret_access_key = Secret key goes here
```

Java: Setting the AWS Region and endpoint

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AmazonDynamoDB` properties.

The following code example instantiates a new `AmazonDynamoDB`.

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.regions.Regions;
...
// This client will default to US West (Oregon)
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.US_WEST_2)
    .build();
```

You can use the `withRegion` method to run your code against DynamoDB in any Region where it is available. For a complete list, see [AWS regions and endpoints](#) in the *Amazon Web Services General Reference*.

If you want to run the code examples using DynamoDB locally on your computer, set the endpoint as follows.

AWS SDK V1

```
AmazonDynamoDB client =
    AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
    .build();
```

AWS SDK V2

```
DynamoDbClient client = DynamoDbClient.builder()
    .endpointOverride(URI.create("http://localhost:8000"))
```

```
// The region is meaningless for local DynamoDb but required for client builder validation  
.region(Region.US_EAST_1)  
.credentialsProvider(StaticCredentialsProvider.create(  
AwsBasicCredentials.create("dummy-key", "dummy-secret")))  
.build();
```

.NET code examples

Topics

- [.NET: Setting your AWS credentials](#)
- [.NET: Setting the AWS Region and endpoint](#)

This guide contains .NET code snippets and ready-to-run programs. You can find these code examples in the following sections:

- [Working with items and attributes](#)
- [Working with tables and data in DynamoDB](#)
- [Query operations in DynamoDB](#)
- [Working with scans in DynamoDB](#)
- [Improving data access with secondary indexes](#)
- [.NET: Document model](#)
- [.NET: Object persistence model](#)
- [Change data capture for DynamoDB Streams](#)

You can get started quickly by using the AWS SDK for .NET with the Toolkit for Visual Studio.

To run the .NET code examples (using Visual Studio)

1. Download and install [Microsoft Visual Studio](#).
2. Download and install the [Toolkit for Visual Studio](#).
3. Start Visual Studio. Choose **File, New, Project**.
4. In **New Project**, choose **AWS Empty Project**, and then choose **OK**.
5. In **AWS Access Credentials**, choose **Use existing profile**, choose your credentials profile from the list, and then choose **OK**.

- If this is your first time using Toolkit for Visual Studio, choose **Use a new profile** to set up your AWS credentials.
6. In your Visual Studio project, choose the tab for your program's source code (**Program.cs**). Copy the code example from the documentation page into the Visual Studio editor, replacing any other code that you see in the editor.
 7. If you see error messages of the form The type or namespace name...could not be found, you need to install the AWS SDK assembly for DynamoDB as follows:
 - a. In Solution Explorer, open the context (right-click) menu for your project, and then choose **Manage NuGet Packages**.
 - b. In NuGet Package Manager, choose **Browse**.
 - c. In the search box, enter **AWSSDK.DynamoDBv2**, and wait for the search to complete.
 - d. Choose **AWSSDK.DynamoDBv2**, and then choose **Install**.
 - e. When the installation is complete, choose the **Program.cs** tab to return to your program.
 8. To run the code, choose **Start** in the Visual Studio toolbar.

The AWS SDK for .NET provides thread-safe clients for working with DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information, see [AWS SDK for .NET](#).

 **Note**

The code examples in this guide are intended for use with the latest version of the AWS SDK for .NET.

.NET: Setting your AWS credentials

The AWS SDK for .NET requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using the SDK Store to manage your AWS credentials file, as described in [Using the SDK store](#) in the *AWS SDK for .NET Developer Guide*.

The Toolkit for Visual Studio supports multiple sets of credentials from any number of accounts. Each set is referred to as a *profile*. Visual Studio adds entries to the project's App.config file so that your application can find the AWS credentials at runtime.

The following example shows the default App.config file that is generated when you create a new project using Toolkit for Visual Studio.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
        <add key="AWSProfileName" value="default"/>
        <add key="AWSRegion" value="us-west-2" />
    </appSettings>
</configuration>
```

At runtime, the program uses the default set of AWS credentials, as specified by the AWSProfileName entry. The AWS credentials themselves are kept in the SDK Store in encrypted form. The Toolkit for Visual Studio provides a graphical user interface for managing your credentials, all from within Visual Studio. For more information, see [Specifying credentials](#) in the [AWS Toolkit for Visual Studio User Guide](#).

 **Note**

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the AWSRegion entry in the App.config file. You can set AWSRegion to any Region where DynamoDB is available. For a complete list, see [AWS regions and endpoints](#) in the *Amazon Web Services General Reference*.

.NET: Setting the AWS Region and endpoint

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the AWSRegion entry in the App.config file. Or, you can change the Region by modifying the AmazonDynamoDBClient properties.

The following code example instantiates a new AmazonDynamoDBClient. The client is modified so that the code runs against DynamoDB in a different Region.

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// This client will access the US East 1 region.
clientConfig.RegionEndpoint = RegionEndpoint.USEast1;
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

For a complete list of Regions, see [AWS regions and endpoints](#) in the *Amazon Web Services General Reference*.

If you want to run the code examples using DynamoDB locally on your computer, set the endpoint as follows.

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// Set the endpoint URL
clientConfig.ServiceURL = "http://localhost:8000";
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

Programming Amazon DynamoDB with Python and Boto3

This guide provides an orientation to programmers wanting to use Amazon DynamoDB with Python. Learn about the different abstraction layers, configuration management, error handling, controlling retry policies, managing keep-alive, and more.

Topics

- [About Boto](#)
- [Using the Boto documentation](#)
- [Understanding the client and resource abstraction layers](#)
- [Using the table resource batch_writer](#)
- [Additional code examples that explore the client and resource layers](#)
- [Understanding how the Client and Resource objects interact with sessions and threads](#)
- [Customizing the Config object](#)
- [Error handling](#)
- [Logging](#)
- [Event hooks](#)
- [Pagination and the Paginator](#)
- [Waiters](#)

About Boto

You can access DynamoDB from Python by using the official AWS SDK for Python, commonly referred to as **Boto3**. The name Boto (pronounced boh-toh) comes from a freshwater dolphin

native to the Amazon River. The Boto3 library is the library's third major version, first released in 2015. The Boto3 library is quite large, as it supports all AWS services, not just DynamoDB. This orientation targets only the parts of Boto3 relevant to DynamoDB.

Boto is maintained and published by AWS as open-source project hosted on GitHub. It's split into two packages: [Botocore](#) and [Boto3](#).

- **Botocore** provides the low-level functionality. In Botocore you'll find the client, session, credentials, config, and exception classes.
- **Boto3** builds on top of Botocore. It offers a higher-level, more Pythonic interface. Specifically, it exposes a DynamoDB table as a Resource and offers a simpler, more elegant interface compared to the lower-level, service-oriented client interface.

Because these projects are hosted on GitHub, you can view the source code, track open issues, or submit your own issues.

Using the Boto documentation

Get started with the Boto documentation with the following resources:

- Begin with the [Quickstart section](#) that provides a solid starting point for the package installation. Go there for instructions on getting Boto3 installed if it's not already (Boto3 is often automatically available within AWS services such as AWS Lambda).
- After that, focus on the documentation's [DynamoDB guide](#). It shows you how to perform the basic DynamoDB activities: create and delete a table, manipulate items, run batch operations, run a query, and perform a scan. Its examples use the **resource** interface. When you see `boto3.resource('dynamodb')` that indicates you're using the higher-level **resource** interface.
- After the guide, you can review the [DynamoDB reference](#). This landing page provides an exhaustive list of the classes and methods available to you. At the top, you'll see the `DynamoDB.Client` class. This provides low-level access to all the control-plane and data-plane operations. At the bottom, look at the `DynamoDB.ServiceResource` class. This is the higher-level Pythonic interface. With it you can create a table, do batch operations across tables, or obtain a `DynamoDB.ServiceResource.Table` instance for table-specific actions.

Understanding the client and resource abstraction layers

The two interfaces you'll be working with are the **client** interface and the **resource** interface.

- The low-level **client** interface provides a 1-to-1 mapping to the underlying service API. Every API offered by DynamoDB is available through the client. This means the client interface can provide complete functionality, but it's often more verbose and complex to use.
- The higher-level **resource** interface does not provide a 1-to-1 mapping of the underlying service API. However, it provides methods that make it more convenient for you to access the service such as `batch_writer`.

Here's an example of inserting an item using the client interface. Notice how all values are passed as a map with the key indicating their type ('S' for string, 'N' for number) and their value as a string. This is known as DynamoDB JSON format.

```
import boto3

dynamodb = boto3.client('dynamodb')

dynamodb.put_item(
    TableName='YourTableName',
    Item={
        'pk': {'S': 'id#1'},
        'sk': {'S': 'cart#123'},
        'name': {'S': 'SomeName'},
        'inventory': {'N': '500'},
        # ... more attributes ...
    }
)
```

Here's the same PutItem operation using the resource interface. The data typing is implicit:

```
import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')

table.put_item(
    Item={
```

```
'pk': 'id#1',
'sk': 'cart#123',
'name': 'SomeName',
'inventory': 500,
# ... more attributes ...
}
)
```

If needed, you can convert between regular JSON and DynamoDB JSON using the `TypeSerializer` and `TypeDeserializer` classes provided with `boto3`:

```
def dynamo_to_python(dynamo_object: dict) -> dict:
    deserializer = TypeDeserializer()
    return {
        k: deserializer.deserialize(v)
        for k, v in dynamo_object.items()
    }

def python_to_dynamo(python_object: dict) -> dict:
    serializer = TypeSerializer()
    return {
        k: serializer.serialize(v)
        for k, v in python_object.items()
    }
```

Here is how to perform a query using the client interface. It expresses the query as a JSON construct. It uses a `KeyConditionExpression` string which requires variable substitution to handle any potential keyword conflicts:

```
import boto3

client = boto3.client('dynamodb')

# Construct the query
response = client.query(
    TableName='YourTableName',
    KeyConditionExpression='pk = :pk_val AND begins_with(sk, :sk_val)',
    FilterExpression='#name = :name_val',
    ExpressionAttributeValues={
        ':pk_val': {'S': 'id#1'},
        ':sk_val': {'S': 'cart#'}
    }
)
```

```
    ':name_val': {'S': 'SomeName'},
},
ExpressionAttributeNames={
    '#name': 'name',
}
)
```

The same query operation using the resource interface can be shortened and simplified:

```
import boto3
from boto3.dynamodb.conditions import Key, Attr

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('YourTableName')

response = table.query(
    KeyConditionExpression=Key('pk').eq('id#1') & Key('sk').begins_with('cart#'),
    FilterExpression=Attr('name').eq('SomeName')
)
```

As a final example, imagine you want to get the approximate size of a table (which is metadata kept on the table that is updated about every 6 hours). With the client interface, you do a `describe_table()` operation and pull the answer from the JSON structure returned:

```
import boto3

dynamodb = boto3.client('dynamodb')

response = dynamodb.describe_table(TableName='YourTableName')
size = response['Table']['TableSizeBytes']
```

With the resource interface, the table performs the describe operation implicitly and presents the data directly as an attribute:

```
import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')
size = table.table_size_bytes
```

Note

When considering whether to develop using the client or resource interface, be aware that new features will not be added to the resource interface per the [resource documentation](#): “The AWS Python SDK team does not intend to add new features to the resources interface in boto3. Existing interfaces will continue to operate during boto3’s lifecycle. Customers can find access to newer service features through the client interface.”

Using the table resource batch_writer

One convenience available only with the higher-level table resource is the `batch_writer`. DynamoDB supports batch write operations allowing up to 25 put or delete operations in one network request. Batching like this improves efficiency by minimizing network round trips.

With the low-level client library, you use the `client.batch_write_item()` operation to run batches. You must manually split your work into batches of 25. After each operation, you also have to request to receive a list of unprocessed items (some of the write operations may succeed while others could fail). You then have to pass those unprocessed items again into a later `batch_write_item()` operation. There's a significant amount of boilerplate code.

The [Table.batch_writer](#) method creates a context manager for writing objects in a batch. It presents an interface where it seems as if you're writing items one at a time, but internally it's buffering and sending the items in batches. It also handles unprocessed item retries implicitly.

```
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')

movies = # long list of movies in {'pk': 'val', 'sk': 'val', etc} format
with table.batch_writer() as writer:
    for movie in movies:
        writer.put_item(Item=movie)
```

Additional code examples that explore the client and resource layers

You can also refer to the following code sample repositories that explore usage of the various functions, using both client and resource:

- [Official AWS single-action code examples.](#)
- [Official AWS scenario-oriented code examples.](#)
- [Community-maintained single-action code examples.](#)

Understanding how the Client and Resource objects interact with sessions and threads

The Resource object is not thread safe and should not be shared across threads or processes. Refer to the [guide on Resource](#) for more details.

The Client object, in contrast, is generally thread safe, except for specific advanced features. Refer to the [guide on Clients](#) for more details.

The Session object is not thread safe. So, each time you make a Client or Resource in a multi-threaded environment you should create a new Session first and then make the Client or Resource from the Session. Refer to the [guide on Sessions](#) for more details.

When you call the `boto3.resource()`, you're implicitly using the default Session. This is convenient for writing single-threaded code. When writing multi-threaded code, you'll want to first construct a new Session for each thread and then retrieve the resource from that Session:

```
# Explicitly create a new Session for this thread
session = boto3.Session()
dynamodb = session.resource('dynamodb')
```

Customizing the Config object

When constructing a Client or Resource object, you can pass optional named parameters to customize behavior. The parameter named `config` unlocks a variety of functionality. It's an instance of `botocore.client.Config` and the [reference documentation for Config](#) shows everything it exposes for you to control. The [guide to Configuration](#) provides a good overview.

Note

You can modify many of these behavioral settings at the Session level, within the AWS configuration file, or as environment variables.

Config for timeouts

One use of a custom config is to adjust networking behaviors:

- **connect_timeout (float or int)** – The time in seconds till a timeout exception is thrown when attempting to make a connection. The default is 60 seconds.
- **read_timeout (float or int)** – The time in seconds till a timeout exception is thrown when attempting to read from a connection. The default is 60 seconds.

Timeouts of 60 seconds are excessive for DynamoDB. It means a transient network glitch will cause a minute's delay for the client before it can try again. The following code shortens the timeouts to a second:

```
import boto3
from botocore.config import Config

my_config = Config(
    connect_timeout = 1.0,
    read_timeout = 1.0
)
dynamodb = boto3.resource('dynamodb', config=my_config)
```

For more discussion about timeouts, see [Tuning AWS Java SDK HTTP request settings for latency-aware DynamoDB applications](#). Note the Java SDK has more timeout configurations than Python.

Config for keep-alive

If you're using botocore 1.27.84 or later, you can also control **TCP Keep-Alive**:

- **tcp_keepalive (bool)** - Enables the TCP Keep-Alive socket option used when creating new connections if set to True (defaults to False). This is only available starting with botocore 1.27.84.

Setting TCP Keep-Alive to True can reduce average latencies. Here's sample code that conditionally sets TCP Keep-Alive to true when you have the right botocore version:

```
import botocore
import boto3
from botocore.config import Config
from distutils.version import LooseVersion
```

```
required_version = "1.27.84"
current_version = botocore.__version__

my_config = Config(
    connect_timeout = 0.5,
    read_timeout = 0.5
)
if LooseVersion(current_version) > LooseVersion(required_version):
    my_config = my_config.merge(Config(tcp_keepalive = True))

dynamodb = boto3.resource('dynamodb', config=my_config)
```

Note

TCP Keep-Alive is different than HTTP Keep-Alive. With TCP Keep-Alive, small packets are sent by the underlying operating system over the socket connection to keep the connection alive and immediately detect any drops. With HTTP Keep-Alive, the web connection built on the underlying socket gets reused. HTTP Keep-Alive is always enabled with boto3.

There's a limit to how long an idle connection can be kept alive. Consider sending periodic requests (say every minute) if you have an idle connection but want the next request to use an already-established connection.

Config for retries

The config also accepts a dictionary called **retries** where you can specify your desired retry behavior. Retries happen within the SDK when the SDK receives an error and the error is of a transient type. If an error is retried internally (and a retry eventually produces a successful response), there's no error seen from the calling code's perspective, just a slightly elevated latency. Here are the values you can specify:

- **max_attempts** – An integer representing the maximum number of retry attempts that will be made on a single request. For example, setting this value to 2 will result in the request being retried at most two times after the initial request. Setting this value to 0 will result in no retries ever being attempted after the initial request.
- **total_max_attempts** – An integer representing the maximum number of total attempts that will be made on a single request. This includes the initial request, so a value of 1 indicates that no requests will be retried. If **total_max_attempts** and **max_attempts** are both

provided, `total_max_attempts` takes precedence. `total_max_attempts` is preferred over `max_attempts` because it maps to the `AWS_MAX_ATTEMPTS` environment variable and the `max_attempts` config file value.

- **mode** – A string representing the type of retry mode botocore should use. Valid values are:
 - **legacy** – The default mode. Waits 50ms the first retry, then uses exponential backoff with a base factor of 2. For DynamoDB, it performs up to 10 total max attempts (unless overridden with the above).

 **Note**

With exponential backoff, the last attempt will wait almost 13 seconds.

- **standard** – Named standard because it's more consistent with other AWS SDKs. Waits a random time from 0ms to 1,000ms for the first retry. If another retry is necessary, it picks another random time from 0ms to 1,000ms and multiplies it by 2. If an additional retry is necessary, it does the same random pick multiplied by 4, and so on. Each wait is capped at 20 seconds. This mode will perform retries on more detected failure conditions than the legacy mode. For DynamoDB, it performs up to 3 total max attempts (unless overridden with the above).
- **adaptive** - An experimental retry mode that includes all the functionality of standard mode but adds automatic client-side throttling. With adaptive rate limiting, SDKs can slow down the rate at which requests are sent to better accommodate the capacity of AWS services. This is a provisional mode whose behavior might change.

An expanded definition of these retry modes can be found in the [guide to retries](#) as well as in the [Retry behavior topic in the SDK reference](#).

Here's an example that explicitly uses the legacy retry policy with a maximum of 3 total requests (2 retries):

```
import boto3
from botocore.config import Config

my_config = Config(
    connect_timeout = 1.0,
    read_timeout = 1.0,
    retries = {
        'mode': 'legacy',
```

```
'total_max_attempts': 3
}
)
dynamodb = boto3.resource('dynamodb', config=my_config)
```

Because DynamoDB is a highly-available, low-latency system, you may want to be more aggressive with the speed of retries than the built-in retry policies allow. You can implement your own retry policy by setting the max attempts to 0, catching the exceptions yourself, and retrying as appropriate from your own code instead of relying on boto3 to do implicit retries.

If you manage your own retry policy, you'll want to differentiate between throttles and errors:

- A **throttle** (indicated by a `ProvisionedThroughputExceededException` or `ThrottlingException`) indicates a healthy service that's informing you that you've exceeded your read or write capacity on a DynamoDB table or partition. Every millisecond that passes, a bit more read or write capacity is made available, so you can retry quickly (such as every 50ms) to attempt to access that newly released capacity. With throttles, you don't especially need exponential backoff because throttles are lightweight for DynamoDB to return and incur no per-request charge to you. Exponential backoff assigns longer delays to client threads that have already waited the longest, which statistically extends the p50 and p99 outward.
- An **error** (indicated by an `InternalServerError` or a `ServiceUnavailable`, among others) indicates a transient issue with the service. This can be for the whole table or possibly just the partition you're reading from or writing to. With errors, you can pause longer before retries (such as 250ms or 500ms) and use jitter to stagger the retries.

Config for max pool connections

Lastly, the config lets you control the connection pool size:

- **max_pool_connections (int)** – The maximum number of connections to keep in a connection pool. If this value is not set, the default value of 10 is used.

This option controls the maximum number of HTTP connections to keep pooled for reuse. A different pool is kept per Session. If you anticipate more than 10 threads going against clients or resources built off the same Session, you should consider raising this, so threads don't have to wait on other threads using a pooled connection.

```
import boto3
```

```
from botocore.config import Config

my_config = Config(
    max_pool_connections = 20
)

# Setup a single session holding up to 20 pooled connections
session = boto3.Session(my_config)

# Create up to 20 resources against that session for handing to threads
# Notice the single-threaded access to the Session and each Resource
resource1 = session.resource('dynamodb')
resource2 = session.resource('dynamodb')
# etc
```

Error handling

AWS service exceptions aren't all statically defined in Boto3. This is because errors and exceptions from AWS services vary widely and are subject to change. Boto3 wraps all service exceptions as a `ClientError` and exposes the details as structured JSON. For example, an error response might be structured like this:

```
{
    'Error': {
        'Code': 'SomeServiceException',
        'Message': 'Details/context around the exception or error'
    },
    'ResponseMetadata': {
        'RequestId': '1234567890ABCDEF',
        'HostId': 'host ID data will appear here as a hash',
        'HTTPStatusCode': 400,
        'HTTPHeaders': {'header metadata key/values will appear here'},
        'RetryAttempts': 0
    }
}
```

The following code catches any `ClientError` exceptions and looks at the string value of the `Code` within the `Error` to determine what action to take:

```
import botocore
import boto3
```

```
dynamodb = boto3.client('dynamodb')

try:
    response = dynamodb.put_item(...)

except botocore.exceptions.ClientError as err:
    print('Error Code: {}'.format(err.response['Error']['Code']))
    print('Error Message: {}'.format(err.response['Error']['Message']))
    print('Http Code: {}'.format(err.response['ResponseMetadata']['HTTPStatusCode']))
    print('Request ID: {}'.format(err.response['ResponseMetadata']['RequestId']))

    if err.response['Error']['Code'] in ('ProvisionedThroughputExceeded', 'ThrottlingException'):
        print("Received a throttle")
    elif err.response['Error']['Code'] == 'InternalServerError':
        print("Received a server error")
    else:
        raise err
```

Some (but not all) exception codes have been materialized as top-level classes. You can choose to handle these directly. When using the Client interface, these exceptions are dynamically populated on your client and you catch these exceptions using your client instance, like this:

```
except ddb_client.exceptions.ProvisionedThroughputExceeded:
```

When using the Resource interface, you have to use `.meta.client` to traverse from the resource to the underlying Client to access the exceptions, like this:

```
except ddb_resource.meta.client.exceptions.ProvisionedThroughputExceeded:
```

To review the list of materialized exception types, you can generate the list dynamically:

```
ddb = boto3.client("dynamodb")
print([e for e in dir(ddb.exceptions) if e.endswith('Exception') or
      e.endswith('Error')])
```

When doing a write operation with a condition expression, you can request that if the expression fails the value of the item should be returned in the error response.

```
try:
    response = table.put_item(
```

```
        Item=item,
        ConditionExpression='attribute_not_exists(pk)',
        ReturnValuesOnConditionCheckFailure='ALL_OLD'
    )
except table.meta.client.exceptions.ConditionalCheckFailedException as e:
    print('Item already exists:', e.response['Item'])
```

For further reading on error handling and exceptions:

- The [boto3 guide on error handling](#) has more information on error handling techniques.
- The [DynamoDB developer guide section on programming errors](#) lists what errors you might encounter.
- The [Common Errors section in the API reference](#).
- The documentation on each API operation lists what errors that call might generate (for example [BatchWriteItem](#)).

Logging

The boto3 library integrates with Python's built-in logging module for tracking what happens during a session. To control logging levels, you can configure the logging module:

```
import logging

logging.basicConfig(level=logging.INFO)
```

This configures the root logger to log INFO and above level messages. Logging messages which are less severe than level will be ignored. Logging levels include DEBUG, INFO, WARNING, ERROR, and CRITICAL. The default is WARNING.

Loggers in boto3 are hierarchical. The library uses a few different loggers, each corresponding to different parts of the library. You can separately control the behavior of each:

- **boto3**: The main logger for the boto3 module.
- **botocore**: The main logger for the botocore package.
- **botocore.auth**: Used for logging AWS signature creation for requests.
- **botocore.credentials**: Used for logging the process of credential fetching and refresh.
- **botocore.endpoint**: Used for logging request creation before it's sent over the network.

- **botocore.hooks**: Used for logging events triggered in the library.
- **botocore.loaders**: Used for logging when parts of AWS service models are loaded.
- **botocore.parsers**: Used for logging AWS service responses before they're parsed.
- **botocore.retryhandler**: Used for logging the processing of AWS service request retries (legacy mode).
- **botocore.retries.standard**: Used for logging the processing of AWS service request retries (standard or adaptive mode).
- **botocore.utils**: Used for logging miscellaneous activities in the library.
- **botocore.waiter**: Used for logging the functionality of waiters, which poll an AWS service until a certain state is reached.

Other libraries log as well. Internally, boto3 uses the third party urllib3 for HTTP connection handling. When latency is important, you can watch its logs to ensure your pool is being well utilized by seeing when urllib3 establishes a new connection or closes an idle one down.

- **urllib3.connectionpool**: Use for logging connection pool handling events.

The following code snippet sets most logging to INFO with DEBUG logging for endpoint and connection pool activity:

```
import logging

logging.getLogger('boto3').setLevel(logging.INFO)
logging.getLogger('botocore').setLevel(logging.INFO)
logging.getLogger('botocore.endpoint').setLevel(logging.DEBUG)
logging.getLogger('urllib3.connectionpool').setLevel(logging.DEBUG)
```

Event hooks

Botocore emits events during various parts of its execution. You can register handlers for these events so that whenever an event is emitted, your handler will be called. This lets you extend the behavior of botocore without having to modify the internals.

For instance, let's say you want to keep track of every time a PutItem operation is called on any DynamoDB table in your application. You might register on the 'provide-client-params.dynamodb.PutItem' event to catch and log every time a PutItem operation is invoked on the associated Session. Here's an example:

```
import boto3
import botocore
import logging

def log_put_params(params, **kwargs):
    if 'TableName' in params and 'Item' in params:
        logging.info(f"PutItem on table {params['TableName']}: {params['Item']}")

logging.basicConfig(level=logging.INFO)

session = boto3.Session()
event_system = session.events

# Register our interest in hooking in when the parameters are provided to PutItem
event_system.register('provide-client-params.dynamodb.PutItem', log_put_params)

# Now, every time you use this session to put an item in DynamoDB,
# it will log the table name and item data.
dynamodb = session.resource('dynamodb')
table = dynamodb.Table('YourTableName')
table.put_item(
    Item={
        'pk': '123',
        'sk': 'cart#123',
        'item_data': 'YourItemData',
        # ... more attributes ...
    }
)
```

Within the handler, you can even manipulate the params programmatically to change behavior:

```
params['TableName'] = "NewTableName"
```

For more information on events, see the [botocore documentation on events](#) and the [boto3 documentation on events](#).

Pagination and the Paginator

Some requests, such as Query and Scan, limit the size of data returned on a single request and require you to make repeated requests to pull subsequent pages.

You can control the maximum number of items to be read for each page with the `limit` parameter. For example, if you want the last 10 items, you can use `limit` to retrieve only the last 10. Note the limit is how much should be read from the table before any filtering is applied. There's no way to specify you want exactly 10 after filtering; you can only control the pre-filtered count and check client-side when you've actually retrieved 10. Regardless of the limit, every response always has a maximum size of 1 MB.

If the response includes a `LastEvaluatedKey`, it indicates the response ended because it hit a count or size limit. The key is the last key evaluated for the response. You can retrieve this `LastEvaluatedKey` and pass it to a follow-up call as `ExclusiveStartKey` to read the next chunk from that starting point. When there's no `LastEvaluatedKey` returned that, means there are no more items matching the Query or Scan.

Here's a simple example (using the Resource interface, but the Client interface has the same pattern) that reads at most 100 items per page and loops until all items have been read.

```
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('YourTableName')

query_params = {
    'KeyConditionExpression': Key('pk').eq('123') & Key('sk').gt(1000),
    'Limit': 100
}

while True:
    response = table.query(**query_params)

    # Process the items however you like
    for item in response['Items']:
        print(item)

    # No LastEvaluatedKey means no more items to retrieve
    if 'LastEvaluatedKey' not in response:
        break

    # If there are possibly more items, update the start key for the next page
    query_params['ExclusiveStartKey'] = response['LastEvaluatedKey']
```

For convenience, boto3 can do this for you with Paginators. However, it only works with the Client interface. Here's the code rewritten to use Paginators:

```
import boto3

dynamodb = boto3.client('dynamodb')

paginator = dynamodb.get_paginator('query')

query_params = {
    'TableName': 'YourTableName',
    'KeyConditionExpression': 'pk = :pk_val AND sk > :sk_val',
    'ExpressionAttributeValues': {
        ':pk_val': {'S': '123'},
        ':sk_val': {'N': '1000'},
    },
    'Limit': 100
}

page_iterator = paginator.paginate(**query_params)

for page in page_iterator:
    # Process the items however you like
    for item in page['Items']:
        print(item)
```

For more information, see the [Guide on Paginators](#) and the [API reference for DynamoDB.Paginator.Query](#).

 **Note**

Paginators also have their own configuration settings named `MaxItems`, `StartingToken`, and `PageSize`. For paginating with DynamoDB, you should ignore these settings.

Waiters

Waiters provide the ability to wait for something to complete before proceeding. At present, they only support waiting for a table to be created or deleted. In the background, the waiter operation does a check for you every 20 seconds up to 25 times. You could do this yourself, but using a waiter is elegant when writing automation.

This code shows how to wait for a particular table to have been created:

```
# Create a table, wait until it exists, and print its ARN
response = client.create_table(...)
waiter = client.get_waiter('table_exists')
waiter.wait(TableName='YourTableName')
print('Table created:', response['TableDescription']['TableArn'])
```

For more information, see the [Guide to Waiters](#) and [Reference on Waiters](#).

Programming Amazon DynamoDB with JavaScript

This guide provides an orientation to programmers wanting to use Amazon DynamoDB with JavaScript. Learn about the AWS SDK for JavaScript, abstraction layers available, configuring connections, handling errors, defining retry policies, managing keep-alive, and more.

Topics

- [About AWS SDK for JavaScript](#)
- [Using the AWS SDK for JavaScript V3](#)
- [Accessing JavaScript documentation](#)
- [Abstraction layers](#)
- [Using the marshall utility function](#)
- [Reading items](#)
- [Conditional writes](#)
- [Pagination](#)
- [Specifying configuration](#)
- [Waiters](#)
- [Error handling](#)
- [Logging](#)
- [Considerations](#)

About AWS SDK for JavaScript

The AWS SDK for JavaScript provides access to AWS services using either browser scripts or Node.js. This documentation focuses on the latest version of the SDK (V3). The AWS SDK for JavaScript V3

is maintained by AWS as an [open-source project hosted on GitHub](#). Issues and feature requests are public and you can access them on the issues page for the GitHub repository.

JavaScript V2 is similar to V3, but contains syntax differences. V3 is more modular, making it easier to ship smaller dependencies, and has first-class TypeScript support. We recommend using the latest version of the SDK.

Using the AWS SDK for JavaScript V3

You can add the SDK to your Node.js application using the Node Package Manager. The examples below show how to add the most common SDK packages for working with DynamoDB.

- `npm install @aws-sdk/client-dynamodb`
- `npm install @aws-sdk/lib-dynamodb`
- `npm install @aws-sdk/util-dynamodb`

Installing packages adds references to the dependency section of your package.json project file. You have the option to use the newer ECMAScript module syntax. For further details on these two approaches, see the Considerations section.

Accessing JavaScript documentation

Get started with JavaScript documentation with the following resources:

- Access the [Developer guide](#) for core JavaScript documentation. Installation instructions are located in the **Setting up** section.
- Access the [API reference](#) documentation to explore all available classes and methods.
- The SDK for JavaScript supports many AWS services other than DynamoDB. Use the following procedure to locate specific API coverage for DynamoDB:
 1. From **Services**, choose **DynamoDB and Libraries**. This documents the low-level client.
 2. Choose **lib-dynamodb**. This documents the high-level client. The two clients represent two different abstraction layers that you have the choice to use. See the section below for more information about abstraction layers.

Abstraction layers

The SDK for JavaScript V3 has a low-level client (`DynamoDBClient`) and a high-level client (`DynamoDBDocumentClient`).

Topics

- [Low-level client \(`DynamoDBClient`\)](#)
- [High-level client \(`DynamoDBDocumentClient`\)](#)

Low-level client (`DynamoDBClient`)

The low-level client provides no extra abstractions over the underlying wire protocol. It gives you full control over all aspects of communication, but because there are no abstractions, you must do things like provide item definitions using the DynamoDB JSON format.

As the example below shows, with this format data types must be stated explicitly. An `S` indicates a string value and an `N` indicates a number value. Numbers on the wire are always sent as strings tagged as number types to ensure no loss in precision. The low-level API calls have a naming pattern such as `PutItemCommand` and `GetItemCommand`.

The following example is using low-level client with Item defined using DynamoDB JSON:

```
const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function addProduct() {
  const params = {
    TableName: "products",
    Item: {
      "id": { S: "Product01" },
      "description": { S: "Hiking Boots" },
      "category": { S: "footwear" },
      "sku": { S: "hiking-sku-01" },
      "size": { N: "9" }
    }
  };
  try {
    const data = await client.send(new PutItemCommand(params));
    console.log('result : ' + JSON.stringify(data));
  }
}
```

```
    } catch (error) {
      console.error("Error:", error);
    }
}
addProduct();
```

High-level client (**DynamoDBDocumentClient**)

The high-level DynamoDB document client offers built-in convenience features, such as eliminating the need to manually marshal data and allowing for direct reads and writes using standard JavaScript objects. The [documentation for lib-dynamodb](#) provides the list of advantages.

To instantiate the `DynamoDBDocumentClient`, construct a low-level `DynamoDBClient` and then wrap it with a `DynamoDBDocumentClient`. The function naming convention differs slightly between the two packages. For instance, the low-level uses `PutItemCommand` while the high-level uses `PutCommand`. The distinct names allow both sets of functions to coexist in the same context, allowing you to mix both in the same script.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, PutCommand } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function addProduct() {
  const params = {
    TableName: "products",
    Item: {
      id: "Product01",
      description: "Hiking Boots",
      category: "footwear",
      sku: "hiking-sku-01",
      size: 9,
    },
  };
}

try {
  const data = await docClient.send(new PutCommand(params));
  console.log('result : ' + JSON.stringify(data));
} catch (error) {
  console.error("Error:", error);
```

```
    }
}

addProduct();
```

The pattern of usage is consistent when you're reading items using API operations such as `GetItem`, `Query`, or `Scan`.

Using the marshall utility function

You can use the low-level client and marshall or unmarshall the data types on your own. The utility package, [util-dynamodb](#), has a `marshall()` utility function that accepts JSON and produces DynamoDB JSON, as well as an `unmarshall()` function, that does the reverse. The following example uses the low-level client with data marshalling handled by the `marshall()` call.

```
const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");
const { marshall } = require("@aws-sdk/util-dynamodb");

const client = new DynamoDBClient({});

async function addProduct() {
  const params = {
    TableName: "products",
    Item: marshall({
      id: "Product01",
      description: "Hiking Boots",
      category: "footwear",
      sku: "hiking-sku-01",
      size: 9,
    }),
  };
  try {
    const data = await client.send(new PutItemCommand(params));
  } catch (error) {
    console.error("Error:", error);
  }
}
addProduct();
```

Reading items

To read a single item from DynamoDB, you use the `GetItem` API operation. Similar to the `PutItem` command, you have the choice to use either the low-level client or the high-level Document client. The example below demonstrates using the high-level Document client to retrieve an item.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, GetCommand } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function getProduct() {
  const params = {
    TableName: "products",
    Key: {
      id: "Product01",
    },
  };
  try {
    const data = await docClient.send(new GetCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

getProduct();
```

Use the `Query` API operation to read multiple items. You can use the low-level client or the Document client. The example below uses the high-level Document client.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  QueryCommand,
} = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});
```

```
const docClient = DynamoDBDocumentClient.from(client);

async function productSearch() {
  const params = {
    TableName: "products",
    IndexName: "GSI1",
    KeyConditionExpression: "#category = :category and begins_with(#sku, :sku)",
    ExpressionAttributeNames: {
      "#category": "category",
      "#sku": "sku",
    },
    ExpressionAttributeValues: {
      ":category": "footwear",
      ":sku": "hiking",
    },
  };
}

try {
  const data = await docClient.send(new QueryCommand(params));
  console.log('result : ' + JSON.stringify(data));
} catch (error) {
  console.error("Error:", error);
}
}

productSearch();
```

Conditional writes

DynamoDB write operations can specify a logical condition expression that must evaluate to true for the write to proceed. If the condition does not evaluate to true, the write operation generates an exception. The condition expression can check if the item already exists or if its attributes match certain constraints.

ConditionExpression = "version = :ver AND size(VideoClip) < :maxsize"

When the conditional expression fails, you can use `ReturnValuesOnConditionCheckFailure` to request that the error response include the item that didn't satisfy the conditions to deduce what the problem was. For more details, see [Handle conditional write errors in high concurrency scenarios with Amazon DynamoDB](#).

```
try {
```

```
const response = await client.send(new PutCommand({
    TableName: "YourTableName",
    Item: item,
    ConditionExpression: "attribute_not_exists(pk)",
    ReturnValuesOnConditionCheckFailure: "ALL_OLD"
}));
} catch (e) {
    if (e.name === 'ConditionalCheckFailedException') {
        console.log('Item already exists:', e.Item);
    } else {
        throw e;
    }
}
```

Additional code examples showing other aspects of JavaScript SDK V3 usage are available in the [JavaScript SDK V3 Documentation](#) and under the [DynamoDB-SDK-Examples GitHub repository](#).

Pagination

Topics

- [Using the paginateScan convenience method](#)

Read requests such as Scan or Query will likely return multiple items in a dataset. If you perform a Scan or Query with a Limit parameter, then once the system has read that many items, a partial response will be sent, and you'll need to paginate to retrieve additional items.

The system will only read a maximum of 1 megabyte of data per request. If you're including a Filter expression, the system will still read a megabyte, at maximum, of data from disk, but will return the items of that megabyte that match the filter. The filter operation could return 0 items for a page, but still require further pagination before the search is exhausted.

You should look for LastEvaluatedKey in the response and using it as the ExclusiveStartKey parameter in a subsequent request to continue data retrieval. This serves as a bookmark as noted in the following example.

 **Note**

The sample passes a null lastEvaluatedKey as the ExclusiveStartKey on the first iteration and this is allowed.

Example using the LastEvaluatedKey:

```
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function paginatedScan() {
  let lastEvaluatedKey;
  let pageCount = 0;

  do {
    const params = {
      TableName: "products",
      ExclusiveStartKey: lastEvaluatedKey,
    };

    const response = await client.send(new ScanCommand(params));
    pageCount++;
    console.log(`Page ${pageCount}, Items:`, response.Items);
    lastEvaluatedKey = response.LastEvaluatedKey;
  } while (lastEvaluatedKey);
}

paginatedScan().catch((err) => {
  console.error(err);
});
```

Using the paginateScan convenience method

The SDK provides convenience methods called `paginateScan` and `paginateQuery` that do this work for you and makes the repeated requests behind the scenes. Specify the max number of items to read per request using the standard `Limit` parameter.

```
const { DynamoDBClient, paginateScan } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function paginatedScanUsingPaginator() {
  const params = {
    TableName: "products",
    Limit: 100
```

```
};

const paginator = paginateScan({client}, params);

let pageCount = 0;

for await (const page of paginator) {
    pageCount++;
    console.log(`Page ${pageCount}, Items:`, page.Items);
}

paginatedScanUsingPaginator().catch((err) => {
    console.error(err);
});
```

 **Note**

Performing full table scans regularly is not a recommended access pattern unless the table is small.

Specifying configuration

Topics

- [Config for timeouts](#)
- [Config for keep-alive](#)
- [Config for retries](#)

When setting up the `DynamoDBClient`, you can specify various configuration overrides by passing a configuration object to the constructor. For example, you can specify the Region to connect to if it's not already known to the calling context or the endpoint URL to use. This is useful if you want to target a DynamoDB Local instance for development purposes.

```
const client = new DynamoDBClient({
    region: "eu-west-1",
    endpoint: "http://localhost:8000",
});
```

Config for timeouts

DynamoDB uses HTTPS for client-server communication. You can control some aspects of the HTTP layer by providing a `NodeHttpHandler` object. For example, you can adjust the key timeout values `connectionTimeout` and `requestTimeout`. The `connectionTimeout` is the maximum duration, in milliseconds, that the client will wait while trying to establish a connection before giving up.

The `requestTimeout` defines how long the client will wait for a response after a request has been sent, also in milliseconds. The defaults for both are zero, meaning the timeout is disabled and there's no limit on how long the client will wait if the response does not arrive. You should set the timeouts to something reasonable so in the event of a network issue the request will error out and a new request can be initiated. For example:

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@smithy/node-http-handler";

const requestHandler = new NodeHttpHandler({
  connectionTimeout: 2000,
  requestTimeout: 2000,
});

const client = new DynamoDBClient({
  requestHandler
});
```

Note

The example provided uses the [Smithy](#) import. Smithy is a language for defining services and SDKs, open-source and maintained by AWS.

In addition to configuring timeout values, you can set the maximum number of sockets, which allows for an increased number of concurrent connections per origin. The developer guide includes [details on configuring the `maxSockets` parameter](#).

Config for keep-alive

When using HTTPS, the first request always takes some back-and-forth communication to establish a secure connection. HTTP Keep-Alive allows subsequent requests to reuse the already-established

connection, making the requests more efficient and lowering latency. HTTP Keep-Alive is enabled by default with JavaScript V3.

There's a limit to how long an idle connection can be kept alive. Consider sending periodic requests, maybe every minute, if you have an idle connection but want the next request to use an already-established connection.

Note

Note that in the older V2 of the SDK, keep-alive was off by default, meaning each connection would get closed immediately after use. If using V2, you can override this setting.

Config for retries

When the SDK receives an error response and the error is resumable as determined by the SDK, such as a throttling exception or a temporary service exception, it will retry again. This happens invisibly to you as the caller, except that you might notice the request took longer to succeed.

The SDK for JavaScript V3 will make 3 total requests, by default, before giving up and passing the error into the calling context. You can adjust the number and frequency of these retries.

The `DynamoDBClient` constructor accepts a `maxAttempts` setting that limits how many attempts will happen. The below example raises the value from the default of 3 to a total of 5. If you set it to 0 or 1, that indicates you don't want any automatic retries and want to handle any resumable errors yourself within your catch block.

```
const client = new DynamoDBClient({  
    maxAttempts: 5,  
});
```

You can also control the timing of the retries with a custom retry strategy. To do this, import the `util-retry` utility package and create a custom backoff function that calculates the wait time between retries based on the current retry count.

The example below says to make a maximum of 5 attempts with delays of 15, 30, 90, and 360 milliseconds should the first attempt fail. The custom backoff function,

`calculateRetryBackoff`, calculates the delays by accepting the retry attempt number (starts with 1 for the first retry) and returns how many milliseconds to wait for that request.

```
const { ConfiguredRetryStrategy } = require("@aws-sdk/util-retry");

const calculateRetryBackoff = (attempt) => {
  const backoffTimes = [15, 30, 90, 360];
  return backoffTimes[attempt - 1] || 0;
};

const client = new DynamoDBClient({
  retryStrategy: new ConfiguredRetryStrategy(
    5, // max attempts.
    calculateRetryBackoff // backoff function.
  ),
});
```

Waiters

The DynamoDB client includes two useful [waiter functions](#) that can be used when creating, modifying, or deleting tables when you want your code to wait to proceed until the table modification has finished. For example, you can deploy a table, call the `waitUntilTableExists` function, and the code will block until the table has been made **ACTIVE**. The waiter internally polls the DynamoDB service with a `describe-table` every 20 seconds.

```
import {waitUntilTableExists, waitUntilTableNotExists} from "@aws-sdk/client-dynamodb";

... <create table details>

const results = await waitUntilTableExists({client: client, maxWaitTime: 180},
  {TableName: "products"});
if (results.state == 'SUCCESS') {
  return results.reason.Table
}
console.error(` ${results.state} ${results.reason}`);
```

The `waitUntilTableExists` feature returns control only when it can perform a `describe-table` command that shows the table status **ACTIVE**. This ensures that you can use `waitUntilTableExists` to wait for the completion of creation, as well as modifications such as adding a GSI index, which may take some time to apply before the table returns to **ACTIVE** status.

Error handling

In the early examples here, we've caught all errors broadly. However, in practical applications, it's important to discern between various error types and implement more precise error handling.

DynamoDB error responses contain metadata, including the name of the error. You can catch errors then match against the possible string names of error conditions to determine how to proceed. For server-side errors, you can leverage the `instanceof` operator with the error types exported by the `@aws-sdk/client-dynamodb` package to manage error handling efficiently.

It's important to note that these errors only manifest after all retries have been exhausted. If an error is retried and is eventually followed by a successful call, from the code's perspective, there's no error just a slightly elevated latency. Retries will show up in Amazon CloudWatch charts as unsuccessful requests, such as throttle or error requests. If the client reaches the maximum retry count, it will give up and generate an exception. This is the client's way of saying it's not going to retry.

Below is a snippet to catch the error and take action based on the type of error that was returned.

```
import {  
    ResourceNotFoundException,  
    ProvisionedThroughputExceededException,  
    DynamoDBServiceException,  
} from "@aws-sdk/client-dynamodb";  
  
try {  
    await client.send(someCommand);  
} catch (e) {  
    if (e instanceof ResourceNotFoundException) {  
        // Handle ResourceNotFoundException  
    } else if (e instanceof ProvisionedThroughputExceededException) {  
        // Handle ProvisionedThroughputExceededException  
    } else if (e instanceof DynamoDBServiceException) {  
        // Handle DynamoDBServiceException  
    } else {  
        // Other errors such as those from the SDK  
        if (e.name === "TimeoutError") {  
            // Handle SDK TimeoutError.  
        } else {  
            // Handle other errors.  
        }  
    }  
}
```

```
}
```

See [the section called “Error handling”](#) for common error strings in the *DynamoDB Developer Guide*. The exact errors possible with any particular API call can be found in the documentation for that API call, such as the [Query API docs](#).

The metadata of errors include additional properties, depending on the error. For a `TimeoutError`, the metadata includes the number of attempts that were made and the `totalRetryDelay`, as shown below.

```
{
  "name": "TimeoutError",
  "$metadata": {
    "attempts": 3,
    "totalRetryDelay": 199
  }
}
```

If you manage your own retry policy, you'll want to differentiate between throttles and errors:

- A **throttle** (indicated by a `ProvisionedThroughputExceeded` or `ThrottlingException`) indicates a healthy service that's informing you that you've exceeded your read or write capacity on a DynamoDB table or partition. Every millisecond that passes, a bit more read or write capacity is made available, and so you can retry quickly, such as every 50ms, to attempt to access that newly released capacity.

With throttles you don't especially need exponential backoff because throttles are lightweight for DynamoDB to return and incur no per-request charge to you. Exponential backoff assigns longer delays to client threads that have already waited the longest, which statistically extends the p50 and p99 outward.

- An **error** (indicated by an `InternalServerError` or a `ServiceUnavailable`, among others) indicates a transient issue with the service, possibly the whole table or just the partition you're reading from or writing to. With errors, you can pause longer before retries, such as 250ms or 500ms, and use jitter to stagger the retries.

Logging

Turn on logging to get more details about what the SDK is doing. You can set a parameter on the `DynamoDBClient` as shown in the example below. More log information will appear in the console

and includes metadata such as the status code and the consumed capacity. If you run the code locally in a terminal window, the logs appear there. If you run the code in AWS Lambda, and you have Amazon CloudWatch logs set up, then the console output will be written there.

```
const client = new DynamoDBClient({  
    logger: console  
});
```

You can also hook into the internal SDK activities and perform custom logging as certain events happen. The example below uses the client's `middlewareStack` to intercept each request as it's being sent from the SDK and logs it as it's happening.

```
const client = new DynamoDBClient({});  
  
client.middlewareStack.add(  
    (next) => async (args) => {  
        console.log("Sending request from AWS SDK", { request: args.request });  
        return next(args);  
    },  
    {  
        step: "build",  
        name: "log-ddb-calls",  
    }  
);
```

The `MiddlewareStack` provides a powerful hook for observing and controlling SDK behavior. See the blog [Introducing Middleware Stack in Modular AWS SDK for JavaScript](#), for more information.

Considerations

When implementing the AWS SDK for JavaScript in your project, here are some further factors to consider.

Module systems

The SDK supports two module systems, CommonJS and ES (ECMAScript). CommonJS uses the `require` function, while ES uses the `import` keyword.

1. **Common JS** – `const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");`

```
2. ES (ECMAScript – import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";
```

The project type dictates the module system to be used and is specified in the type section of your package.json file. The default is CommonJS. Use "type": "module" to indicate an ES project. If you have an existing Node.JS project that uses the CommonJS package format, you can still add functions with the more modern SDK V3 Import syntax by naming your function files with the .mjs extension. This will allow the code file to be treated as ES (ECMAScript).

Asynchronous operations

You'll see many code samples using callbacks and promises to handle the result of DynamoDB operations. With modern JavaScript this complexity is no longer needed and developers can take advantage of the more succinct and readable async/await syntax for asynchronous operations.

Web browser runtime

Web and mobile developers building with React or React Native can use the SDK for JavaScript in their projects. With the earlier V2 of the SDK, web developers would have to load the full SDK into the browser, referencing an SDK image hosted at <https://sdk.amazonaws.com/js/>.

With V3, it's possible to bundle just the required V3 client modules and all required JavaScript functions into a single JavaScript file using Webpack, and add it in a script tag in the <head> of your HTML pages, as explained in the [Getting started in a browser script](#) section of the SDK documentation.

DAX data plane operations

The SDK for JavaScript V3 does not at this time provide support for the Amazon DynamoDB Streams Accelerator (DAX) data plane operations. If you request DAX support, consider using the SDK for JavaScript V2 which supports DAX data plane operations.

Programming Amazon DynamoDB with AWS SDK for Java 2.x

This guide provides an orientation to programmers wanting to use Amazon DynamoDB with Java. This guide covers the different concepts, such as abstraction layers, configuration management, error handling, controlling retry policies, and managing keep-alive.

Topics

- [About AWS SDK for Java 2.x](#)

- [Getting started with AWS SDK for Java 2.x](#)
- [Using the AWS SDK for Java 2.x documentation](#)
- [Supported interfaces](#)
- [Additional code examples](#)
- [Synchronous and asynchronous programming](#)
- [HTTP clients](#)
- [Configuring an HTTP client](#)
- [Error handling](#)
- [AWS request ID](#)
- [Logging](#)
- [Pagination](#)
- [Data class annotations](#)

About AWS SDK for Java 2.x

You can access DynamoDB from Java using the official AWS SDK for Java. The SDK for Java has two versions: 1.x and 2.x. For 1.x the end-of-support was [announced](#) on January 12, 2024. It entered maintenance mode on July 31, 2024 and its end-of-support is due on December 31, 2025. For new development, we highly recommend that you use 2.x, which was first released in 2018. This guide exclusively targets 2.x and focuses only on the parts of the SDK relevant to DynamoDB.

You can find more information about maintenance and support for SDKs in the [AWS SDK and Tools maintenance policy](#) and [AWS SDKs and Tools version support matrix](#) topics of the [AWS SDKs and Tools Reference Guide](#).

The AWS SDK for Java 2.x is a major rewrite of the 1.x code base to support modern Java features, such as the non-blocking I/O introduced in Java 8. SDK for Java 2.x also adds support for pluggable HTTP client implementations to provide more flexibility and configurability for network connections.

A noticeable change between the SDK for Java 1.x and the SDK for Java 2.x is the use of a new package name. The Java 1.x SDK uses the `com.amazonaws` package name, while the Java 2.x SDK uses `software.amazon.awssdk`. Similarly, Maven artifacts for the Java 1.x SDK use the `com.amazonaws` groupId, while Java 2.x SDK artifacts use the `software.amazon.awssdk` groupId.

Important

The AWS SDK for Java 1.x has a DynamoDB package named com.amazonaws.dynamodbv2. The v2 in the package name doesn't indicate it's Java v2. v2 indicates that the package supports the [second version](#) of the DynamoDB low-level API instead of the [original version](#) of the low-level API.

Support for Java versions

The AWS SDK for Java 2.x provides full support for long-term support (LTS) [Java releases](#).

Getting started with AWS SDK for Java 2.x

The following tutorial shows you how to use [Apache Maven](#) for defining dependencies for the SDK for Java 2.x. This tutorial also shows you how to write the code that connects to DynamoDB for listing the available DynamoDB tables. This tutorial is based on the [Get started with the AWS SDK for Java 2.x](#). We've edited this tutorial to make calls to DynamoDB instead of Amazon S3.

Perform the following steps to complete this tutorial:

- [Step 1: Set up for this tutorial](#)
- [Step 2: Create the project](#)
- [Step 3: Write the code](#)
- [Step 4: Build and run the application](#)

Step 1: Set up for this tutorial

Before you begin this tutorial, you need the following:

- Permission to access Amazon DynamoDB.
- A Java development environment that is configured to access AWS services using single sign-on to the AWS IAM Identity Center

Use the instructions in [Setup overview](#) to get set up for this tutorial. After you have [configured your development environment with single sign-on access](#) for the Java SDK and you have an [active AWS access portal session](#), continue with [Step 2](#) of this tutorial.

Step 2: Create the project

To create the project for this tutorial, you run a Maven command that prompts you for input on how to configure the project. After all input is entered and confirmed, Maven finishes building out the project by creating a `pom.xml` file and creating stub Java files.

1. Open a terminal or command prompt window and navigate to a directory of your choice, for example, your Desktop or Home folder.
2. Enter the following command at the terminal and press Enter.

```
mvn archetype:generate \
-DarchetypeGroupId=software.amazon.awssdk \
-DarchetypeArtifactId=archetype-app-quickstart \
-DarchetypeVersion=2.22.0
```

3. Enter the value listed in the second column for each prompt.

Prompt	Value to enter
Define value for property 'service':	dynamodb
Define value for property 'httpClient' :	apache-client
Define value for property 'nativeImage' :	false
Define value for property 'credentialProvider'	identity-center
Define value for property 'groupId':	org.example
Define value for property 'artifactId':	getstarted
Define value for property 'version' 1.0-SNAPSHOT:	<Enter>

Prompt	Value to enter
Define value for property 'package' org.example:	<Enter>

4. After the last value is entered, Maven lists the choices you made. Confirm by entering *Y* or re-enter values by entering *N*.

Maven creates the project folder named `getstarted` based on the `artifactId` value that you entered. Inside the `getstarted` folder, find a `README.md` file that you can review, a `pom.xml` file, and a `src` directory.

Maven builds the following directory tree.

```
getstarted
### README.md
### pom.xml
### src
    ### main
        #   ### java
        #   #   ### org
        #   #       ### example
        #   #           ### App.java
        #   #           ### DependencyFactory.java
        #   #           ### Handler.java
        #   ### resources
        #       ### simplelogger.properties
    ### test
        ### java
            ### org
                ### example
                    ### HandlerTest.java
```

10 directories, 7 files

The following shows the contents of the `pom.xml` project file.

pom.xml

The `dependencyManagement` section contains a dependency to the AWS SDK for Java 2.x and the `dependencies` section has a dependency for Amazon DynamoDB. Specifying these dependencies

forces Maven to include the relevant jar files into your Java class path. By default, the AWS SDK doesn't include all the classes for all AWS services. For DynamoDB, you should have dependency on dynamodb artifact in case you use the low-level interface or dynamodb-enhanced if you use the high-level interface. If you don't include the relevant dependencies, your code will not compile. The project uses Java 1.8 because of the 1.8 value in the maven.compiler.source and maven.compiler.target properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>getstarted</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.shade.plugin.version>3.2.1</maven.shade.plugin.version>
        <maven.compiler.plugin.version>3.6.1</maven.compiler.plugin.version>
        <exec-maven-plugin.version>1.6.0</exec-maven-plugin.version>
        <aws.java.sdk.version>2.22.0</aws.java.sdk.version> <----- SDK version picked up from archetype version.
        <slf4j.version>1.7.28</slf4j.version>
        <junit5.version>5.8.1</junit5.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>software.amazon.awssdk</groupId>
                <artifactId>bom</artifactId>
                <version>${aws.java.sdk.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
```

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>dynamodb</artifactId> <----- DynamoDB dependency
    <exclusions>
        <exclusion>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>netty-nio-client</artifactId>
        </exclusion>
        <exclusion>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>apache-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>sso</artifactId> <----- Required for identity center authentication.
</dependency>

<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>ssooidc</artifactId> <----- Required for identity center authentication.
</dependency>

<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId> <----- HTTP client specified.
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
```

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>${slf4j.version}</version>
</dependency>

<!-- Needed to adapt Apache Commons Logging used by Apache HTTP Client to
Slf4j to avoid
ClassNotFoundException: org.apache.commons.logging.impl.LogFactoryImpl during
runtime -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>${slf4j.version}</version>
</dependency>

<!-- Test Dependencies -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${junit5.version}</version>
    <scope>test</scope>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven.compiler.plugin.version}</version>
        </plugin>
    </plugins>
</build>

</project>
```

Step 3: Write the code

The following code shows the App class created by Maven. The `main` method is the entry point into the application, which creates an instance of the `Handler` class and then calls its `sendRequest` method.

App class

```
package org.example;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class App {
    private static final Logger logger = LoggerFactory.getLogger(App.class);

    public static void main(String... args) {
        logger.info("Application starts");

        Handler handler = new Handler();
        handler.sendRequest();

        logger.info("Application ends");
    }
}
```

The DependencyFactory class created by Maven contains the dynamoDbClient factory method that builds and returns an [DynamoDbClient](#) instance. The DynamoDbClient instance uses an instance of the Apache-based HTTP client. This is because you specified apache-client when Maven prompted you for which HTTP client to use.

The DependencyFactory is shown in the following code.

DependencyFactory class

```
package org.example;

import software.amazon.awssdk.http.apache.ApacheHttpClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

/**
 * The module containing all dependencies required by the {@link Handler}.
 */
public class DependencyFactory {

    private DependencyFactory() {}

    /**
     * @return an instance of DynamoDbClient

```

```
 */
public static DynamoDbClient dynamoDbClient() {
    return DynamoDbClient.builder()
        .httpClientBuilder(ApacheHttpClient.builder())
        .build();
}
```

The `Handler` class contains the main logic of your program. When an instance of `Handler` is created in the `App` class, the `DependencyFactory` furnishes the `DynamoDbClient` service client. Your code uses the `DynamoDbClient` instance to call the DynamoDB service.

Maven generates the following `Handler` class with a *TODO* comment. The next step in the tutorial replaces the *TODO* with code.

Handler class, Maven-generated

```
package org.example;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

public class Handler {
    private final DynamoDbClient dynamoDbClient;

    public Handler() {
        dynamoDbClient = DependencyFactory.dynamoDbClient();
    }

    public void sendRequest() {
        // TODO: invoking the API calls using dynamoDbClient.
    }
}
```

To fill in the logic, replace the entire contents of the `Handler` class with the following code. The `sendRequest` method is filled in and the necessary imports are added.

Handler class, implemented

The following code uses the [DynamoDbClient](#) instance to retrieve a list of existing tables. If tables exist for a given account and Region, the code uses the `Logger` instance to log the names of these tables.

```
package org.example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;

public class Handler {
    private final DynamoDbClient dynamoDbClient;

    public Handler() {
        dynamoDbClient = DependencyFactory.dynamoDbClient();
    }

    public void sendRequest() {
        Logger logger = LoggerFactory.getLogger(Handler.class);

        logger.info("calling the DynamoDB API to get a list of existing tables");
        ListTablesResponse response = dynamoDbClient.listTables();

        if (!response.hasTableNames()) {
            logger.info("No existing tables found for the configured account & region");
        } else {
            response.tableNames().forEach(tableName -> logger.info("Table: " + tableName));
        }
    }
}
```

Step 4: Build and run the application

After the project is created and contains the complete `Handler` class, build and run the application.

1. Make sure that you have an active IAM Identity Center session. To do so, run the AWS Command Line Interface command `aws sts get-caller-identity` and check the response. If you don't have an active session, see [Sign in using the AWS CLI](#) for instructions.
2. Open a terminal or command prompt window and navigate to your project directory `getstarted`.

3. Use the following command to build your project:

```
mvn clean package
```

4. Use the following command to run the application.

```
mvn exec:java -Dexec.mainClass="org.example.App"
```

After you view the file, delete the object, and then delete the bucket.

Success

If your Maven project built and ran without error, then congratulations! You have successfully built your first Java application using the SDK for Java 2.x.

Cleanup

To clean up the resources you created during this tutorial, do the following:

- Delete the project folder `getstarted`.

Using the AWS SDK for Java 2.x documentation

The [AWS SDK for Java 2.x documentation](#) covers all aspects of the SDK across all AWS services. Use the following topics as your starting points:

- [Migrate from version 1.x to 2.x](#) — Includes a detailed explanation of the differences between 1.x and 2.x. This topic also contains instructions about how to use both major versions side-by-side.
- [DynamoDB guide for Java 2.x SDK](#) — Shows you how to perform basic DynamoDB operations: creating a table, manipulating items, and retrieving items. These examples use the low-level interface. Java has several interfaces, as explained in the following section: [Supported interfaces](#).

Tip

We recommend that after going through these topics in the AWS SDK for Java 2.x documentation, you bookmark the [Javadoc documentation](#). It covers all AWS services and will act as your main API reference.

Supported interfaces

The AWS SDK for Java 2.x supports the following interfaces depending on the level of abstraction you want.

Topics in this section

- [Low-level interface](#)
- [High-level interface](#)
- [Document Interface](#)
- [Comparing interfaces with a Query example](#)

Low-level interface

The low-level interface provides a one-to-one mapping to the underlying service API. Every DynamoDB API is available through this interface. This means the low-level interface can provide complete functionality, but it's often more verbose and complex to use. For example, you have to use the `.s()` functions to hold strings and the `.n()` functions to hold numbers. The following example of [PutItem](#) inserts an item using the low-level interface.

```
import org.slf4j.*;
import software.amazon.awssdk.http.crt.AwsCrtHttpClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;

public class PutItem {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbClient DYNAMODB_CLIENT = DynamoDbClient.create();
    private static final Logger LOGGER = LoggerFactory.getLogger(PutItem.class);

    private void putItem() {
        PutItemResponse response = DYNAMODB_CLIENT.putItem(PutItemRequest.builder()
            .item(Map.of(
                "pk", AttributeValue.builder().s("123").build(),
                "sk", AttributeValue.builder().s("cart#123").build(),
                "item_data",
                AttributeValue.builder().s("YourItemData").build(),
            ))
        );
    }
}
```

```
        "inventory", AttributeValue.builder().n("500").build()
        // ... more attributes ...
    ))
    .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
    .tableName("YourTableName")
    .build());
    LOGGER.info("PutItem call consumed [" +
response.consumedCapacity().capacityUnits() + "] Write Capacity Units (WCUs)");
}
}
```

High-level interface

The high-level interface in AWS SDK for Java 2.x is called the DynamoDB enhanced client. This interface provides a more idiomatic code authoring experience.

The enhanced client offers a way to map between client-side data classes and DynamoDB tables designed to store that data. You define the relationships between tables and their corresponding model classes in your code. Then, you can rely on the SDK to manage the data type manipulation. For more information about the enhanced client, see [DynamoDB enhanced client API](#) in the [AWS SDK for Java 2.x documentation](#).

The following example of [PutItem](#) uses the high-level interface. In this example, the `DynamoDbBean` named `YourItem` creates a `TableSchema` allowing its direct use as input for the `putItem()` call.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientPutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourItem> DYNAMODB_TABLE =
ENHANCED_DYNAMODB_CLIENT.table("YourTableName", TableSchema.fromBean(YourItem.class));
    private static final Logger LOGGER = LoggerFactory.getLogger(PutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourItem> response =
DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourItem.class)
```

```
        .item(new YourItem("123", "cart#123", "YourItemData", 500))
        .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
        .build());
    LOGGER.info("PutItem call consumed [" +
response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
}

{@DynamoDbBean
public static class YourItem {

    public YourItem() {}

    public YourItem(String pk, String sk, String itemData, int inventory) {
        this.pk = pk;
        this.sk = sk;
        this.itemData = itemData;
        this.inventory = inventory;
    }

    private String pk;
    private String sk;
    private String itemData;

    private int inventory;

    @DynamoDbPartitionKey
    public void setPk(String pk) {
        this.pk = pk;
    }

    public String getPk() {
        return pk;
    }

    @DynamoDbSortKey
    public void setSk(String sk) {
        this.sk = sk;
    }

    public String getSk() {
        return sk;
    }

    public void setItemData(String itemData) {
```

```
        this.itemData = itemData;
    }

    public String getItemData() {
        return itemData;
    }

    public void setInventory(int inventory) {
        this.inventory = inventory;
    }

    public int getInventory() {
        return inventory;
    }
}
}
```

AWS SDK for Java 1.x has its own high-level interface, which is often referred to by its main class `DynamoDBMapper`. The AWS SDK for Java 2.x is published in a separate package (and Maven artifact) named `software.amazon.awssdk.enhanced.dynamodb`. The Java 2.x SDK is often referred to by its main class `DynamoDbEnhancedClient`.

High-level interface using immutable data classes

The mapping feature of the DynamoDB enhanced client API also works with immutable data classes. An immutable class has only getters and requires a builder class that the SDK uses to create instances of the class. Immutability in Java is a commonly used style that allows developers to create classes that are side-effect free and, therefore, more predictable in their behavior in complex multi-threaded applications. Instead of using the `@DynamoDbBean` annotation as shown in the [High-level interface example](#), immutable classes use the `@DynamoDbImmutable` annotation, which takes the builder class as its input.

The following example takes the builder class `DynamoDbEnhancedClientImmutablePutItem` as input to create a table schema. The example then provides the schema as input for the [PutItem](#) API call.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;
```

```
public class DynamoDbEnhancedClientImmutablePutItem {  
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =  
DynamoDbEnhancedClient.builder().build();  
    private static final DynamoDbTable<YourImmutableItem>  
DYNAMODB_TABLE = ENHANCED_DYNAMODB_CLIENT.table("YourTableName",  
TableSchema.fromImmutableClass(YourImmutableItem.class));  
    private static final Logger LOGGER =  
LoggerFactory.getLogger(DynamoDbEnhancedClientImmutablePutItem.class);  
  
    private void putItem() {  
        PutItemEnhancedResponse<YourImmutableItem> response =  
DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourImmutableItem.class)  
            .item(YourImmutableItem.builder()  
                .pk("123")  
                .sk("cart#123")  
                .itemData("YourItemData")  
                .inventory(500)  
                .build())  
            .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)  
            .build());  
        LOGGER.info("PutItem call consumed [" +  
response.consumedCapacity().capacityUnits() + "] Write Capacity Units (WCU)");  
    }  
}
```

The following example shows the immutable data class.

```
@DynamoDbImmutable(builder = YourImmutableItem.YourImmutableItemBuilder.class)  
class YourImmutableItem {  
    private final String pk;  
    private final String sk;  
    private final String itemData;  
    private final int inventory;  
    public YourImmutableItem(YourImmutableItemBuilder builder) {  
        this.pk = builder.pk;  
        this.sk = builder.sk;  
        this.itemData = builder.itemData;  
        this.inventory = builder.inventory;  
    }  
  
    public static YourImmutableItemBuilder builder() { return new  
YourImmutableItemBuilder(); }
```

```
@DynamoDbPartitionKey
public String getPk() {
    return pk;
}

@dynamoDbSortKey
public String getSk() {
    return sk;
}

public String getItemData() {
    return itemData;
}

public int getInventory() {
    return inventory;
}

static final class YourImmutableItemBuilder {
    private String pk;
    private String sk;
    private String itemData;
    private int inventory;

    private YourImmutableItemBuilder() {}

    public YourImmutableItemBuilder pk(String pk) { this.pk = pk; return this; }
    public YourImmutableItemBuilder sk(String sk) { this.sk = sk; return this; }
    public YourImmutableItemBuilder itemData(String itemData) { this.itemData = itemData; return this; }
    public YourImmutableItemBuilder inventory(int inventory) { this.inventory = inventory; return this; }

    public YourImmutableItem build() { return new YourImmutableItem(this); }
}
}
```

High-level interface using immutable data classes and third-party boilerplate generation libraries

The immutable data classes example mentioned in the previous section require some boilerplate code. For example, the getter and setter logic on the data classes in addition to the Builder classes. Third-party libraries, such as [Project Lombok](#), can help you generate that type of boilerplate code.

The [Use third-party libraries, such as Lombok](#) section in the AWS SDK for Java 2.x documentation introduces the concept of leveraging the Project Lombok library. Reducing most of the boilerplate code helps you limit the amount of code needed for working with immutable data classes and the AWS SDK. This further results in improved productivity and readability of your code.

The following example demonstrates how Project Lombok simplifies the code needed to use the DynamoDB enhanced client API.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientImmutableLombokPutItem {

    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourImmutableLombokItem>
DYNAMODB_TABLE = ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
TableSchema.fromImmutableClass(YourImmutableLombokItem.class));
    private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedClientImmutableLombokPutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourImmutableLombokItem> response =
DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourImmutableLombokItem.clas
            .item(YourImmutableLombokItem.builder()
                .pk("123")
                .sk("cart#123")
                .itemData("YourItemData")
                .inventory(500)
                .build())
            .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
            .build());
        LOGGER.info("PutItem call consumed [" +
response.consumedCapacity().capacityUnits() + "] Write Capacity Units (WCU)");
    }
}
```

The following example shows the immutable data object of the immutable data class.

```
import lombok.*;
```

```
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;

@Builder
@DynamoDbImmutable(builder =
    YourImmutableLombokItem.YourImmutableLombokItemBuilder.class)
@Value
public class YourImmutableLombokItem {

    @Getter(onMethod_=@DynamoDbPartitionKey)
    String pk;
    @Getter(onMethod_=@DynamoDbSortKey)
    String sk;
    String itemData;
    int inventory;
}
```

The `YourImmutableLombokItem` class uses the following annotations provided by Project Lombok and the AWS SDK:

- [@Builder](#) — Produces complex builder APIs for data classes provided by Project Lombok.
- [@DynamoDbImmutable](#) — Identifies the `DynamoDbImmutable` class as a DynamoDB mappable entity annotation provided by the AWS SDK.
- [@Value](#) — The immutable variant of `@Data`; all fields are made private and final by default, and setters are not generated. This annotation is provided by Project Lombok.

Document Interface

The Document interface avoids the need to specify data type descriptors. The data types are implied by the semantics of the data itself. If you're familiar with the AWS SDK for Java 1.x Document interface, the Document interface in AWS SDK for Java 2.x offers a similar but redesigned interface.

The following [Document interface example](#) shows the `PutItem` call expressed using the Document interface. The example also uses `EnhancedDocument`. To execute commands against a DynamoDB table using the enhanced document API, you must first associate the table with your document table schema to create a `DynamoDBTable` resource object. The Document table schema builder requires the primary index key and attribute converter providers.

You can use `AttributeConverterProvider.defaultProvider()` to convert document attributes of default types. You can change the overall default behavior with a custom

AttributeConverterProvider implementation. You can also change the converter for a single attribute. The [AWS SDK documentation](#) provides more details and examples about how to use custom converters. Their primary use is for attributes of your domain classes that don't have a default converter available. Using a custom converter, you can provide the SDK with the needed information to write or read to DynamoDB.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.document EnhancedDocument;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedDocumentClientPutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<EnhancedDocument> DYNAMODB_TABLE =
        ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
TableSchema.documentSchemaBuilder()

.addIndexPartitionKey(TableMetadata.primaryIndexName(),"pk", AttributeValueType.S)
        .addIndexSortKey(TableMetadata.primaryIndexName(), "sk",
AttributeValueType.S)

.attributeConverterProviders(AttributeConverterProvider.defaultProvider())
        .build());

    private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedDocumentClientPutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<EnhancedDocument> response =
DYNAMODB_TABLE.putItemWithResponse(
            PutItemEnhancedRequest.builder(EnhancedDocument.class)
                .item(
                    EnhancedDocument.builder()

.attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                .putString("pk", "123")
                .putString("sk", "cart#123")
                .putString("item_data", "YourItemData")
                .putNumber("inventory", 500)
                .build())
        .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
```

```
        .build());
    LOGGER.info("PutItem call consumed [" +
response.consumedCapacity().capacityUnits() + "] Write Capacity Units (WCUs)");
}
}
```

Using the following utility methods you can convert JSON documents to and from the native Amazon DynamoDB data types:

- [EnhancedDocument.fromJson\(String json\)](#)—Creates a new EnhancedDocument instance from a JSON string.
- [EnhancedDocument.toJson\(\)](#)—Creates a JSON string representation of the document which allows you to use it in your application like any other JSON object.

Comparing interfaces with a Query example

This section shows the same [Query](#) call expressed using the various interfaces. These queries use a couple of attributes to fine tune the query results:

- You must specify the partition key completely because DynamoDB will target to one specific partition key value.
- The sort key has a key condition expression using `begins_with` so that only cart items are targeted with this query.
- We limit the query to return maximum of 100 item by using `limit()`.
- We set the `scanIndexForward` to false. The results are returned in order of UTF-8 bytes, which usually means the cart item with the lowest number is returned first. By setting the `scanIndexForward` to false we reverse the order and the cart item with the highest number is returned first.
- We apply a filter to remove any result that does not match the criteria. The data being filtered consumes read capacity whether or not the item matches the filter.

Example Query using low-level interface

The following example queries a table named `YourTableName` using a `keyConditionExpression`, which limits the query to a specific partition key value and sort key value that begin with a specific prefix value. These key conditions limit the amount of data read

from DynamoDB. Finally, the query applies a filter on the data retrieved from DynamoDB using a `filterExpression`.

```
import org.slf4j.*;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;

public class Query {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbClient DYNAMODB_CLIENT =
DynamoDbClient.builder().build();
    private static final Logger LOGGER = LoggerFactory.getLogger(Query.class);

    private static void query() {
        QueryResponse response = DYNAMODB_CLIENT.query(QueryRequest.builder()
            .expressionAttributeNames(Map.of("#name", "name"))
            .expressionAttributeValues(Map.of(
                ":pk_val", AttributeValue.fromS("id#1"),
                ":sk_val", AttributeValue.fromS("cart#"),
                ":name_val", AttributeValue.fromS("SomeName")))
            .filterExpression("#name = :name_val")
            .keyConditionExpression("pk = :pk_val AND begins_with(sk, :sk_val)")
            .limit(100)
            .scanIndexForward(false)
            .tableName("YourTableName")
            .build());

        LOGGER.info("nr of items: " + response.count());
        LOGGER.info("First item pk: " + response.items().get(0).get("pk"));
        LOGGER.info("First item sk: " + response.items().get(0).get("sk"));
    }
}
```

Example Query using Document interface

The following example queries a table named `YourTableName` using document interface.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.document EnhancedDocument;
import software.amazon.awssdk.enhanced.dynamodb.model.*;

import java.util.Map;

public class DynamoDbEnhancedDocumentClientQuery {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<EnhancedDocument> DYNAMODB_TABLE =
        ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
TableSchema.documentSchemaBuilder()
        .addIndexPartitionKey(TableMetadata.primaryIndexName(),"pk",
AttributeValueType.S)
        .addIndexSortKey(TableMetadata.primaryIndexName(), "sk",
AttributeValueType.S)

.attributeConverterProviders(AttributeConverterProvider.defaultProvider())
        .build());
    private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedDocumentClientQuery.class);

    private void query() {
        PageIterable<EnhancedDocument> response =
DYNAMODB_TABLE.query(QueryEnhancedRequest.builder()
        .filterExpression(Expression.builder()
            .expression("#name = :name_val")
            .expressionNames(Map.of("#name", "name"))
            .expressionValues(Map.of(":name_val",
AttributeValue.fromS("SomeName"))))
        .build())
        .limit(100)
        .queryConditional(QueryConditional.sortBeginsWith(Key.builder()
            .partitionValue("id#1")
            .sortValue("cart#")
            .build())))
        .scanIndexForward(false)
        .build());

        LOGGER.info("nr of items: " + response.items().stream().count());
    }
}
```

```
        LOGGER.info("First item pk: " +
response.items().iterator().next().getString("pk"));
        LOGGER.info("First item sk: " +
response.items().iterator().next().getString("sk"));

    }
}
```

Example Query using high-level interface

The following example queries a table named `YourTableName` using the DynamoDB enhanced client API.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;

import java.util.Map;

public class DynamoDbEnhancedClientQuery {

    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourItem> DYNAMODB_TABLE =
ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
TableSchema.fromBean(DynamoDbEnhancedClientQuery.YourItem.class));
    private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedClientQuery.class);

    private void query() {
        PageIterable<YourItem> response =
DYNAMODB_TABLE.query(QueryEnhancedRequest.builder()
            .filterExpression(Expression.builder()
                .expression("#name = :name_val")
                .expressionNames(Map.of("#name", "name"))
                .expressionValues(Map.of(":name_val",
AttributeValue.fromString("SomeName"))))
            .build())
            .limit(100)
            .queryConditional(QueryConditional.sortBeginsWith(Key.builder()
                .partitionValue("id#1")
                .build())))
            .build());
    }
}
```

```
        .sortValue("cart#")
        .build()))
.scanIndexForward(false)
.build());
```

```
LOGGER.info("nr of items: " + response.items().stream().count());
LOGGER.info("First item pk: " + response.items().iterator().next().getPk());
LOGGER.info("First item sk: " + response.items().iterator().next().getSk());
}
```

```
@DynamoDbBean
public static class YourItem {

    public YourItem() {}

    public YourItem(String pk, String sk, String name) {
        this.pk = pk;
        this.sk = sk;
        this.name = name;
    }

    private String pk;
    private String sk;
    private String name;

    @DynamoDbPartitionKey
    public void setPk(String pk) {
        this.pk = pk;
    }

    public String getPk() {
        return pk;
    }

    @DynamoDbSortKey
    public void setSk(String sk) {
        this.sk = sk;
    }

    public String getSk() {
        return sk;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

}
```

High-level interface using immutable data classes

When you perform a Query with the high-level immutable data classes, the code is the same as the high-level interface example except for the construction of the entity class YourItem or YourImmutableItem. For more information, see the [PutItem](#) example.

High-level interface using immutable data classes and third-party boilerplate generation libraries

When you perform a Query with the high-level immutable data classes, the code is the same as the high-level interface example except for the construction of the entity class YourItem or YourImmutableLombokItem. For more information, see the [PutItem](#) example.

Additional code examples

You can also refer to the following code sample repositories that provide additional examples to use DynamoDB with SDK for Java 2.x:

- [Official AWS single-action code examples](#)
- [Community-maintained single-action code examples](#)
- [Official AWS scenario-oriented code examples](#)

Synchronous and asynchronous programming

The AWS SDK for Java 2.x provides both *synchronous* and *asynchronous* clients for AWS services, such as DynamoDB.

The `DynamoDbClient` and `DynamoDbEnhancedClient` classes provide synchronous methods that block your thread's execution until the client receives a response from the service. This client is the most straightforward way of interacting with DynamoDB if you have no need for asynchronous operations.

The `DynamoDbAsyncClient` and `DynamoDbEnhancedAsyncClient` classes provide asynchronous methods that return immediately, and give control back to the calling thread without waiting for a response. The non-blocking client has an advantage that it allows for high concurrency across a few threads, which provides efficient handling of I/O requests with minimal compute resources. This improves throughput and responsiveness.

The AWS SDK for Java 2.x uses the native support for non-blocking I/O. The AWS SDK for Java 1.x had to simulate non-blocking I/O.

The synchronous methods return before a response is available, so you need a way to get the response when it's ready. The asynchronous methods in the AWS SDK for Java return a [CompletableFuture](#) object that contains the results of the asynchronous operation in the future. When you call `get()` or `join()` on these `CompletableFuture` objects, your code will block till the result is available. If you make this call at the same time as making the request, it's similar behavior to a plain synchronous call.

The AWS SDK documentation for [asynchronous programming](#) provides more details about how to leverage this asynchronous style.

HTTP clients

For supporting every client, there exists an HTTP client that handles communication with the AWS services. You can plug in alternative HTTP clients, choosing one that has the characteristics that best fit your application. Some are more lightweight; some have more configuration options.

Some HTTP clients support only synchronous use, while others support only asynchronous use. The AWS SDK documentation provides a [flowchart](#) that helps you select the optimal HTTP client for your workload.

The following list presents some of the possible HTTP clients:

Topics

- [Apache-based HTTP client](#)
- [URLConnection-based HTTP client](#)
- [Netty-based HTTP client](#)
- [AWS CRT-based HTTP client](#)

Apache-based HTTP client

[ApacheHttpClient](#) supports synchronous service clients and is the default HTTP client for synchronous use. For information about configuring the ApacheHttpClient, see [Configure the Apache-based HTTP client](#) in the AWS SDK for Java 2.x documentation.

URLConnection-based HTTP client

The [URLConnectionHttpClient](#) is another option for synchronous clients. It loads more quickly than the Apache-based HTTP client, but has fewer features. For information about configuring the [URLConnectionHttpClient](#), see [Configure the URLConnection-based HTTP client](#).

Netty-based HTTP client

[NettyNioAsyncHttpClient](#) supports async clients and is the default choice for async use. For information about configuring the [NettyNioAsyncHttpClient](#), see [Configure the Netty-based HTTP client](#).

AWS CRT-based HTTP client

The newer [AwsCrtHttpClient](#) and [AwsCrtAsyncHttpClient](#) from the AWS Common Runtime (CRT) libraries is another option that supports both synchronous and asynchronous clients.

Compared to other HTTP clients, AWS CRT offers:

- Faster SDK startup time
- Smaller memory footprint
- Reduced latency time
- Connection health management
- DNS load balancing

For information about configuring [AwsCrtHttpClient](#) and [AwsCrtAsyncHttpClient](#), see [Configure the AWS CRT-based HTTP clients](#).

AWS CRT-based HTTP client isn't the default because that would break backward compatibility for existing applications. However, for DynamoDB we recommend that you use the AWS CRT-based HTTP client for both sync and async uses.

For an introduction about the AWS CRT-based HTTP client, see [Announcing availability of the AWS CRT HTTP Client in the AWS SDK for Java 2.x](#).

Configuring an HTTP client

When configuring a client, you can provide various configuration options, including:

- Setting timeouts for different aspects of API calls
- Controlling if TCP Keep-Alive is enabled
- Controlling the retry policy when encountering errors
- Specifying execution attributes that [Execution interceptor](#) instances can modify. Execution interceptors can write code that intercept the execution of your API requests and responses. This enables you to perform tasks, such as publish metrics and modify requests in-flight.
- Adding or manipulating HTTP headers
- Enabling the tracking of [client-side performance metrics](#). Using this feature helps you to collect metrics about the service clients in your application and analyze the output in Amazon CloudWatch.
- Specifying an alternate executor service to be used for scheduling tasks, such as async retry attempts and timeout tasks

You control the configuration by providing a [ClientOverrideConfiguration](#) object to the service client `Builder` class. You'll see this in some code examples in the following sections.

The `ClientOverrideConfiguration` provides standard configuration choices. The different pluggable HTTP clients have implementation-specific configuration possibilities as well. Each of these clients also maintains its own documentation:

- [Apache-based HTTP client](#)
- [URLConnection-based HTTP client](#)
- [Netty-based HTTP client](#)
- [AWS CRT-based HTTP clients](#)

Topics in this section

- [Timeout configuration](#)
- [RetryMode](#)
- [DefaultsMode](#)
- [Keep-Alive configuration](#)

Timeout configuration

You can adjust the client configuration to control various timeouts related to the service calls. DynamoDB provides lower latencies compared to other AWS services. Therefore, you might want to adjust these properties to lower timeout values so you can fail fast if there's a networking issue.

You can customize the latency related behavior using `ClientOverrideConfiguration` on the DynamoDB client or by changing detailed configuration options on the underlying HTTP client implementation.

You can configure the following impactful properties using `ClientOverrideConfiguration`:

- `apiCallAttemptTimeout`—The amount of time to wait for a single attempt for an HTTP request to complete before giving up and timing out.
- `apiCallTimeout`—The amount of time to allow the client to complete the execution of an API call. This includes the request handler execution that consists of all the HTTP requests including retries.

The AWS SDK for Java 2.x provides [default values](#) for some timeout options, such as connection timeout and socket timeouts. The SDK doesn't provide default values for API call timeouts or individual API call attempt timeouts. If these timeouts aren't set in the `ClientOverrideConfiguration`, the SDK will use the socket timeout value instead for the overall API call timeout. The socket timeout has a default value of 30 seconds.

RetryMode

Another configuration related to the timeout configuration that you should consider is the `RetryMode` configuration object. This configuration object contains a collection of retry behaviors.

The SDK for Java 2.x supports the following retry modes:

- `legacy` — The default retry mode if you don't explicitly change it. This retry mode is specific to the Java SDK and characterized by:
 - Up to three retries, or more for services, such as DynamoDB which has up to 8 retries.
- `standard` — Named standard because it's more consistent with other AWS SDKs. This mode waits for a random amount of time ranging from 0ms to 1,000ms for the first retry. If another retry is necessary, it picks another random time from 0ms to 1,000ms and multiplies it by two. If an additional retry is necessary, it does the same random pick multiplied by 4, and so on. Each wait is capped at 20 seconds. This mode will perform retries on more detected failure conditions

than the legacy mode. For DynamoDB, it performs up to three total max attempts unless you overridde with [numRetries](#).

- adaptive — Builds on standard mode and dynamically limits the rate of AWS requests to maximize success rate. This may done be at the expense of request latency. We do not recommend adaptive retry mode when predictable latency is important.

You can find an expanded definition of these retry modes in the [Retry behavior](#) topic in the *AWS SDKs and Tools Reference Guide*.

Retry policies

All RetryMode configurations have a [RetryPolicy](#), which is built based on one or more [RetryCondition](#) configurations. The [TokenBucketRetryCondition](#) is especially important to the retry behaviour of the DynamoDB SDK client implementation. This condition limits the number of retries made by the SDK using a token bucket algorithm. Depending on the selected retry mode, throttling exceptions may or may not subtract tokens from the TokenBucket.

When a client encounters a retryable error, such as a throttling exception or a temporary server error, the SDK will automatically retry the request. You can control how many times and how quickly these retries happen.

When configuring a client, you can provide a [RetryPolicy](#) that supports the following parameters:

- numRetries—The maximum number of retries that should be applied before a request is considered to be failed. Default value is 8 regardless of the retry mode you use.

Warning

Make sure that you change this default value after due consideration.

- backoffStrategy—The [BackoffStrategy](#) to apply to the retries, with [FullJitterBackoffStrategy](#) being the default strategy. This strategy performs an exponential delay between additional retries based on the current number of retries, a base delay, and a maximum backoff time. It then adds jitter to provide a bit of randomness. The base delay used in the exponential delay is 25 ms regardless of the retry mode.
- retryCondition—The [RetryCondition](#) determines whether a request should be retried at all. By default it will retry a specific set of HTTP status codes and exceptions that it believes are retryable. For most situations, the default configuration should be sufficient.

The following code provides an alternative retry policy. It specifies to allow a total of five retries (six total requests). The first retry should come after approximately a 100ms delay, with each additional retry doubling that time exponentially, up to a maximum of one second delay.

```
DynamoDbClient client = DynamoDbClient.builder()
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .retryPolicy(RetryPolicy.builder()
            .backoffStrategy(FullJitterBackoffStrategy.builder()
                .baseDelay(Duration.ofMillis(100))
                .maxBackoffTime(Duration.ofSeconds(1))
                .build())
            .numRetries(5)
            .build())
        .build())
    .build();
```

DefaultsMode

The timeout properties not managed by `ClientOverrideConfiguration` and the `RetryMode` are typically not configured explicitly. Instead, their configuration is set implicitly by specifying a `DefaultsMode`.

Support for `DefaultsMode` was introduced in AWS SDK for Java 2.x (version 2.17.102 or later) to provide a set of default values for common configurable settings, such as HTTP communication settings, retry behavior, service Regional endpoint settings, and, potentially, any SDK-related configuration. When you use this feature, you can get new configuration defaults tailored to common usage scenarios.

The default modes are standardized across all of the AWS SDKs. The SDK for Java 2.x supports the following default modes:

- `legacy` — Provides default settings that vary by SDK and existed before `DefaultsMode` was established.
- `standard` — Provides default non-optimized settings for most scenarios.
- `in-region` — Builds on the standard mode and includes settings tailored for applications that call AWS services from within the same AWS Region.
- `cross-region` — Builds on the standard mode and includes settings with high timeouts for applications that call AWS services in a different AWS Region.

- **mobile** — Builds on the standard mode and includes settings with high timeouts tailored for mobile applications with higher latencies.
- **auto** — Builds on the standard mode and includes experimental features. The SDK attempts to discover the runtime environment to determine the appropriate settings automatically. The auto-detection is heuristics-based and does not provide 100% accuracy. If the runtime environment can't be determined, standard mode is used. The auto-detection might query [Instance metadata and user data](#), which might introduce latency. If startup latency is critical to your application, we recommend choosing an explicit `DefaultsMode` instead.

You can configure the defaults mode in the following ways:

- Directly on a client through `AwsClientBuilder.Builder#defaultsMode(DefaultsMode)`.
- On a configuration profile through the `defaults_mode` profile file property.
- Globally through the `aws.defaultsMode` system property.
- Globally through the `AWS_DEFAULTS_MODE` environment variable.

 **Note**

For any mode other than legacy, the vended default values might change as best practices evolve. Therefore, we encourage you to perform testing when upgrading the SDK if you are using a mode other than legacy.

The [Smart configuration defaults](#) in the *AWS SDKs and Tools Reference Guide* provides a list of configuration properties and their default values in the different default modes.

You choose the defaults mode value based on your application's characteristics and the AWS service your application interacts with.

These values are configured with a broad selection of AWS services in mind. For a typical DynamoDB deployment where both your DynamoDB table(s) and application are deployed in one Region, the `in-region` defaults mode is most relevant among the standard default modes.

Example DynamoDB SDK client configuration tuned for low-latency calls

The following example adjusts the timeouts to lower values for an expected low-latency DynamoDB call.

```
DynamoDbAsyncClient asyncClient = DynamoDbAsyncClient.builder()
    .defaultsMode(DefaultsMode.IN_REGION)
    .httpClientBuilder(AwsCrtAsyncHttpClient.builder())
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ofSeconds(3))
        .apiCallAttemptTimeout(Duration.ofMillis(500)))
    .build())
.build();
```

The individual HTTP client implementation may provide you with even more granular control over the timeout and connection usage behavior. For example, in case of the AWS CRT-based client, you can enable `ConnectionHealthConfiguration` which allows the AWS CRT-based client to actively monitor the health of the used connections. For more details, see the [documentation](#) for AWS SDK for Java 2.x.

Keep-Alive configuration

Enabling keep-alive can reduce latencies by reusing connections. There are two different kinds of keep-alive: HTTP Keep-Alive and TCP Keep-Alive.

- HTTP Keep-Alive attempts to maintain the HTTPS connection between the client and server so later requests can reuse that connection. This skips the heavyweight HTTPS authentication on later requests. HTTP Keep-Alive is enabled by default on all clients.
- TCP Keep-Alive requests the underlying operating system to send small packets over the socket connection to provide extra assurance the socket is kept alive and to immediately detect any drops. This ensures a later request won't spend time trying to use a dropped socket. TCP Keep-Alive is disabled by default on all clients. You may want to enable it. The following code examples show how to do this with each HTTP client. When enabled for all non-CRT based HTTP clients, the actual Keep-Alive mechanism is dependent on the operating system. Therefore, you must configure additional TCP Keep-Alive values, such as timeout and number of packets, through the operating system. You can do this using `sysctl` on a Linux or Mac machine, or Registry values on a Windows machine.

Example to enable TCP Keep-Alive on an Apache-based HTTP client

```
DynamoDbClient client = DynamoDbClient.builder()
    .httpClientBuilder(ApacheHttpClient.builder().tcpKeepAlive(true))
    .build();
```

URLConnection-based HTTP client

Any synchronous client using the URLConnection-based HTTP client [HttpURLConnection](#) doesn't have a [mechanism](#) to enable keep-alive.

Example to enable TCP Keep-Alive on a Netty-based HTTP client

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .httpClientBuilder(NettyNioAsyncHttpClient.builder().tcpKeepAlive(true))
    .build();
```

Example to enable TCP Keep-Alive on an AWS CRT-based HTTP client

With the AWS CRT-based HTTP client, you can enable TCP keep-alive and control the duration.

```
DynamoDbClient client = DynamoDbClient.builder()
    .httpClientBuilder(AwsCrtHttpClient.builder())
    .tcpKeepAliveConfiguration(TcpKeepAliveConfiguration.builder()
        .keepAliveInterval(Duration.ofSeconds(50))
        .keepAliveTimeout(Duration.ofSeconds(5))
        .build()))
    .build();
```

When using the asynchronous DynamoDB client, you can enable TCP Keep-Alive as shown in the following code.

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .httpClientBuilder(AwsCrtAsyncHttpClient.builder())
    .tcpKeepAliveConfiguration(TcpKeepAliveConfiguration.builder()
        .keepAliveInterval(Duration.ofSeconds(50))
        .keepAliveTimeout(Duration.ofSeconds(5))
        .build()))
    .build();
```

Error handling

When it comes to exception handling, the AWS SDK for Java 2.x uses runtime (unchecked) exceptions.

The base exception, covering all SDK exceptions, is [SdkServiceException](#), which extends from the Java unchecked RuntimeException. If you catch this, you'll catch all exceptions thrown by the SDK.

SdkServiceException has a subclass called [AwsServiceException](#). This subclass indicates any issue in communication with the AWS service. It has a subclass called [DynamoDbException](#), which indicates an issue in communication with DynamoDB. If you catch this, you'll catch all exceptions related to DynamoDB but no other SDK exceptions.

There are more specific [exception types](#) under DynamoDbException. Some of these exception types apply to control-plane operations such as [TableAlreadyExistsException](#). Others apply to data-plane operations. The following is an example of a common data-plane exception:

- [ConditionalCheckFailedException](#)—You specified a condition in the request that evaluated to false. For example, you might have tried to perform a conditional update on an item, but the actual value of the attribute did not match the expected value in the condition. A request that fails in this manner will not be retried.

Other situations don't have a specific exception defined. For example, when your requests get throttled the specific ProvisionedThroughputExceededException might get thrown, while in other cases the more generic DynamoDbException is thrown. In either case you can determine if the exception was caused by throttling by checking if `isThrottlingException()` returns true.

Depending on your application needs, you can catch all AwsServiceException or DynamoDbException instances. But often you need different behavior in different situations. The logic to deal with a condition check failure will be different compared to dealing with throttling. Define which exceptional paths you want to deal with and make sure to test the alternative paths. This helps you make sure you're able to deal with all relevant scenarios.

The [Error handling with DynamoDB](#) lists some common errors you might encounter. You can also refer to [Common Errors](#) to see a list of common errors. For a particular API call, you can also find the exact errors possible along with its documentation, such as the [Query API](#). For information about handling exceptions, see [Exception handling for the AWS SDK for Java 2.x](#).

AWS request ID

Each request includes a Request ID which can be useful to pull if you're working with AWS Support to diagnose an issue. Each exception derived from SdkServiceException has a method [requestId\(\)](#) available to retrieve the request ID.

Logging

Using the logging provided by the SDK can be useful both for catching any important messages from the client libraries and for more in-depth debugging purposes. Loggers are hierarchical and the SDK uses `software.amazon.awssdk` as its root logger. The level may be configured with one of TRACE, DEBUG, INFO, WARN, ERROR, ALL, or OFF. The configured level will apply to that logger and down into the logger hierarchy.

The AWS SDK for Java 2.x leverages the Simple Logging Façade for Java (SLF4J) for its logging, which acts as an abstraction layer around other loggers. This enables you to plug in the logger that you prefer. For instructions about plugging in loggers, see the [SLF4J user manual](#).

Each logger has a particular behavior. The Log4j 2.x logger by default creates a `ConsoleAppender` which appends log events to `System.out` and defaults to the `ERROR` log level.

The `SimpleLogger` logger included in SLF4J outputs by default to `System.err` and defaults to the `INFO` log level.

We recommend that you set the level to `WARN` for `software.amazon.awssdk` for any production deployments to catch any important messages from the SDK's client libraries while limiting the output quantity.

If SLF4J can't find a supported logger on the class path (no SLF4J binding), SLF4J will default to a [no operation implementation](#). This implementation results in logging messages to `System.err` explaining that SLF4J could not find a logger implementation on the classpath. To prevent this situation, you need to add a logger implementation. To do this, you can add a dependency in your Apache Maven `pom.xml` on artifacts, such as `org.slf4j.slf4j-simple` or `org.apache.logging.log4j.log4j-slf4j2-imp`.

The documentation for AWS SDK for Java 2.x explains how to configure the logging in the SDK including adding logging dependencies to your application configuration. For information, see [Logging with the SDK for Java 2.x](#).

The following configuration in the `Log4j2.xml` file shows how to adjust the logging behavior if you're using the Apache Log4j 2 logger. This configuration sets the root logger level to `WARN`. This log level will be inherited by all loggers in the hierarchy including the `software.amazon.awssdk` logger.

By default, the output will go to `System.out`. In the following example, we still override the default output Log4j appender to apply a tailored Log4j `PatternLayout`.

Example of Log4j2.xml configuration file

The following configuration will log messages at the ERROR and WARN levels to the console for all logger hierarchies.

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="WARN">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

AWS request ID logging

You can find RequestIds within exceptions when something goes wrong. However, if you want the RequestIds for requests that aren't generating exceptions, you can use logging.

Request IDs are outputted at DEBUG level by the `software.amazon.awssdk.request` logger. The following example extends the previous [configuration example](#) to keep the root logger level at ERROR, the `software.amazon.awssdk` at level WARN, and the `software.amazon.awssdk.request` at level DEBUG. Setting these levels helps to catch the Request IDs and other request related details, such as the endpoint and status code.

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="ERROR">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
  </Loggers>
</Configuration>
```

```
<Logger name="software.amazon.awssdk.request" level="DEBUG" />
</Loggers>
</Configuration>
```

Here is an example of the log output:

```
2022-09-23 16:02:08 [main] DEBUG software.amazon.awssdk.request:85 - Sending Request:
DefaultSdkHttpFullRequest(httpMethod=POST, protocol=https, host=dynamodb.us-
east-1.amazonaws.com, encodedPath=/, headers=[amz-sdk-invocation-id, Content-Length,
Content-Type, User-Agent, X-Amz-Target], queryParameters[])
2022-09-23 16:02:08 [main] DEBUG software.amazon.awssdk.request:85 - Received
successful response: 200, Request ID:
QS9DUMME2NHEDH8TGT9N5V530JVV4KQNS05AEMVJF66Q9ASUAAJG, Extended Request ID: not
available
```

Pagination

Some requests, such as [Query](#) and [Scan](#), limit the size of data returned on a single request and require you make repeated requests to pull subsequent pages.

You can control the maximum number of items to be read for each page with the `Limit` parameter. For example, you can use the `Limit` parameter to retrieve only the last 10 items. This limit is how many items should be read from the table before any filtering is applied. There's no way to specify you want exactly 10 after filtering. You can only control the pre-filtered count and check client-side when you've actually retrieved 10 items. Regardless of the limit, every response always has a maximum size of 1 MB.

In the API response a `LastEvaluatedKey` might be included, which indicates that the response ended because it hit a count limit or a size limit. This key is the last key evaluated for that response. Interacting directly with the API you can retrieve this `LastEvaluatedKey` and pass it to a follow-up call as `ExclusiveStartKey` to read the next chunk from that starting point. If no `LastEvaluatedKey` is returned, it means that there're no more items matching the `Query` or `Scan` API call.

The following example uses the low-level interface to limit the items to 100 based on the `keyConditionExpression` parameter.

```
QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
    .expressionAttributeValues(Map.of(
```

```
        ":pk_val", AttributeValue.fromS("123"),
        ":sk_val", AttributeValue.fromN("1000")))
    .keyConditionExpression("pk = :pk_val AND sk > :sk_val")
    .limit(100)
    .tableName(TABLE_NAME);

while (true) {
    QueryResponse queryResponse = DYNAMODB_CLIENT.query(queryRequestBuilder.build());

    queryResponse.items().forEach(item -> {
        LOGGER.info("item PK: [" + item.get("pk") + "] and SK: [" + item.get("sk") +
    "']");
    });

    if (!queryResponse.hasLastEvaluatedKey()) {
        break;
    }
    queryRequestBuilder.exclusiveStartKey(queryResponse.lastEvaluatedKey());
}
```

The AWS SDK for Java 2.x can simplify this interaction with DynamoDB by providing auto-pagination methods that make multiple service calls to get the next pages of results for you automatically. This simplifies your code, but does take away some control on resource usage that you would keep by reading pages manually.

By using the `Iterable` methods available in the DynamoDB client such as [QueryPaginator](#) and [ScanPaginator](#), the SDK takes care of the pagination. The return type of these methods is a custom iterable that you can use to iterate through all the pages. The SDK will internally handle service calls for you. Using the Java Stream API, you could handle the result of `QueryPaginator` as shown in the following example.

```
QueryPublisher queryPublisher =
    DYNAMODB_CLIENT.queryPaginator(QueryRequest.builder()
        .expressionAttributeValues(Map.of(
            ":pk_val", AttributeValue.fromS("123"),
            ":sk_val", AttributeValue.fromN("1000")))
        .keyConditionExpression("pk = :pk_val AND sk > :sk_val")
        .limit(100)
        .tableName("YourTableName")
        .build()));

queryPublisher.items().subscribe(item ->
```

```
System.out.println(item.get("itemData"))).join();
```

Data class annotations

The Java SDK provides several annotations which you can put on your data class' attributes that will influence how the SDK interacts with them. By adding the annotation, you can have an attribute behave as an implicit atomic counter, maintain an auto-generated timestamp value, or track an item version number. For more information, see [Data class annotations](#).

Working with tables, items, queries, scans, and indexes

This section provides details about working with tables, items, queries, and more in Amazon DynamoDB.

Topics

- [Working with tables and data in DynamoDB](#)
- [Global tables - multi-Region replication for DynamoDB](#)
- [Working with read and write operations](#)
- [Improving data access with secondary indexes](#)
- [Managing complex workflows with DynamoDB transactions](#)
- [Change data capture with Amazon DynamoDB](#)
- [Using On-Demand backup and restore for DynamoDB](#)
- [Point-in-time recovery for DynamoDB](#)

Working with tables and data in DynamoDB

This section describes how to use the AWS Command Line Interface (AWS CLI) and the AWS SDKs to create, update, and delete tables in Amazon DynamoDB.

Note

You can also perform these same tasks using the AWS Management Console. For more information, see [Using the console](#).

This section also provides more information about throughput capacity using DynamoDB auto scaling or manually setting provisioned throughput.

Topics

- [Basic operations on DynamoDB tables](#)
- [Considerations when changing read/write Capacity Mode](#)
- [Considerations when choosing a table class](#)

- [Managing settings on DynamoDB provisioned capacity tables](#)
- [DynamoDB Item sizes and formats](#)
- [Managing throughput capacity automatically with DynamoDB auto scaling](#)
- [Adding tags and labels to resources](#)
- [Working with DynamoDB tables in Java](#)
- [Working with DynamoDB tables in .NET](#)

Basic operations on DynamoDB tables

Similar to other database systems, Amazon DynamoDB stores data in tables. You can manage your tables using a few basic operations.

Topics

- [Creating a table](#)
- [Describing a table](#)
- [Updating a table](#)
- [Deleting a table](#)
- [Using deletion protection](#)
- [Listing table names](#)
- [Describing provisioned throughput quotas](#)

Creating a table

Use the `CreateTable` operation to create a table in Amazon DynamoDB. To create the table, you must provide the following information:

- **Table name.** The name must conform to the DynamoDB naming rules, and must be unique for the current AWS account and Region. For example, you could create a `People` table in US East (N. Virginia) and another `People` table in Europe (Ireland). However, these two tables would be entirely different from each other. For more information, see [Supported data types and naming rules in Amazon DynamoDB](#).
- **Primary key.** The primary key can consist of one attribute (partition key) or two attributes (partition key and sort key). You need to provide the attribute names, data types, and the role of

each attribute: HASH (for a partition key) and RANGE (for a sort key). For more information, see [Primary key](#).

- **Throughput settings (for provisioned tables).** If using provisioned mode, you must specify the initial read and write throughput settings for the table. You can modify these settings later, or enable DynamoDB auto scaling to manage the settings for you. For more information, see [Managing settings on DynamoDB provisioned capacity tables](#) and [Managing throughput capacity automatically with DynamoDB auto scaling](#).

Example 1: Create a provisioned table

The following AWS CLI example shows how to create a table (Music). The primary key consists of Artist (partition key) and SongTitle (sort key), each of which has a data type of String. The maximum throughput for this table is 10 read capacity units and 5 write capacity units.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5
```

The CreateTable operation returns metadata for the table, as shown following.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        }
    }
}
```

```
"ProvisionedThroughput": {  
    "NumberOfDecreasesToday": 0,  
    "WriteCapacityUnits": 5,  
    "ReadCapacityUnits": 10  
},  
"TableSizeBytes": 0,  
"TableName": "Music",  
"TableStatus": "CREATING",  
"TableId": "12345678-0123-4567-a123-abcdefghijkl",  
"KeySchema": [  
    {  
        "KeyType": "HASH",  
        "AttributeName": "Artist"  
    },  
    {  
        "KeyType": "RANGE",  
        "AttributeName": "SongTitle"  
    }  
],  
"ItemCount": 0,  
"CreationDateTime": 1542397215.37  
}  
}
```

The TableStatus element indicates the current state of the table (CREATING). It might take a while to create the table, depending on the values you specify for ReadCapacityUnits and WriteCapacityUnits. Larger values for these require DynamoDB to allocate more resources for the table.

Example 2: Create an on-demand table

To create the same table Music using on-demand mode.

```
aws dynamodb create-table \  
    --table-name Music \  
    --attribute-definitions \  
        AttributeName=Artist,AttributeType=S \  
        AttributeName=SongTitle,AttributeType=S \  
    --key-schema \  
        AttributeName=Artist,KeyType=HASH \  
        AttributeName=SongTitle,KeyType=RANGE \  
    --billing-mode=PAY_PER_REQUEST
```

The `CreateTable` operation returns metadata for the table, as shown following.

```
{  
    "TableDescription": {  
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "Artist",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "AttributeType": "S"  
            }  
        ],  
        "ProvisionedThroughput": {  
            "NumberOfDecreasesToday": 0,  
            "WriteCapacityUnits": 0,  
            "ReadCapacityUnits": 0  
        },  
        "TableSizeBytes": 0,  
        "TableName": "Music",  
        "BillingModeSummary": {  
            "BillingMode": "PAY_PER_REQUEST"  
        },  
        "TableStatus": "CREATING",  
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",  
        "KeySchema": [  
            {  
                "KeyType": "HASH",  
                "AttributeName": "Artist"  
            },  
            {  
                "KeyType": "RANGE",  
                "AttributeName": "SongTitle"  
            }  
        ],  
        "ItemCount": 0,  
        "CreationDateTime": 1542397468.348  
    }  
}
```

⚠️ Important

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

Example 3: Create a table using the DynamoDB standard-infrequent access table class

To create the same Music table using the DynamoDB Standard-Infrequent Access table class.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

The `CreateTable` operation returns metadata for the table, as shown following.

```
{
  "TableDescription": {
    "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
    "AttributeDefinitions": [
      {
        "AttributeName": "Artist",
        "AttributeType": "S"
      },
      {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
      }
    ],
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "WriteCapacityUnits": 5,
      "ReadCapacityUnits": 10
    },
  }
}
```

```
"TableClassSummary": {  
    "LastUpdateDateTime": 1542397215.37,  
    "TableClass": "STANDARD_INFREQUENT_ACCESS"  
},  
"TableSizeBytes": 0,  
"TableName": "Music",  
"TableStatus": "CREATING",  
"TableId": "12345678-0123-4567-a123-abcdefghijkl",  
"KeySchema": [  
    {  
        "KeyType": "HASH",  
        "AttributeName": "Artist"  
    },  
    {  
        "KeyType": "RANGE",  
        "AttributeName": "SongTitle"  
    }  
],  
"ItemCount": 0,  
"CreationDateTime": 1542397215.37  
}  
}
```

Describing a table

To view details about a table, use the `DescribeTable` operation. You must provide the table name. The output from `DescribeTable` is in the same format as that from `CreateTable`. It includes the timestamp when the table was created, its key schema, its provisioned throughput settings, its estimated size, and any secondary indexes that are present.

Important

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

Example

```
aws dynamodb describe-table --table-name Music
```

The table is ready for use when the `TableStatus` has changed from `CREATING` to `ACTIVE`.

Note

If you issue a `DescribeTable` request immediately after a `CreateTable` request, DynamoDB might return an error (`ResourceNotFoundException`). This is because `DescribeTable` uses an eventually consistent query, and the metadata for your table might not be available at that moment. Wait for a few seconds, and then try the `DescribeTable` request again.

For billing purposes, your DynamoDB storage costs include a per-item overhead of 100 bytes. (For more information, go to [DynamoDB Pricing](#).) This extra 100 bytes per item is not used in capacity unit calculations or by the `DescribeTable` operation.

Updating a table

The `UpdateTable` operation allows you to do one of the following:

- Modify a table's provisioned throughput settings (for provisioned mode tables).
- Change the table's read/write capacity mode.
- Manipulate global secondary indexes on the table (see [Using Global Secondary Indexes in DynamoDB](#)).
- Enable or disable DynamoDB Streams on the table (see [Change data capture for DynamoDB Streams](#)).

Example

The following AWS CLI example shows how to modify a table's provisioned throughput settings.

```
aws dynamodb update-table --table-name Music \
    --provisioned-throughput ReadCapacityUnits=20,WriteCapacityUnits=10
```

Note

When you issue an `UpdateTable` request, the status of the table changes from `AVAILABLE` to `UPDATING`. The table remains fully available for use while it is `UPDATING`. When this process is completed, the table status changes from `UPDATING` to `AVAILABLE`.

Example

The following AWS CLI example shows how to modify a table's read/write capacity mode to on-demand mode.

```
aws dynamodb update-table --table-name Music \
    --billing-mode PAY_PER_REQUEST
```

Deleting a table

You can remove an unused table with the `DeleteTable` operation. Deleting a table is an unrecoverable operation.

Example

The following AWS CLI example shows how to delete a table.

```
aws dynamodb delete-table --table-name Music
```

When you issue a `DeleteTable` request, the table's status changes from ACTIVE to DELETING. It might take a while to delete the table, depending on the resources it uses (such as the data stored in the table, and any streams or indexes on the table).

When the `DeleteTable` operation concludes, the table no longer exists in DynamoDB.

Using deletion protection

You can protect a table from accidental deletion with the deletion protection property. Enabling this property for tables helps ensure that tables do not get accidentally deleted during regular table management operations by your administrators. This will help prevent disruption to your normal business operations.

The table owner or an authorized administrator controls the deletion protection property for each table. The deletion protection property for every table is off by default. This includes global replicas, and tables restored from backups. When deletion protection is disabled for a table, the table can be deleted by any users authorized by an Identity and Access Management (IAM) policy. When deletion protection is enabled for a table, it cannot be deleted by anyone.

To change this setting, go to the table's **Additional settings**, navigate to the **Deletion Protection** panel and select **Enable delete protection**.

The deletion protection property is supported by the DynamoDB console, API, CLI/SDK and AWS CloudFormation. The `CreateTable` API supports the deletion protection property at table creation time, and the `UpdateTable` API supports changing the deletion protection property for existing tables.

 **Note**

- If an AWS account is deleted, all of that account's data including tables are still deleted within 90 days.
- If DynamoDB loses access to a customer managed key that was used to encrypt a table, it will still archive the table. Archiving involves making a backup of the table and deleting the original.

Listing table names

The `ListTables` operation returns the names of the DynamoDB tables for the current AWS account and Region.

Example

The following AWS CLI example shows how to list the DynamoDB table names.

```
aws dynamodb list-tables
```

Describing provisioned throughput quotas

The `DescribeLimits` operation returns the current read and write capacity quotas for the current AWS account and Region.

Example

The following AWS CLI example shows how to describe the current provisioned throughput quotas.

```
aws dynamodb describe-limits
```

The output shows the upper quotas of read and write capacity units for the current AWS account and Region.

For more information about these quotas, and how to request quota increases, see [Throughput default quotas](#).

Considerations when changing read/write Capacity Mode

You can switch between read/write capacity modes once every 24 hours. The only exception to this is if you switch a provisioned mode table to on-demand mode: you can switch back to provisioned mode in the same 24-hour period. Consider the following when updating your read/write capacity mode in Amazon DynamoDB.

Managing capacity

When you update a table from provisioned to on-demand mode, you don't need to specify how much read and write throughput you expect your application to perform.

Consider the following when you update a table from on-demand to provisioned mode:

- If you're using the AWS CLI or AWS SDK, choose the right provisioned capacity settings of your table and global secondary indexes by using Amazon CloudWatch to look at your historical consumption (`ConsumedWriteCapacityUnits` and `ConsumedReadCapacityUnits` metrics) to determine the new throughput settings.

 **Note**

If you're switching a global table to provisioned mode, look at the maximum consumption across all your regional replicas for base tables and global secondary indexes when determining the new throughput settings.

- If you are switching from on-demand mode back to provisioned mode, make sure to set the initial provisioned units high enough to handle your table or index capacity during the transition.

Managing auto scaling

When you update a table from provisioned to on-demand mode:

- If you're using the console, all of your auto scaling settings (if any) will be deleted.
- If you're using the AWS CLI or AWS SDK, all of your auto scaling settings will be preserved. These settings can be applied when you update your table to provisioned billing mode again.
- If the table is a global table, auto scaling settings are not preserved.

When you update a table from on-demand to provisioned mode:

- If you're using the console, DynamoDB recommends enabling auto scaling with the following defaults:
 - Target utilization: 70%
 - Minimum provisioned capacity: 5 units
 - Maximum provisioned capacity: The Region maximum
- If you're using the AWS CLI or SDK, your previous auto scaling settings (if any) are preserved.
- If the table is a global table, any previous auto scaling settings will be overwritten with the recommended defaults.

Considerations when choosing a table class

DynamoDB offers two table classes designed to help you optimize for cost. The DynamoDB Standard table class is the default, and is recommended for the vast majority of workloads. The DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) table class is optimized for tables where storage is the dominant cost. For example, tables that store infrequently accessed data, such as application logs, old social media posts, e-commerce order history, and past gaming achievements, are good candidates for the Standard-IA table class.

Every DynamoDB table is associated with a table class. All secondary indexes associated with the table use the same table class. You can set your table class when creating your table (DynamoDB Standard by default) and update the table class of an existing table using the AWS Management Console, AWS CLI, or AWS SDK. DynamoDB also supports managing your table class using AWS CloudFormation for single-region tables (tables that are not global tables). Each table class offers different pricing for data storage as well as read and write requests. When choosing a table class for your table, keep the following in mind:

- The DynamoDB Standard table class offers lower throughput costs than DynamoDB Standard-IA and is the most cost-effective option for tables where throughput is the dominant cost.
- The DynamoDB Standard-IA table class offers lower storage costs than DynamoDB Standard, and is the most cost-effective option for tables where storage is the dominant cost. When storage exceeds 50% of the throughput (reads and writes) cost of a table using the DynamoDB Standard table class, the DynamoDB Standard-IA table class can help you reduce your total table cost.
- DynamoDB Standard-IA tables offer the same performance, durability, and availability as DynamoDB Standard tables.

- Switching between the DynamoDB Standard and DynamoDB Standard-IA table classes does not require changing your application code. You use the same DynamoDB APIs and service endpoints regardless of the table class your tables use.
- DynamoDB Standard-IA tables are compatible with all existing DynamoDB features such as auto scaling, on-demand mode, time-to-live (TTL), on-demand backups, point-in-time recovery (PITR), and global secondary indexes.

The most cost-effective table class for your table depends on your table's expected storage and throughput usage patterns. You can look at your table's historical storage and throughput cost and usage with AWS Cost and Usage Reports and the AWS Cost Explorer. Use this historical data to determine the most cost-effective table class for your table. To learn more about using AWS Cost and Usage Reports and the AWS Cost Explorer, see the [AWS Billing and Cost Management Documentation](#). See [Amazon DynamoDB Pricing](#) for details about table class pricing.

 **Note**

A table class update is a background process. You can still access your table normally during a table class update. The time to update your table class depends on your table traffic, storage size, and other related variables. No more than two table class updates on your table are allowed in a 30-day trailing period.

Managing settings on DynamoDB provisioned capacity tables

When you create a new provisioned table in Amazon DynamoDB, you must specify its *provisioned throughput capacity*. This is the amount of read and write activity that the table can support. DynamoDB uses this information to reserve sufficient system resources to meet your throughput requirements.

 **Note**

You can create an on-demand mode table instead so that you don't have to manage any capacity settings for servers, storage, or throughput. DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level hits a new peak, DynamoDB adapts rapidly to accommodate the workload. For more information, see [On-demand mode](#).

You can optionally allow DynamoDB auto scaling to manage your table's throughput capacity. However, you still must provide initial settings for read and write capacity when you create the table. DynamoDB auto scaling uses these initial settings as a starting point, and then adjusts them dynamically in response to your application's requirements. For more information, see [Managing throughput capacity automatically with DynamoDB auto scaling](#).

As your application data and access requirements change, you might need to adjust your table's throughput settings. If you're using DynamoDB auto scaling, the throughput settings are automatically adjusted in response to actual workloads. You can also use the `UpdateTable` operation to manually adjust your table's throughput capacity. You might decide to do this if you need to bulk-load data from an existing data store into your new DynamoDB table. You could create the table with a large write throughput setting and then reduce this setting after the bulk data load is complete.

You specify throughput requirements in terms of *capacity units*—the amount of data your application needs to read or write per second. You can modify these settings later, if needed, or enable DynamoDB auto scaling to modify them automatically.

Topics

- [Read capacity units](#)
- [Write capacity units](#)
- [Request throttling and burst capacity](#)
- [Request throttling and adaptive capacity](#)
- [Choosing initial throughput settings](#)
- [Modifying throughput settings](#)
- [Troubleshooting throttling issues](#)
- [Using CloudWatch metrics to investigate throttling](#)

Read capacity units

A *read capacity unit* represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size.

Note

To learn more about DynamoDB read consistency models, see [Read consistency](#).

For example, suppose that you create a table with 10 provisioned read capacity units. This allows you to perform 10 strongly consistent reads per second, or 20 eventually consistent reads per second, for items up to 4 KB.

Reading an item larger than 4 KB consumes more read capacity units. For example, a strongly consistent read of an item that is 8 KB ($4\text{ KB} \times 2$) consumes 2 read capacity units. An eventually consistent read on that same item consumes only 1 read capacity unit.

Item sizes for reads are rounded up to the next 4 KB multiple. For example, reading a 3,500-byte item consumes the same throughput as reading a 4 KB item.

Capacity unit consumption for reads

The following describes how DynamoDB read operations consume read capacity units:

- **GetItem**—Reads a single item from a table. To determine the number of capacity units that GetItem will consume, take the item size and round it up to the next 4 KB boundary. If you specified a strongly consistent read, this is the number of capacity units required. For an eventually consistent read (the default), divide this number by two.

For example, if you read an item that is 3.5 KB, DynamoDB rounds the item size to 4 KB. If you read an item of 10 KB, DynamoDB rounds the item size to 12 KB.

- **BatchGetItem**—Reads up to 100 items, from one or more tables. DynamoDB processes each item in the batch as an individual GetItem request, so DynamoDB first rounds up the size of each item to the next 4 KB boundary, and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if BatchGetItem reads a 1.5 KB item and a 6.5 KB item, DynamoDB calculates the size as 12 KB ($4\text{ KB} + 8\text{ KB}$), not 8 KB ($1.5\text{ KB} + 6.5\text{ KB}$).
- **Query**—Reads multiple items that have the same partition key value. All items returned are treated as a single read operation, where DynamoDB computes the total size of all items and then rounds up to the next 4 KB boundary. For example, suppose your query returns 10 items whose combined size is 40.8 KB. DynamoDB rounds the item size for the operation to 44 KB. If a query returns 1500 items of 64 bytes each, the cumulative size is 96 KB.
- **Scan**—Reads all items in a table. DynamoDB considers the size of the data that is evaluated, not the size of the data returned by the scan.

If you perform a read operation on an item that does not exist, DynamoDB still consumes provisioned read throughput: A strongly consistent read request consumes one read capacity unit, while an eventually consistent read request consumes 0.5 of a read capacity unit.

For any operation that returns items, you can request a subset of attributes to retrieve. However, doing so has no impact on the item size calculations. In addition, Query and Scan can return item counts instead of attribute values. Getting the count of items uses the same quantity of read capacity units and is subject to the same item size calculations. This is because DynamoDB has to read each item in order to increment the count.

Read operations and read consistency

The preceding calculations assume strongly consistent read requests. For an eventually consistent read request, the operation consumes only half the capacity units. For an eventually consistent read, if the total item size is 80 KB, the operation consumes only 10 capacity units.

Write capacity units

A *write capacity unit* represents one write per second, for an item up to 1 KB in size.

For example, suppose that you create a table with 10 write capacity units. This allows you to perform 10 writes per second, for items up to 1 KB in size per second.

Item sizes for writes are rounded up to the next 1 KB multiple. For example, writing a 500-byte item consumes the same throughput as writing a 1 KB item.

Capacity unit consumption for writes

The following describes how DynamoDB write operations consume write capacity units:

- `PutItem`—Writes a single item to a table. If an item with the same primary key exists in the table, the operation replaces the item. For calculating provisioned throughput consumption, the item size that matters is the larger of the two.
- `UpdateItem`—Modifies a single item in the table. DynamoDB considers the size of the item as it appears before and after the update. The provisioned throughput consumed reflects the larger of these item sizes. Even if you update just a subset of the item's attributes, `UpdateItem` will still consume the full amount of provisioned throughput (the larger of the "before" and "after" item sizes).
- `DeleteItem`—Removes a single item from a table. The provisioned throughput consumption is based on the size of the deleted item.

- **BatchWriteItem**—Writes up to 25 items to one or more tables. DynamoDB processes each item in the batch as an individual PutItem or DeleteItem request (updates are not supported). So DynamoDB first rounds up the size of each item to the next 1 KB boundary, and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if BatchWriteItem writes a 500-byte item and a 3.5 KB item, DynamoDB calculates the size as 5 KB (1 KB + 4 KB), not 4 KB (500 bytes + 3.5 KB).

For PutItem, UpdateItem, and DeleteItem operations, DynamoDB rounds the item size up to the next 1 KB. For example, if you put or delete an item of 1.6 KB, DynamoDB rounds the item size up to 2 KB.

PutItem, UpdateItem, and DeleteItem allow *conditional writes*, where you specify an expression that must evaluate to true in order for the operation to succeed. If the expression evaluates to false, DynamoDB still consumes write capacity units from the table. The amount consumed is dependent on the size of the item (whether it's an existing item in the table or a new one you are attempting to create or update). For example, if an existing item is 300kb and the new item you are trying to create or update is 310kb, the write capacity units consumed will be the 310kb item.

Request throttling and burst capacity

If your application performs reads or writes at a higher rate than your table can support, DynamoDB begins to *throttle* those requests. When DynamoDB throttles a read or write, it returns a ProvisionedThroughputExceededException to the caller. The application can then take appropriate action, such as waiting for a short interval before retrying the request.

Note

We recommend that you use the AWS SDKs for software development. The AWS SDKs provide built-in support for retrying throttled requests; you do not need to write this logic yourself. For more information, see [Error retries and exponential backoff](#).

The DynamoDB console displays Amazon CloudWatch metrics for your tables, so you can monitor throttled read requests and write requests. If you encounter excessive throttling, you should consider increasing your table's provisioned throughput settings.

In some cases, DynamoDB uses *burst capacity* to accommodate reads or writes in excess of your table's throughput settings. With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled. For more information, see [Using burst capacity effectively](#).

Request throttling and adaptive capacity

DynamoDB automatically distributes your data across partitions, which are stored on multiple servers in the AWS Cloud (For more information, see [Partitions and data distribution](#)). It's not always possible to distribute read and write activity evenly all the time. When data access is imbalanced, a "hot" partition can receive such a higher volume of read and write traffic compared to other partitions. Adaptive capacity works by automatically increasing throughput capacity for partitions that receive more traffic. For more information, see [Understanding DynamoDB adaptive capacity](#).

Choosing initial throughput settings

Every application has different requirements for reading and writing from a database. When you are determining the initial throughput settings for a DynamoDB table, take the following inputs into consideration:

- **Item sizes.** Some items are small enough that they can be read or written using a single capacity unit. Larger items require multiple capacity units. By estimating the sizes of the items that will be in your table, you can specify accurate settings for your table's provisioned throughput.
- **Expected read and write request rates.** In addition to item size, you should estimate the number of reads and writes you need to perform per second.
- **Read consistency requirements.** Read capacity units are based on strongly consistent read operations, which consume twice as many database resources as eventually consistent reads. You should determine whether your application requires strongly consistent reads, or whether it can relax this requirement and perform eventually consistent reads instead. (Read operations in DynamoDB are eventually consistent, by default. You can request strongly consistent reads for these operations if necessary.)

For example, suppose that you want to read 80 items per second from a table. The items are 3 KB in size, and you want strongly consistent reads. For this scenario, each read requires one provisioned read capacity unit. To determine this number, you divide the item size of the operation by 4 KB, and then round up to the nearest whole number, as in this example:

- $3 \text{ KB} / 4 \text{ KB} = 0.75$, or **1** read capacity unit

For this scenario, you have to set the table's provisioned read throughput to 80 read capacity units:

- 1 read capacity unit per item \times 80 reads per second = **80** read capacity units

Now suppose that you want to write 100 items per second to your table, and that the items are 512 bytes in size. For this scenario, each write requires one provisioned write capacity unit. To determine this number, you divide the item size of the operation by 1 KB, and then round up to the nearest whole number:

- $512 \text{ bytes} / 1 \text{ KB} = 0.5$, or **1**

For this scenario, you would want to set the table's provisioned write throughput to 100 write capacity units:

- 1 write capacity unit per item \times 100 writes per second = **100** write capacity units

 **Note**

For recommendations on provisioned throughput and related topics, see [Best practices for designing and using partition keys effectively](#).

Modifying throughput settings

If you have enabled DynamoDB auto scaling for a table, then its throughput capacity is dynamically adjusted in response to actual usage. No manual intervention is required.

You can modify your table's provisioned throughput settings using the AWS Management Console or the `UpdateTable` operation. For more information about throughput increases and decreases per day, see [Service, account, and table quotas in Amazon DynamoDB](#).

Troubleshooting throttling issues

For troubleshooting issues that appear to be related to throttling, an important first is to confirm if the throttling is coming from DynamoDB or from the application.

Following are some common scenarios, and possible steps to help resolve them.

The DynamoDB table appears to have enough provisioned capacity, but requests are being throttled.

This can occur when the throughput is below the average per minute, but throughput exceeds the amount available per second. DynamoDB only reports minute level metrics to CloudWatch, which are then calculated as the sum for one minute and averaged. But DynamoDB itself applies rate limits per second. So if too much of that throughput occurs within a small portion of that minute, such as few seconds or less, then requests for the rest of that minute can be throttled. For example, if we have provisioned 60 WCU on a table then it can do 3600 writes in 1 minute. But if all 3600 WCU requests hit in the same second, then the rest of that minute will be throttled.

One way to resolve this scenario can be to add some jitters and exponential back off to the API calls. For more information see this post about [backoff and jitter](#).

Autoscaling is enabled, but tables are still being throttled.

This can occur during sudden spikes in traffic. Auto scaling can be triggered when two data points breach the configured target utilization value within a one-minute span. Therefore, auto scaling can take place because the consumed capacity is above target utilization for two consistent minutes. But if the spikes are more than a minute apart, auto scaling might not be triggered. Similarly, a scale down event can be triggered when 15 consecutive data points are lower than the target utilization. In either case, after auto scaling is triggered an [UpdateTable](#) call is invoked. It can then take several minutes to update the provisioned capacity for the table or the index. During this period, any requests that exceed the previous provisioned capacity of the tables will be throttled.

In summary, Autoscaling requires consecutive data points where the target utilization value is being breached to scale up a DynamoDB table. For this reason Autoscaling is not recommended as a solution for dealing with spiked workloads. Please refer to the [Autoscaling documentation](#) for more information.

Tables are in "On-Demand" capacity mode but are still being throttled.

For On-Demand tables, DynamoDB automatically allocates more capacity as your traffic volume increases. As long as the access is distributed evenly across partitions, and the table does not exceed double its previous peak traffic, the overall table will not be throttled. However, throttling can still occur if the throughput exceeds double the previous peak within the same 30 minutes. Please refer to [OnDemand scaling](#) for further information.

A hot key may be causing throttling issues.

In DynamoDB, a partition key that does not have high cardinality can result in many requests which target just a few partitions. If a resulting "hot partition" goes past the per partition limits of 3000 RCU or 1000 WCU per second, this can result in throttling. The diagnostic tool CloudWatch Contributor Insights, can help debug this, by providing CCI graphs for each table's item access patterns. You can continuously monitor your DynamoDB tables' most frequently accessed keys and other traffic trends. To find out more about CloudWatch Contributor Insights see [CCI](#). For specific information on choosing the right hot key see [Choosing the right hot key](#).

Using CloudWatch metrics to investigate throttling

Below are some DynamoDB metrics to monitor during throttling events, to help locate which operations are creating throttled requests and identify root issues.

1. ThrottledRequests

- One throttled request can contain multiple throttled events, so events can be more relevant to examine.

For example, when you update an item in a table with GSIs there are multiple events - a write to the table, and a write to each index. Even if one or more of these events are throttled, there will only be one ThrottledRequest.

2. ReadThrottleEvents

- Watch for requests that exceed the provisioned RCU for a table or GSI.

3. WriteThrottleEvents

- Watch for requests that exceed the provisioned WCU for a table or GSI.

4. OnlineIndexConsumedWriteCapacity

- Pay attention to the number of WCU consumed when adding a new GSI to a table. Note that ConsumedWriteCapacityUnits for a GSI does not include the WCU consumed during index creation.
- If you have set the WCU for a GSI too low, then incoming write activity during the backfill phase might be throttled.

5. OnlineIndexThrottleEvents

- Review the number of write throttle events that occur when adding a new GSI to a table.
- If you find that the WCU is set too low and is being throttled, you can update the WCU value for a GSI even during backfill.

6. Provisioned Read/Write

- View how many provisioned read or write capacity units were consumed over the specified time period, for a table or a specified global secondary index.
- Note that the TableName dimension returns ProvisionedReadCapacityUnits for the table only by default. To view the numer of provisioned read or write capacity units for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

7. Consumed Read/Write

- View how many read or write capacity units were consumed over the specified time period.

For more information on DynamoDB CloudWatch Metrics, see [Metrics and Dimensions](#).

DynamoDB Item sizes and formats

DynamoDB tables are schemaless, except for the primary key, so the items in a table can all have different attributes, sizes, and data types.

The total size of an item is the sum of the lengths of its attribute names and values, plus any applicable overhead as described below. You can use the following guidelines to estimate attribute sizes:

- Strings are Unicode with UTF-8 binary encoding. The size of a string is *(number of UTF-8-encoded bytes of attribute name) + (number of UTF-8-encoded bytes)*.
- Numbers are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed. The size of a number is approximately *(number of UTF-8-encoded bytes of attribute name) + (1 byte per two significant digits) + (1 byte)*.
- A binary value must be encoded in base64 format before it can be sent to DynamoDB, but the value's raw byte length is used for calculating size. The size of a binary attribute is *(number of UTF-8-encoded bytes of attribute name) + (number of raw bytes)*.

- The size of a null attribute or a Boolean attribute is *(number of UTF-8-encoded bytes of attribute name) + (1 byte)*.
- An attribute of type List or Map requires 3 bytes of overhead, regardless of its contents. The size of a List or Map is *(number of UTF-8-encoded bytes of attribute name) + sum (size of nested elements) + (3 bytes)*. The size of an empty List or Map is *(number of UTF-8-encoded bytes of attribute name) + (3 bytes)*.
- Each List or Map element also requires 1 byte of overhead.

 **Note**

We recommend that you choose shorter attribute names rather than long ones. This helps you reduce the amount of storage required, but also can lower the amount of RCU/WCUs you use.

For storage billing purposes, each item includes a per-item storage overhead that depends on the features you have enabled.

- All items in DynamoDB require 100 bytes of storage overhead for indexing.
- Some DynamoDB features (global tables, transactions, change data capture for Kinesis Data Streams with DynamoDB) require additional storage overhead to account for system-created attributes resulting from enabling those features. For example, global tables requires an additional 48 bytes of storage overhead.

Managing throughput capacity automatically with DynamoDB auto scaling

Many database workloads are cyclical in nature, while others are difficult to predict in advance. For one example, consider a social networking app where most of the users are active during daytime hours. The database must be able to handle the daytime activity, but there's no need for the same levels of throughput at night. For another example, consider a new mobile gaming app that is experiencing unexpectedly rapid adoption. If the game becomes too popular it could exceed the available database resources, resulting in slow performance and unhappy customers. These kinds of workloads often require manual intervention to scale database resources up or down in response to varying usage levels.

Amazon DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. This enables a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases, Application Auto Scaling decreases the throughput so that you don't pay for unused provisioned capacity.

Note

If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default. You can modify your auto scaling settings at any time. For more information, see [Using the AWS Management Console with DynamoDB auto scaling](#).

When you delete a table or global table replica then any associated scalable targets, scaling policies, or CloudWatch alarms are not automatically deleted with it.

With Application Auto Scaling, you create a *scaling policy* for a table or a global secondary index. The scaling policy specifies whether you want to scale read capacity or write capacity (or both), and the minimum and maximum provisioned capacity unit settings for the table or index.

The scaling policy also contains a *target utilization*—the percentage of consumed provisioned throughput at a point in time. Application Auto Scaling uses a *target tracking* algorithm to adjust the provisioned throughput of the table (or index) upward or downward in response to actual workloads, so that the actual capacity utilization remains at or near your target utilization.

Auto scaling can be triggered when two data points breach the configured target utilization value within a one-minute span. Therefore, auto scaling can take place because the consumed capacity is above target utilization for two consistent minutes. But if the spikes are more than a minute apart, auto scaling might not be triggered. Similarly, a scale down event can be triggered when 15 consecutive data points are lower than the target utilization. In either case, after auto scaling is triggered an [UpdateTable](#) call is invoked. It can then take several minutes to update the provisioned capacity for the table or the index. During this period, any requests that exceed the previous provisioned capacity of the tables will be throttled.

Important

You can't adjust the number of data points to breach to trigger the underlying alarm (though the current number could change in the future).

You can set the auto scaling target utilization values between 20 and 90 percent for your read and write capacity.

Note

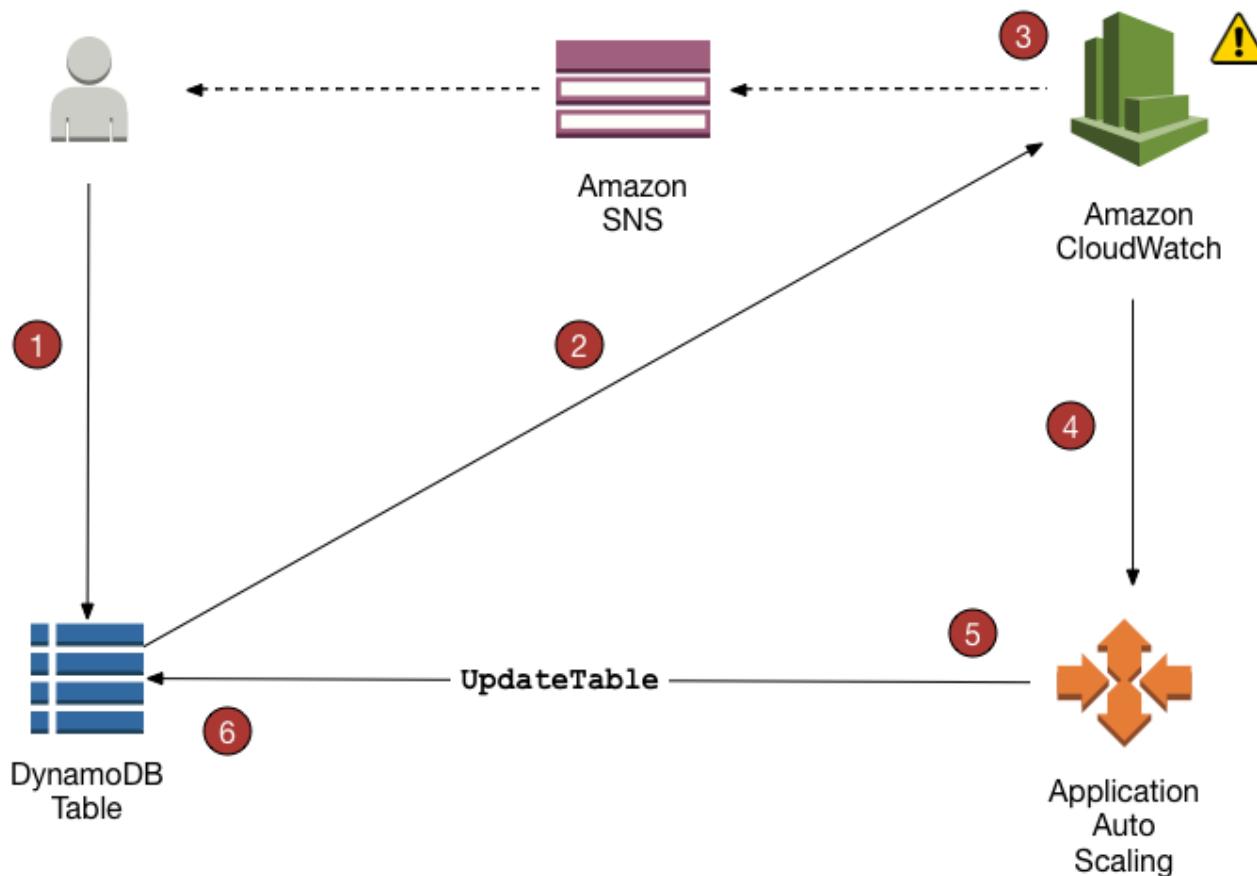
In addition to tables, DynamoDB auto scaling also supports global secondary indexes. Every global secondary index has its own provisioned throughput capacity, separate from that of its base table. When you create a scaling policy for a global secondary index, Application Auto Scaling adjusts the provisioned throughput settings for the index to ensure that its actual utilization stays at or near your desired utilization ratio.

How DynamoDB auto scaling works

Note

To get started quickly with DynamoDB auto scaling, see [Using the AWS Management Console with DynamoDB auto scaling](#).

The following diagram provides a high-level overview of how DynamoDB auto scaling manages throughput capacity for a table.



The following steps summarize the auto scaling process as shown in the previous diagram:

1. You create an Application Auto Scaling policy for your DynamoDB table.
2. DynamoDB publishes consumed capacity metrics to Amazon CloudWatch.
3. If the table's consumed capacity exceeds your target utilization (or falls below the target) for a specific length of time, Amazon CloudWatch triggers an alarm. You can view the alarm on the console and receive notifications using Amazon Simple Notification Service (Amazon SNS).
4. The CloudWatch alarm invokes Application Auto Scaling to evaluate your scaling policy.
5. Application Auto Scaling issues an **UpdateTable** request to adjust your table's provisioned throughput.
6. DynamoDB processes the **UpdateTable** request, dynamically increasing (or decreasing) the table's provisioned throughput capacity so that it approaches your target utilization.

To understand how DynamoDB auto scaling works, suppose that you have a table named `ProductCatalog`. The table is bulk-loaded with data infrequently, so it doesn't incur very much

write activity. However, it does experience a high degree of read activity, which varies over time. By monitoring the Amazon CloudWatch metrics for ProductCatalog, you determine that the table requires 1,200 read capacity units (to avoid DynamoDB throttling read requests when activity is at its peak). You also determine that ProductCatalog requires 150 read capacity units at a minimum, when read traffic is at its lowest point. For more information about preventing throttling, see [Troubleshooting throttling issues in Amazon DynamoDB](#).

Within the range of 150 to 1,200 read capacity units, you decide that a target utilization of 70 percent would be appropriate for the ProductCatalog table. *Target utilization* is the ratio of consumed capacity units to provisioned capacity units, expressed as a percentage. Application Auto Scaling uses its target tracking algorithm to ensure that the provisioned read capacity of ProductCatalog is adjusted as required so that utilization remains at or near 70 percent.

Note

DynamoDB auto scaling modifies provisioned throughput settings only when the actual workload stays elevated or depressed for a sustained period of several minutes. The Application Auto Scaling target tracking algorithm seeks to keep the target utilization at or near your chosen value over the long term.

Sudden, short-duration spikes of activity are accommodated by the table's built-in burst capacity. For more information, see [Using burst capacity effectively](#).

To enable DynamoDB auto scaling for the ProductCatalog table, you create a scaling policy. This policy specifies the following:

- The table or global secondary index that you want to manage
- Which capacity type to manage (read capacity or write capacity)
- The upper and lower boundaries for the provisioned throughput settings
- Your target utilization

When you create a scaling policy, Application Auto Scaling creates a pair of Amazon CloudWatch alarms on your behalf. Each pair represents the upper and lower boundaries for your provisioned throughput settings. These CloudWatch alarms are triggered when the table's actual utilization deviates from your target utilization for a sustained period of time.

When one of the CloudWatch alarms is triggered, Amazon SNS sends you a notification (if you have enabled it). The CloudWatch alarm then invokes Application Auto Scaling, which in turn notifies DynamoDB to adjust the ProductCatalog table's provisioned capacity upward or downward as appropriate.

During a scaling event, AWS Config is charged per configuration item recorded. When a scaling event occurs, four CloudWatch alarms are created for each read and write auto-scaling event: ProvisionedCapacity alarms: ProvisionedCapacityLow, ProvisionedCapacityHigh and ConsumedCapacity alarms: AlarmHigh, AlarmLow. This results in a total of eight alarms. Therefore, AWS Config records eight configuration items for every scaling event.

 **Note**

You can also schedule your DynamoDB scaling so it happens at certain times. Learn the basic steps [here](#).

Usage notes

Before you begin using DynamoDB auto scaling, you should be aware of the following:

- DynamoDB auto scaling can increase read capacity or write capacity as often as necessary, in accordance with your auto scaling policy. All DynamoDB quotas remain in effect, as described in [Service, account, and table quotas in Amazon DynamoDB](#).
- DynamoDB auto scaling doesn't prevent you from manually modifying provisioned throughput settings. These manual adjustments don't affect any existing CloudWatch alarms that are related to DynamoDB auto scaling.
- If you enable DynamoDB auto scaling for a table that has one or more global secondary indexes, we highly recommend that you also apply auto scaling uniformly to those indexes. This will help ensure better performance for table writes and reads, and help avoid throttling. You can enable auto scaling by selecting **Apply same settings to global secondary indexes** in the AWS Management Console. For more information, see [Enabling DynamoDB auto scaling on existing tables](#).
- When you delete a table or global table replica, any associated scalable targets, scaling policies or CloudWatch alarms are not automatically deleted with it.

- When creating a GSI for an existing table, auto scaling is not enabled for the GSI. You will have to manually manage the capacity while the GSI is being built. Once the backfill on the GSI completes and it reaches active status, auto scaling will operate as normal.

Using the AWS Management Console with DynamoDB auto scaling

When you use the AWS Management Console to create a new table, Amazon DynamoDB auto scaling is enabled for that table by default. You can also use the console to enable auto scaling for existing tables, modify auto scaling settings, or disable auto scaling.

Note

For more advanced features like setting scale-in and scale-out cooldown times, use the AWS Command Line Interface (AWS CLI) to manage DynamoDB auto scaling. For more information, see [Using the AWS CLI to manage DynamoDB auto scaling](#).

Topics

- [Before you begin: Granting user permissions for DynamoDB auto scaling](#)
- [Creating a new table with auto scaling enabled](#)
- [Enabling DynamoDB auto scaling on existing tables](#)
- [Viewing auto scaling activities on the console](#)
- [Modifying or disabling DynamoDB auto scaling settings](#)

Before you begin: Granting user permissions for DynamoDB auto scaling

In AWS Identity and Access Management (IAM), the AWS managed policy `DynamoDBFullAccess` provides the required permissions for using the DynamoDB console. However, for DynamoDB auto scaling, users require additional permissions.

Important

To delete an auto scaling-enabled table, `application-autoscaling:*` permissions are required. The AWS managed policy `DynamoDBFullAccess` includes such permissions.

To set up a user for DynamoDB console access and DynamoDB auto scaling, create a role and add the **AmazonDynamoDBFullAccess** policy to that role. Then assign the role to a user.

Creating a new table with auto scaling enabled

Note

DynamoDB auto scaling requires the presence of a service-linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see [Service-linked roles for Application Auto Scaling](#) in the *Application Auto Scaling User Guide*.

To create a new table with auto scaling enabled

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create table**.
3. On the **Create table** page, enter a **Table name** and primary key.
4. If **Default settings** is selected, the table will be created with auto scaling enabled.

Otherwise, for custom settings:

- a. Select **Customize settings**.
- b. In the **Read/write capacity settings** section, select **Provisioned** capacity mode and set **Auto scaling to On** for **Read capacity**, **Write capacity**, or both. For each of these, set your desired scaling policy for the table and, optionally, all global secondary indexes of the table.
 - **Minimum capacity units** – Enter your lower boundary for the auto scaling range.
 - **Maximum capacity units** – Enter your upper boundary for the auto scaling range.
 - **Target utilization** – Enter your target utilization percentage for the table.

Note

If you create a global secondary index for the new table, the index's capacity at time of creation will be the same as your base table's capacity. You can change the index's capacity in the table's settings after you create the table.

- When the settings are as you want them, choose **Create table**. Your table is created with the auto scaling parameters.

Enabling DynamoDB auto scaling on existing tables

Note

DynamoDB auto scaling requires the presence of a service-linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see [Service-linked roles for Application Auto Scaling](#).

To enable DynamoDB auto scaling for an existing table

- Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
- In the navigation pane on the left side of the console, choose **Tables**.
- Choose the table that you want to work with and choose the **Additional settings** tab.
- In the **Read/write capacity** section, choose **Edit**.
- In the **Capacity mode** section, choose **Provisioned**.
- In the **Table capacity** section, set **Auto scaling** to **On** for **Read capacity**, **Write capacity**, or both. For each of these, set your desired scaling policy for the table and, optionally, all global secondary indexes of the table.
 - Minimum capacity units** – Enter your lower boundary for the auto scaling range.
 - Maximum capacity units** – Enter your upper boundary for the auto scaling range.
 - Target utilization** – Enter your target utilization percentage for the table.

- **Use the same capacity read/write capacity settings for all global secondary indexes –** Choose whether global secondary indexes should use the same auto scaling policy as the base table.

 **Note**

For best performance, we recommend that you enable **Use the same read/write capacity settings for all global secondary indexes**. This option allows DynamoDB auto scaling to uniformly scale all the global secondary indexes on the base table. This includes existing global secondary indexes, and any others that you create for this table in the future.

With this option enabled, you can't set a scaling policy on an individual global secondary index.

7. When the settings are as you want them, choose **Save**.

Viewing auto scaling activities on the console

As your application drives read and write traffic to your table, DynamoDB auto scaling dynamically modifies the table's throughput settings. Amazon CloudWatch keeps track of provisioned and consumed capacity, throttled events, latency, and other metrics for all of your DynamoDB tables and secondary indexes.

To view these metrics in the DynamoDB console, choose the table that you want to work with and choose the **Monitor** tab. To create a customizable view of table metrics, select **View all in CloudWatch**.

For more information about CloudWatch monitoring in DynamoDB, see [Monitoring with Amazon CloudWatch](#).

Modifying or disabling DynamoDB auto scaling settings

You can use the AWS Management Console to modify your DynamoDB auto scaling settings. To do this, go to the **Additional settings** tab for your table, and choose **Edit** in the **Read/write capacity** section. For more information about these settings, see [Enabling DynamoDB auto scaling on existing tables](#).

Using the AWS CLI to manage DynamoDB auto scaling

Instead of using the AWS Management Console, you can use the AWS Command Line Interface (AWS CLI) to manage Amazon DynamoDB auto scaling. The tutorial in this section demonstrates how to install and configure the AWS CLI for managing DynamoDB auto scaling. In this tutorial, you do the following:

- Create a DynamoDB table named TestTable. The initial throughput settings are 5 read capacity units and 5 write capacity units.
- Create an Application Auto Scaling policy for TestTable. The policy seeks to maintain a 50 percent target ratio between consumed write capacity and provisioned write capacity. The range for this metric is between 5 and 10 write capacity units. (Application Auto Scaling is not allowed to adjust the throughput beyond this range.)
- Run a Python program to drive write traffic to TestTable. When the target ratio exceeds 50 percent for a sustained period of time, Application Auto Scaling notifies DynamoDB to adjust the throughput of TestTable upward to maintain the 50 percent target utilization.
- Verify that DynamoDB has successfully adjusted the provisioned write capacity for TestTable.

 **Note**

You can also schedule your DynamoDB scaling so it happens at certain times. Learn the basic steps [here](#).

Topics

- [Before you begin](#)
- [Step 1: Create a DynamoDB table](#)
- [Step 2: Register a scalable target](#)
- [Step 3: Create a scaling policy](#)
- [Step 4: Drive write traffic to TestTable](#)
- [Step 5: View Application Auto Scaling actions](#)
- [\(Optional\) Step 6: Clean up](#)

Before you begin

Complete the following tasks before starting the tutorial.

Install the AWS CLI

If you haven't already done so, you must install and configure the AWS CLI. To do this, follow these instructions in the *AWS Command Line Interface User Guide*:

- [Installing the AWS CLI](#)
- [Configuring the AWS CLI](#)

Install Python

Part of this tutorial requires you to run a Python program (see [Step 4: Drive write traffic to TestTable](#)). If you don't already have it installed, you can [download Python](#).

Step 1: Create a DynamoDB table

In this step, you use the AWS CLI to create TestTable. The primary key consists of pk (partition key) and sk (sort key). Both of these attributes are of type Number. The initial throughput settings are 5 read capacity units and 5 write capacity units.

1. Use the following AWS CLI command to create the table.

```
aws dynamodb create-table \
    --table-name TestTable \
    --attribute-definitions \
        AttributeName=pk,AttributeType=N \
        AttributeName=sk,AttributeType=N \
    --key-schema \
        AttributeName=pk,KeyType=HASH \
        AttributeName=sk,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

2. To check the status of the table, use the following command.

```
aws dynamodb describe-table \
    --table-name TestTable \
    --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

The table is ready for use when its status is ACTIVE.

Step 2: Register a scalable target

Next you register the table's write capacity as a scalable target with Application Auto Scaling. This allows Application Auto Scaling to adjust the provisioned write capacity for *TestTable*, but only within the range of 5–10 capacity units.

Note

DynamoDB auto scaling requires the presence of a service linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see [Service-linked roles for Application Auto Scaling](#) in the *Application Auto Scaling User Guide*.

1. Enter the following command to register the scalable target.

```
aws application-autoscaling register-scalable-target \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable" \  
  --scalable-dimension "dynamodb:table:WriteCapacityUnits" \  
  --min-capacity 5 \  
  --max-capacity 10
```

2. To verify the registration, use the following command.

```
aws application-autoscaling describe-scalable-targets \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable"
```

Note

You can also register a scalable target against a global secondary index. For example, for a global secondary index ("test-index"), the resource ID and scalable dimension arguments are updated appropriately.

```
aws application-autoscaling register-scalable-target \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable/index/test-index" \  
  --scalable-dimension "dynamodb:index:WriteCapacityUnits" \  
  --min-capacity 5 \  
  --max-capacity 10
```

```
--min-capacity 5 \
--max-capacity 10
```

Step 3: Create a scaling policy

In this step, you create a scaling policy for `TestTable`. The policy defines the details under which Application Auto Scaling can adjust your table's provisioned throughput, and what actions to take when it does so. You associate this policy with the scalable target that you defined in the previous step (write capacity units for the `TestTable` table).

The policy contains the following elements:

- **PredefinedMetricSpecification**—The metric that Application Auto Scaling is allowed to adjust. For DynamoDB, the following values are valid values for `PredefinedMetricType`:
 - `DynamoDBReadCapacityUtilization`
 - `DynamoDBWriteCapacityUtilization`
- **ScaleOutCooldown**—The minimum amount of time (in seconds) between each Application Auto Scaling event that increases provisioned throughput. This parameter allows Application Auto Scaling to continuously, but not aggressively, increase the throughput in response to real-world workloads. The default setting for `ScaleOutCooldown` is 0.
- **ScaleInCooldown**—The minimum amount of time (in seconds) between each Application Auto Scaling event that decreases provisioned throughput. This parameter allows Application Auto Scaling to decrease the throughput gradually and predictably. The default setting for `ScaleInCooldown` is 0.
- **TargetValue**—Application Auto Scaling ensures that the ratio of consumed capacity to provisioned capacity stays at or near this value. You define `TargetValue` as a percentage.

Note

To further understand how `TargetValue` works, suppose that you have a table with a provisioned throughput setting of 200 write capacity units. You decide to create a scaling policy for this table, with a `TargetValue` of 70 percent.

Now suppose that you begin driving write traffic to the table so that the actual write throughput is 150 capacity units. The consumed-to-provisioned ratio is now $(150 / 200)$, or 75 percent. This ratio exceeds your target, so Application Auto Scaling increases the

provisioned write capacity to 215 so that the ratio is (150 / 215), or 69.77 percent—as close to your TargetValue as possible, but not exceeding it.

For TestTable, you set TargetValue to 50 percent. Application Auto Scaling adjusts the table's provisioned throughput within the range of 5–10 capacity units (see [Step 2: Register a scalable target](#)) so that the consumed-to-provisioned ratio remains at or near 50 percent. You set the values for ScaleOutCooldown and ScaleInCooldown to 60 seconds.

1. Create a file named scaling-policy.json with the following contents.

```
{  
    "PredefinedMetricSpecification": {  
        "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"  
    },  
    "ScaleOutCooldown": 60,  
    "ScaleInCooldown": 60,  
    "TargetValue": 50.0  
}
```

2. Use the following AWS CLI command to create the policy.

```
aws application-autoscaling put-scaling-policy \  
    --service-namespace dynamodb \  
    --resource-id "table/TestTable" \  
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \  
    --policy-name "MyScalingPolicy" \  
    --policy-type "TargetTrackingScaling" \  
    --target-tracking-scaling-policy-configuration file://scaling-policy.json
```

3. In the output, note that Application Auto Scaling has created two Amazon CloudWatch alarms—one each for the upper and lower boundary of the scaling target range.
4. Use the following AWS CLI command to view more details about the scaling policy.

```
aws application-autoscaling describe-scaling-policies \  
    --service-namespace dynamodb \  
    --resource-id "table/TestTable" \  
    --policy-name "MyScalingPolicy"
```

5. In the output, verify that the policy settings match your specifications from [Step 2: Register a scalable target](#) and [Step 3: Create a scaling policy](#).

Step 4: Drive write traffic to TestTable

Now you can test your scaling policy by writing data to TestTable. To do this, you run a Python program.

1. Create a file named bulk-load-test-table.py with the following contents.

```
import boto3
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table("TestTable")

filler = "x" * 100000

i = 0
while (i < 10):
    j = 0
    while (j < 10):
        print (i, j)

        table.put_item(
            Item={
                'pk':i,
                'sk':j,
                'filler':{'S':filler}
            }
        )
        j += 1
    i += 1
```

2. Enter the following command to run the program.

```
python bulk-load-test-table.py
```

The provisioned write capacity for TestTable is very low (5 write capacity units), so the program stalls occasionally due to write throttling. This is expected behavior.

Let the program continue running while you move on to the next step.

Step 5: View Application Auto Scaling actions

In this step, you view the Application Auto Scaling actions that are initiated on your behalf. You also verify that Application Auto Scaling has updated the provisioned write capacity for TestTable.

1. Enter the following command to view the Application Auto Scaling actions.

```
aws application-autoscaling describe-scaling-activities \
--service-namespace dynamodb
```

Rerun this command occasionally, while the Python program is running. (It takes several minutes before your scaling policy is invoked.) You should eventually see the following output.

```
...
{
    "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
    "Description": "Setting write capacity units to 10.",
    "ResourceId": "table/TestTable",
    "ActivityId": "0cc6fb03-2a7c-4b51-b67f-217224c6b656",
    "StartTime": 1489088210.175,
    "ServiceNamespace": "dynamodb",
    "EndTime": 1489088246.85,
    "Cause": "monitor alarm AutoScaling-table/TestTable-
AlarmHigh-1bb3c8db-1b97-4353-baf1-4def76f4e1b9 in state ALARM triggered policy
MyScalingPolicy",
    "StatusMessage": "Successfully set write capacity units to 10. Change
successfully fulfilled by dynamodb.",
    "StatusCode": "Successful"
},
```

This indicates that Application Auto Scaling has issued an UpdateTable request to DynamoDB.

2. Enter the following command to verify that DynamoDB increased the table's write capacity.

```
aws dynamodb describe-table \
--table-name TestTable \
--query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

The WriteCapacityUnits should have been scaled from 5 to 10.

(Optional) Step 6: Clean up

In this tutorial, you created several resources. You can delete these resources if you no longer need them.

1. Delete the scaling policy for TestTable.

```
aws application-autoscaling delete-scaling-policy \
--service-namespace dynamodb \
--resource-id "table/TestTable" \
--scalable-dimension "dynamodb:table:WriteCapacityUnits" \
--policy-name "MyScalingPolicy"
```

2. Deregister the scalable target.

```
aws application-autoscaling deregister-scaling-target \
--service-namespace dynamodb \
--resource-id "table/TestTable" \
--scalable-dimension "dynamodb:table:WriteCapacityUnits"
```

3. Delete the TestTable table.

```
aws dynamodb delete-table --table-name TestTable
```

Using the AWS SDK to configure auto scaling on Amazon DynamoDB tables

In addition to using the AWS Management Console and the AWS Command Line Interface (AWS CLI), you can write applications that interact with Amazon DynamoDB auto scaling. This section contains two Java programs that you can use to test this functionality:

- `EnableDynamoDBAutoscaling.java`
- `DisableDynamoDBAutoscaling.java`

Enabling Application Auto Scaling for a table

The following program shows an example of setting up an auto scaling policy for a DynamoDB table (TestTable). It proceeds as follows:

- The program registers write capacity units as a scalable target for TestTable. The range for this metric is between 5 and 10 write capacity units.
- After the scalable target is created, the program builds a target tracking configuration. The policy seeks to maintain a 50 percent target ratio between consumed write capacity and provisioned write capacity.
- The program then creates the scaling policy, based on the target tracking configuration.

 **Note**

When you manually remove a table or global table replica, you do not automatically remove any associated scalable targets, scaling policies, or CloudWatch alarms.

The program requires that you provide an Amazon Resource Name (ARN) for a valid Application Auto Scaling service linked role. (For example: arn:aws:iam::122517410325:role/AWSServiceRoleForApplicationAutoScaling_DynamoDBTable.) In the following program, replace SERVICE_ROLE_ARN_Goes_Here with the actual ARN.

```
package com.amazonaws.codesamples.autoscaling;

import com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import
com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClientBuilder;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.MetricType;
import com.amazonaws.services.applicationautoscaling.model.PolicyType;
import
com.amazonaws.services.applicationautoscaling.model.PredefinedMetricSpecification;
```

```
import com.amazonaws.services.applicationautoscaling.model.PutScalingPolicyRequest;
import
com.amazonaws.services.applicationautoscaling.model.RegisterScalableTargetRequest;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;
import
com.amazonaws.services.applicationautoscaling.model.TargetTrackingScalingPolicyConfiguration;

public class EnableDynamoDBAutoscaling {

    static AWSApplicationAutoScalingClient aaClient = (AWSApplicationAutoScalingClient)
AWSApplicationAutoScalingClientBuilder
    .standard().build();

    public static void main(String args[]) {

        ServiceNamespace ns = ServiceNamespace.Dynamodb;
        ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
        String resourceID = "table/TestTable";

        // Define the scalable target
        RegisterScalableTargetRequest rstRequest = new RegisterScalableTargetRequest()
            .withServiceNamespace(ns)
            .withResourceId(resourceID)
            .withScalableDimension(tableWCUs)
            .withMinCapacity(5)
            .withMaxCapacity(10)
            .withRoleARN("SERVICE_ROLE_ARN_Goes_Here");

        try {
            aaClient.registerScalableTarget(rstRequest);
        } catch (Exception e) {
            System.err.println("Unable to register scalable target: ");
            System.err.println(e.getMessage());
        }

        // Verify that the target was created
        DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
            .withServiceNamespace(ns)
            .withScalableDimension(tableWCUs)
            .withResourceIds(resourceID);
        try {
            DescribeScalableTargetsResult dsaResult =
aaClient.describeScalableTargets(dscRequest);
```

```
System.out.println("DescribeScalableTargets result: ");
System.out.println(dsaResult);
System.out.println();
} catch (Exception e) {
    System.err.println("Unable to describe scalable target: ");
    System.err.println(e.getMessage());
}

System.out.println();

// Configure a scaling policy
TargetTrackingScalingPolicyConfiguration targetTrackingScalingPolicyConfiguration =
new TargetTrackingScalingPolicyConfiguration()
    .withPredefinedMetricSpecification(
        new PredefinedMetricSpecification()
            .withPredefinedMetricType(MetricType.DynamoDBWriteCapacityUtilization))
    .withTargetValue(50.0)
    .withScaleInCooldown(60)
    .withScaleOutCooldown(60);

// Create the scaling policy, based on your configuration
PutScalingPolicyRequest pspRequest = new PutScalingPolicyRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID)
    .withPolicyName("MyScalingPolicy")
    .withPolicyType(PolicyType.TargetTrackingScaling)

    .withTargetTrackingScalingPolicyConfiguration(targetTrackingScalingPolicyConfiguration);

try {
    aaClient.putScalingPolicy(pspRequest);
} catch (Exception e) {
    System.err.println("Unable to put scaling policy: ");
    System.err.println(e.getMessage());
}

// Verify that the scaling policy was created
DescribeScalingPoliciesRequest dspRequest = new DescribeScalingPoliciesRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID);

try {
```

```
    DescribeScalingPoliciesResult dspResult =
aaClient.describeScalingPolicies(dspRequest);
System.out.println("DescribeScalingPolicies result: ");
System.out.println(dspResult);
} catch (Exception e) {
e.printStackTrace();
System.err.println("Unable to describe scaling policy: ");
System.err.println(e.getMessage());
}

}

}
```

Disabling Application Auto Scaling for a table

The following program reverses the previous process. It removes the auto scaling policy and then deregisters the scalable target.

```
package com.amazonaws.codesamples.autoscaling;

import com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import com.amazonaws.services.applicationautoscaling.model.DeleteScalingPolicyRequest;
import
com.amazonaws.services.applicationautoscaling.model.DeregisterScalableTargetRequest;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import
com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;

public class DisableDynamoDBAutoscaling {

static AWSApplicationAutoScalingClient aaClient = new
AWSApplicationAutoScalingClient();

public static void main(String args[]) {
```

```
ServiceNamespace ns = ServiceNamespace.Dynamodb;
ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
String resourceId = "table/TestTable";

// Delete the scaling policy
DeleteScalingPolicyRequest delSPRequest = new DeleteScalingPolicyRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceId)
    .withPolicyName("MyScalingPolicy");

try {
    aaClient.deleteScalingPolicy(delSPRequest);
} catch (Exception e) {
    System.err.println("Unable to delete scaling policy: ");
    System.err.println(e.getMessage());
}

// Verify that the scaling policy was deleted
DescribeScalingPoliciesRequest descSPRequest = new DescribeScalingPoliciesRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceId);

try {
    DescribeScalingPoliciesResult dspResult =
    aaClient.describeScalingPolicies(descSPRequest);
    System.out.println("DescribeScalingPolicies result: ");
    System.out.println(dspResult);
} catch (Exception e) {
    e.printStackTrace();
    System.err.println("Unable to describe scaling policy: ");
    System.err.println(e.getMessage());
}

System.out.println();

// Remove the scalable target
DeregisterScalableTargetRequest delSTRequest = new DeregisterScalableTargetRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceId);
```

```
try {
    aaClient.deregisterScalableTarget(delSTRequest);
} catch (Exception e) {
    System.err.println("Unable to deregister scalable target: ");
    System.err.println(e.getMessage());
}

// Verify that the scalable target was removed
DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceIds(resourceID);

try {
    DescribeScalableTargetsResult dsaResult =
aaClient.describeScalableTargets(dscRequest);
    System.out.println("DescribeScalableTargets result: ");
    System.out.println(dsaResult);
    System.out.println();
} catch (Exception e) {
    System.err.println("Unable to describe scalable target: ");
    System.err.println(e.getMessage());
}

}

}
```

Adding tags and labels to resources

You can label Amazon DynamoDB resources using *tags*. Tags let you categorize your resources in different ways, for example, by purpose, owner, environment, or other criteria. Tags can help you do the following:

- Quickly identify a resource based on the tags that you assigned to it.
- See AWS bills broken down by tags.

Note

Any local secondary indexes (LSI) and global secondary indexes (GSI) related to tagged tables are labeled with the same tags automatically. Currently, DynamoDB Streams usage cannot be tagged.

Tagging is supported by AWS services like Amazon EC2, Amazon S3, DynamoDB, and more. Efficient tagging can provide cost insights by enabling you to create reports across services that carry a specific tag.

To get started with tagging, do the following:

1. Understand [Tagging restrictions in DynamoDB](#).
2. Create tags by using [Tagging resources in DynamoDB](#).
3. Use [Cost allocation reports](#) to track your AWS costs per active tag.

Finally, it is good practice to follow optimal tagging strategies. For information, see [AWS tagging strategies](#).

Tagging restrictions in DynamoDB

Each tag consists of a key and a value, both of which you define. The following restrictions apply:

- Each DynamoDB table can have only one tag with the same key. If you try to add an existing tag (same key), the existing tag value is updated to the new value.
- Tag keys and values are case sensitive.
- The maximum key length is 128 Unicode characters.
- The maximum value length is 256 Unicode characters.
- The allowed characters are letters, white space, and numbers, plus the following special characters: + - = . _ : /
- The maximum number of tags per resource is 50.
- AWS-assigned tag names and values are automatically assigned the aws: prefix, which you can't assign. AWS-assigned tag names don't count toward the tag limit of 50. User-assigned tag names have the prefix user: in the cost allocation report.

- You can't backdate the application of a tag.

Tagging resources in DynamoDB

You can use the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI) to add, list, edit, or delete tags. You can then activate these user-defined tags so that they appear on the AWS Billing and Cost Management console for cost allocation tracking. For more information, see [Cost allocation reports](#).

For bulk editing, you can also use Tag Editor on the AWS Management Console. For more information, see [Working with Tag Editor](#).

To use the DynamoDB API instead, see the following operations in the [Amazon DynamoDB API Reference](#):

- [TagResource](#)
- [UntagResource](#)
- [ListTagsOfResource](#)

Topics

- [Setting permissions to filter by tags](#)
- [Adding tags to new or existing tables \(AWS Management Console\)](#)
- [Adding tags to new or existing tables \(AWS CLI\)](#)

Setting permissions to filter by tags

To use tags to filter your table list in the DynamoDB console, make sure your user's policies include access to the following operations:

- `tag:GetTagKeys`
- `tag:GetTagValues`

You can access these operations by attaching a new IAM policy to your user by following the steps below.

1. Go to the [IAM console](#) with an Admin user.

2. Select "Policies" in the left navigation menu.
3. Select "Create policy."
4. Paste the following policy into the JSON editor.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "tag:GetTagKeys",  
                "tag:GetTagValues"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

5. Complete the wizard and assign a name to the policy (for example, TagKeysAndValuesReadAccess).
6. Select "Users" in the left navigation menu.
7. From the list, select the user you normally use to access the DynamoDB console.
8. Select "Add permissions."
9. Select "Attach existing policies directly."
10. From the list, select the policy you created previously.
11. Complete the wizard.

Adding tags to new or existing tables (AWS Management Console)

You can use the DynamoDB console to add tags to new tables when you create them, or to add, edit, or delete tags for existing tables.

To tag resources on creation (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**, and then choose **Create table**.

3. On the **Create DynamoDB table** page, provide a name and primary key. In the **Tags** section, choose **Add new tag** and enter the tags that you want to use.

For information about tag structure, see [Tagging restrictions in DynamoDB](#).

For more information about creating tables, see [Basic operations on DynamoDB tables](#).

To tag existing resources (console)

Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.

1. In the navigation pane, choose **Tables**.
2. Choose a table in the list, and then choose the **Additional settings** tab. You can add, edit, or delete your tags in the **Tags** section at the bottom of the page.

Adding tags to new or existing tables (AWS CLI)

The following examples show how to use the AWS CLI to specify tags when you create tables and indexes, and to tag existing resources.

To tag resources on creation (AWS CLI)

- The following example creates a new `Movies` table and adds the `Owner` tag with a value of `blueTeam`:

```
aws dynamodb create-table \
  --table-name Movies \
  --attribute-definitions AttributeName=Title,AttributeType=S \
  --key-schema AttributeName=Title,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --tags Key=Owner,Value=blueTeam
```

To tag existing resources (AWS CLI)

- The following example adds the `Owner` tag with a value of `blueTeam` for the `Movies` table:

```
aws dynamodb tag-resource \
  --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies \
  --tags Key=Owner,Value=blueTeam
```

To list all tags for a table (AWS CLI)

- The following example lists all the tags that are associated with the Movies table:

```
aws dynamodb list-tags-of-resource \
--resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies
```

Cost allocation reports

AWS uses tags to organize resource costs on your cost allocation report. AWS provides two types of cost allocation tags:

- An AWS-generated tag. AWS defines, creates, and applies this tag for you.
- User-defined tags. You define, create, and apply these tags.

You must activate both types of tags separately before they can appear in Cost Explorer or on a cost allocation report.

To activate AWS-generated tags:

- Sign in to the AWS Management Console and open the Billing and Cost Management console at <https://console.aws.amazon.com/billing/home#/>.
- In the navigation pane, choose **Cost Allocation Tags**.
- Under **AWS-Generated Cost Allocation Tags**, choose **Activate**.

To activate user-defined tags:

- Sign in to the AWS Management Console and open the Billing and Cost Management console at <https://console.aws.amazon.com/billing/home#/>.
- In the navigation pane, choose **Cost Allocation Tags**.
- Under **User-Defined Cost Allocation Tags**, choose **Activate**.

After you create and activate tags, AWS generates a cost allocation report with your usage and costs grouped by your active tags. The cost allocation report includes all of your AWS costs for each billing period. The report includes both tagged and untagged resources, so that you can clearly organize the charges for resources.

Note

Currently, any data transferred out from DynamoDB won't be broken down by tags on cost allocation reports.

For more information, see [Using cost allocation tags](#).

Working with DynamoDB tables in Java

You can use the AWS SDK for Java to create, update, and delete Amazon DynamoDB tables, list all the tables in your account, or get information about a specific table.

Topics

- [Creating a table](#)
- [Updating a table](#)
- [Deleting a table](#)
- [Listing tables](#)
- [Example: Create, update, delete, and list tables using the AWS SDK for Java document API](#)

Creating a table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values. The following code snippet creates an example table that uses a numeric type attribute ID as its primary key.

To create a table using the AWS SDK for Java API

1. Create an instance of the DynamoDB class.
2. Instantiate a `CreateTableRequest` to provide the request information.

You must provide the table name, attribute definitions, key schema, and provisioned throughput values.

3. Run the `createTable` method by providing the request object as a parameter.

The following code example demonstrates the preceding steps.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

List<AttributeDefinition> attributeDefinitions= new ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new
    KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH));

CreateTableRequest request = new CreateTableRequest()
    .withTableName(tableName)
    .withKeySchema(keySchema)
    .withAttributeDefinitions(attributeDefinitions)
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits(5L)
        .withWriteCapacityUnits(6L)));

Table table = dynamoDB.createTable(request);

table.waitForActive();
```

The table is not ready for use until DynamoDB creates it and sets its status to *ACTIVE*. The `createTable` request returns a `Table` object that you can use to obtain more information about the table.

Example

```
TableDescription tableDescription =
    dynamoDB.getTable(tableName).describe();

System.out.printf("%s: %s \t ReadCapacityUnits: %d \t WriteCapacityUnits: %d",
    tableDescription.getTableStatus(),
    tableDescription.getTableName(),
    tableDescription.getProvisionedThroughput().getReadCapacityUnits(),
    tableDescription.getProvisionedThroughput().getWriteCapacityUnits());
```

You can call the `describe` method of the client to get table information at any time.

Example

```
TableDescription tableDescription = dynamoDB.getTable(tableName).describe();
```

Updating a table

You can update only the provisioned throughput values of an existing table. Depending on your application requirements, you might need to update these values.

Note

For more information about throughput increases and decreases per day, see [Service, account, and table quotas in Amazon DynamoDB](#).

To update a table using the AWS SDK for Java API

1. Create an instance of the Table class.
2. Create an instance of the ProvisionedThroughput class to provide the new throughput values.
3. Run the updateTable method by providing the ProvisionedThroughput instance as a parameter.

The following code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()
    .withReadCapacityUnits(15L)
    .withWriteCapacityUnits(12L);

table.updateTable(provisionedThroughput);

table.waitForActive();
```

Deleting a table

To delete a table using the AWS SDK for Java API

1. Create an instance of the Table class.
2. Create an instance of the DeleteTableRequest class and provide the table name that you want to delete.
3. Run the deleteTable method by providing the Table instance as a parameter.

The following code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

table.delete();

table.waitForDelete();
```

Listing tables

To list tables in your account, create an instance of DynamoDB and run the `listTables` method. The [ListTables](#) operation requires no parameters.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableCollection<ListTablesResult> tables = dynamoDB.listTables();
Iterator<Table> iterator = tables.iterator();

while (iterator.hasNext()) {
    Table table = iterator.next();
    System.out.println(table.getTableName());
}
```

Example: Create, update, delete, and list tables using the AWS SDK for Java document API

The following code example uses the AWS SDK for Java Document API to create, update, and delete an Amazon DynamoDB table (ExampleTable). As part of the table update, it increases the provisioned throughput values. The example also lists all the tables in your account and gets the description of a specific table. For step-by-step instructions to run the following example, see [Java code examples](#).

```
package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.TableCollection;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ListTablesResult;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.TableDescription;

public class DocumentAPITableExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "ExampleTable";

    public static void main(String[] args) throws Exception {

        createExampleTable();
        listMyTables();
        getTableInformation();
        updateExampleTable();
    }
}
```

```
        deleteExampleTable();
    }

    static void createExampleTable() {

        try {

            List<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
            attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

            List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
            keySchema.add(new
KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition

            // key

            CreateTableRequest request = new
CreateTableRequest().withTableName(tableName).withKeySchema(keySchema)

.withAttributeDefinitions(attributeDefinitions).withProvisionedThroughput(
            new
ProvisionedThroughput().withReadCapacityUnits(5L).withWriteCapacityUnits(6L));

            System.out.println("Issuing CreateTable request for " + tableName);
            Table table = dynamoDB.createTable(request);

            System.out.println("Waiting for " + tableName + " to be created...this may
take a while...");
            table.waitForActive();

            getTableInformation();

        } catch (Exception e) {
            System.err.println("CreateTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    static void listMyTables() {

        TableCollection<ListTablesResult> tables = dynamoDB.listTables();
```

```
Iterator<Table> iterator = tables.iterator();

System.out.println("Listing table names");

while (iterator.hasNext()) {
    Table table = iterator.next();
    System.out.println(table.getTableName());
}

static void getTableInformation() {

    System.out.println("Describing " + tableName);

    TableDescription tableDescription = dynamoDB.getTable(tableName).describe();
    System.out.format(
        "Name: %s\n" + "Status: %s \n" + "Provisioned Throughput (read
capacity units/sec): %d \n"
        + "Provisioned Throughput (write capacity units/sec): %d \n",
        tableDescription.getTableName(), tableDescription.getTableStatus(),
        tableDescription.getProvisionedThroughput().getReadCapacityUnits(),
        tableDescription.getProvisionedThroughput().getWriteCapacityUnits());
}

static void updateExampleTable() {

    Table table = dynamoDB.getTable(tableName);
    System.out.println("Modifying provisioned throughput for " + tableName);

    try {
        table.updateTable(new
ProvisionedThroughput().withReadCapacityUnits(6L).withWriteCapacityUnits(7L));

        table.waitForActive();
    } catch (Exception e) {
        System.err.println("UpdateTable request failed for " + tableName);
        System.err.println(e.getMessage());
    }
}

static void deleteExampleTable() {

    Table table = dynamoDB.getTable(tableName);
    try {
```

```
        System.out.println("Issuing DeleteTable request for " + tableName);
        table.delete();

        System.out.println("Waiting for " + tableName + " to be deleted...this may
take a while...");

        table.waitForDelete();
    } catch (Exception e) {
        System.err.println("DeleteTable request failed for " + tableName);
        System.err.println(e.getMessage());
    }
}

}
```

Working with DynamoDB tables in .NET

You can use the AWS SDK for .NET to create, update, and delete tables, list all the tables in your account, or get information about a specific table.

The following are the common steps for Amazon DynamoDB table operations using the AWS SDK for .NET.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

For example, create a `CreateTableRequest` object to create a table and `UpdateTableRequest` object to update an existing table.

3. Run the appropriate method provided by the client that you created in the preceding step.

 **Note**

The examples in this section don't work with .NET core because it doesn't support synchronous methods. For more information, see [AWS asynchronous APIs for .NET](#).

Topics

- [Creating a table](#)
- [Updating a table](#)
- [Deleting a table](#)
- [Listing tables](#)
- [Example: Create, update, delete, and list tables using the AWS SDK for .NET low-level API](#)

Creating a table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values.

To create a table using the AWS SDK for .NET low-level API

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, primary key, and the provisioned throughput values.
3. Run the `AmazonDynamoDBClient.CreateTable` method by providing the request object as a parameter.

The following C# example demonstrates the preceding steps. The sample creates a table (`ProductCatalog`) that uses `Id` as the primary key and set of provisioned throughput values. Depending on your application requirements, you can update the provisioned throughput values by using the `UpdateTable` API.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new CreateTableRequest
{
    TableName = tableName,
    AttributeDefinitions = new List<AttributeDefinition>()
    {
        new AttributeDefinition
        {
            AttributeName = "Id",
            AttributeType = "N"
        }
    }
}
```

```
},
KeySchema = new List<KeySchemaElement>()
{
    new KeySchemaElement
    {
        AttributeName = "Id",
        KeyType = "HASH" //Partition key
    }
},
ProvisionedThroughput = new ProvisionedThroughput
{
    ReadCapacityUnits = 10,
    WriteCapacityUnits = 5
}
};

var response = client.CreateTable(request);
```

You must wait until DynamoDB creates the table and sets its status to ACTIVE. The `CreateTable` response includes the `TableDescription` property that provides the necessary table information.

Example

```
var result = response.CreateTableResult;
var tableDescription = result.TableDescription;
Console.WriteLine("{1}: {0} \t ReadCapacityUnits: {2} \t WriteCapacityUnits: {3}",
    tableDescription.TableStatus,
    tableDescription.TableName,
    tableDescription.ProvisionedThroughput.ReadCapacityUnits,
    tableDescription.ProvisionedThroughput.WriteCapacityUnits);

string status = tableDescription.TableStatus;
Console.WriteLine(tableName + " - " + status);
```

You can also call the `DescribeTable` method of the client to get table information at any time.

Example

```
var res = client.DescribeTable(new DescribeTableRequest{TableName = "ProductCatalog"});
```

Updating a table

You can update only the provisioned throughput values of an existing table. Depending on your application requirements, you might need to update these values.

Note

You can increase throughput capacity as often as needed, and decrease it within certain constraints. For more information about throughput increases and decreases per day, see [Service, account, and table quotas in Amazon DynamoDB](#).

To update a table using the AWS SDK for .NET low-level API

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `UpdateTableRequest` class to provide the request information.

You must provide the table name and the new provisioned throughput values.

3. Run the `AmazonDynamoDBClient.UpdateTable` method by providing the request object as a parameter.

The following C# example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ExampleTable";

var request = new UpdateTableRequest()
{
    TableName = tableName,
    ProvisionedThroughput = new ProvisionedThroughput()
    {
        // Provide new values.
        ReadCapacityUnits = 20,
        WriteCapacityUnits = 10
    }
};
var response = client.UpdateTable(request);
```

Deleting a table

Follow these steps to delete a table using the .NET low-level API.

To delete a table using the AWS SDK for .NET low-level API

1. Create an instance of the AmazonDynamoDBClient class.
2. Create an instance of the DeleteTableRequest class, and provide the table name that you want to delete.
3. Run the AmazonDynamoDBClient.DeleteTable method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ExampleTable";

var request = new DeleteTableRequest{ TableName = tableName };
var response = client.DeleteTable(request);
```

Listing tables

To list tables in your account using the AWS SDK for .NET low-level API, create an instance of the AmazonDynamoDBClient and run the `ListTables` method.

The [ListTables](#) operation requires no parameters. However, you can specify optional parameters. For example, you can set the `Limit` parameter if you want to use paging to limit the number of table names per page. This requires you to create a `ListTablesRequest` object and provide optional parameters as shown in the following C# example. Along with the page size, the request sets the `ExclusiveStartTableName` parameter. Initially, `ExclusiveStartTableName` is null. However, after fetching the first page of results, to retrieve the next page of results, you must set this parameter value to the `LastEvaluatedTableName` property of the current result.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

// Initial value for the first page of table names.
```

```
string lastEvaluatedTableName = null;
do
{
    // Create a request object to specify optional parameters.
    var request = new ListTablesRequest
    {
        Limit = 10, // Page size.
        ExclusiveStartTableName = lastEvaluatedTableName
    };

    var response = client.ListTables(request);
    ListTablesResult result = response.ListTablesResult;
    foreach (string name in result.TableNames)
        Console.WriteLine(name);

    lastEvaluatedTableName = result.LastEvaluatedTableName;

} while (lastEvaluatedTableName != null);
```

Example: Create, update, delete, and list tables using the AWS SDK for .NET low-level API

The following C# example creates, updates, and deletes a table (ExampleTable). It also lists all the tables in your account and gets the description of a specific table. The table update increases the provisioned throughput values. For step-by-step instructions to test the following example, see [.NET code examples](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelTableExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "ExampleTable";

        static void Main(string[] args)
        {
```

```
try
{
    CreateExampleTable();
    ListMyTables();
    GetTableInformation();
    UpdateExampleTable();

    DeleteExampleTable();

    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}
catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
catch (Exception e) { Console.WriteLine(e.Message); }
}

private static void CreateExampleTable()
{
    Console.WriteLine("\n*** Creating table ***");
    var request = new CreateTableRequest
    {
        AttributeDefinitions = new List<AttributeDefinition>()
    {
        new AttributeDefinition
        {
            AttributeName = "Id",
            AttributeType = "N"
        },
        new AttributeDefinition
        {
            AttributeName = "ReplyDateTime",
            AttributeType = "N"
        }
    },
    KeySchema = new List<KeySchemaElement>
    {
        new KeySchemaElement
        {
            AttributeName = "Id",
            KeyType = "HASH" //Partition key
        },
        new KeySchemaElement
        {

```

```
        AttributeName = "ReplyDateTime",
        KeyType = "RANGE" //Sort key
    }
},
ProvisionedThroughput = new ProvisionedThroughput
{
    ReadCapacityUnits = 5,
    WriteCapacityUnits = 6
},
TableName = tableName
};

var response = client.CreateTable(request);

var tableDescription = response.TableDescription;
Console.WriteLine("{1}: {0} \t ReadsPerSec: {2} \t WritesPerSec: {3}",
    tableDescription.TableStatus,
    tableDescription.TableName,
    tableDescription.ProvisionedThroughput.ReadCapacityUnits,
    tableDescription.ProvisionedThroughput.WriteCapacityUnits);

string status = tableDescription.TableStatus;
Console.WriteLine(tableName + " - " + status);

WaitUntilTableReady(tableName);
}

private static void ListMyTables()
{
    Console.WriteLine("\n*** listing tables ***");
    string lastTableNameEvaluated = null;
    do
    {
        var request = new ListTablesRequest
        {
            Limit = 2,
            ExclusiveStartTableName = lastTableNameEvaluated
        };

        var response = client.ListTables(request);
        foreach (string name in response.TableNames)
            Console.WriteLine(name);

        lastTableNameEvaluated = response.LastEvaluatedTableName;
```

```
        } while (lastTableNameEvaluated != null);
    }

private static void GetTableInformation()
{
    Console.WriteLine("\n*** Retrieving table information ***");
    var request = new DescribeTableRequest
    {
        TableName = tableName
    };

    var response = client.DescribeTable(request);

    TableDescription description = response.Table;
    Console.WriteLine("Name: {0}", description.TableName);
    Console.WriteLine("# of items: {0}", description.ItemCount);
    Console.WriteLine("Provision Throughput (reads/sec): {0}",
                      description.ProvisionedThroughput.ReadCapacityUnits);
    Console.WriteLine("Provision Throughput (writes/sec): {0}",
                      description.ProvisionedThroughput.WriteCapacityUnits);
}

private static void UpdateExampleTable()
{
    Console.WriteLine("\n*** Updating table ***");
    var request = new UpdateTableRequest()
    {
        TableName = tableName,
        ProvisionedThroughput = new ProvisionedThroughput()
        {
            ReadCapacityUnits = 6,
            WriteCapacityUnits = 7
        }
    };

    var response = client.UpdateTable(request);

    WaitUntilTableReady(tableName);
}

private static void DeleteExampleTable()
{
    Console.WriteLine("\n*** Deleting table ***");
    var request = new DeleteTableRequest
```

```
        {
            TableName = tableName
        };

        var response = client.DeleteTable(request);

        Console.WriteLine("Table is being deleted...");
    }

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
            status = res.Table.TableStatus;
        }
        catch (ResourceNotFoundException)
        {
            // DescribeTable is eventually consistent. So you might
            // get resource not found. So we handle the potential exception.
        }
    } while (status != "ACTIVE");
}
```

Global tables - multi-Region replication for DynamoDB

Amazon DynamoDB *global tables* are a fully managed, multi-Region, and multi-active database option that delivers fast and localized read and write performance for massively scaled global applications.

Global tables provide a fully managed solution for deploying a multi-Region, multi-active database, without having to build and maintain your own replication solution. You can specify the AWS Regions where you want the tables to be available and DynamoDB will propagate ongoing data changes to all of them.

Specific benefits for using global tables include:

- Replicating your DynamoDB tables automatically across your choice of AWS Regions
- Eliminating the difficult work of replicating data between Regions and resolving update conflicts, so you can focus on your application's business logic.
- Helping your applications stay highly available even in the unlikely event of isolation or degradation of an entire Region.

DynamoDB global tables are ideal for massively scaled applications with globally dispersed users. In such an environment, users expect very fast application performance. Global tables provide automatic multi-active replication to AWS Regions worldwide. They enable you to deliver low-latency data access to your users no matter where they are located.

The following video will give you an introductory look at global tables.

You can set up global tables in the AWS Management Console or AWS CLI. Global tables use existing DynamoDB APIs, so no application changes are required. You pay only for the resources provisioned with no upfront costs or commitments.

[Global tables for inter-Regional replication](#)

Topics

- [Replicate data seamlessly across Regions with global tables](#)
- [Provide security and access for your global tables with AWS KMS](#)
- [Global tables: How it works](#)
- [Best practices and requirements for managing global tables](#)
- [Tutorial: Creating a global table](#)

- [Monitoring global tables](#)
- [Using IAM with global tables](#)
- [Determining the global table version you are using](#)
- [Upgrading global tables to Current \(2019.11.21\) version from Legacy\(2017.11.29\) version](#)

Replicate data seamlessly across Regions with global tables

Suppose you have a large customer base spread across three geographic areas—the US East Coast, the US West Coast, and Western Europe. These customers can update their profile information using your application. To satisfy this use case, you need to create three identical DynamoDB tables named `CustomerProfiles`, in three different AWS Regions where the customers are located. These three tables would be entirely separate from each other—changes to the data in one table would not be reflected in the others. Without a managed replication solution, you would have to write code to replicate the data changes. However, doing this would be a time-consuming and labor-intensive effort.

Instead of writing your own code, you could create a global table consisting of your three Region-specific `CustomerProfiles` tables. DynamoDB would then automatically replicate data changes among those tables so that changes to `CustomerProfiles` data in one Region would seamlessly propagate to the other Regions. In addition, if one of the AWS Regions were to become temporarily unavailable, your customers could still access the same `CustomerProfiles` data in the other Regions.

Note

- Region support for global tables [Global tables version 2017.11.29 \(Legacy\)](#) is limited to US East (N. Virginia), US East (Ohio), US West (N. California), US West (Oregon), Europe (Ireland), Europe (London), Europe (Frankfurt), Asia Pacific (Singapore), Asia Pacific (Sydney), Asia Pacific (Tokyo), and Asia Pacific (Seoul).
- Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the region where the write is made originally. Transactions are not supported across regions in global tables. For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) regions and perform a `TransactWriteItems` operation in the US East (N. Virginia) Region, you may observe partially completed transactions in US West (Oregon) Region as changes are replicated.

Changes will only be replicated to other regions once they have been committed in the source region.

- If you [disable an AWS Region](#), DynamoDB will remove this replica from the replication group, 20 hours after detecting the AWS Region as inaccessible. The replica will not be deleted and replication will stop from and to this region.
- You must wait 24 hours from the time you add a read replica to successfully delete a source table. If you attempt to delete a table during the first 24 hours after adding a read replica, you will receive an error message stating: "Replica cannot be deleted because it has acted as a source region for new replicas being added in the table in the last 24 hours".
- There is no performance impact on source regions when adding new replicas.
- When you change the read and write capacity of a replica, the new write capacity is reflected to other synchronized replicas but the new read capacity is not.

For information about the AWS Region availability and pricing, see [Amazon DynamoDB Pricing](#).

Provide security and access for your global tables with AWS KMS

- You can perform AWS KMS operations on your global tables by using the `AWSServiceRoleForDynamoDBReplication` service-linked role against the [customer managed key](#) or the [AWS managed key](#) used to encrypt the replica.
- If the customer managed key used to encrypt a replica is inaccessible, DynamoDB will remove this replica from the replication group. The replica will not be deleted and replication will stop from and to this region, 20 hours after detecting the KMS key as inaccessible.
- If you want to disable your [customer managed key](#) that is used to encrypt a replica table, you must do so only if the key is no longer used to encrypt a replica table. After issuing a command to delete a replica table, you must wait for the delete operation to complete and for the global table to become Active before disabling the key. Not doing so could result in partial data replication from and to the replica table.
- If you want to modify or delete your IAM role policy for the replica table, you must do so when the replica table is in the Active state. If you don't do this, creating, updating, or deleting the replica table could fail.

- Global tables are created with deletion protection disabled by default. Even when deletion protection is enabled for a global table, any replicas of that table will start with deletion protection disabled by default.
- While deletion protection is disabled for a table, it can be accidentally deleted. While deletion protection is enabled for a table, no one can delete it.
- Changing the deletion protection setting for one replica table will not update other replicas in the group.

 **Note**

Customer managed keys are not supported in [Global tables version 2017.11.29 \(Legacy\)](#).

If you want to use a customer managed key in a DynamoDB Global Table, you need to upgrade the table to [Global Tables version 2019.11.21 \(Current\)](#) and then enable it.

Global tables: How it works

The following sections describe the concepts and behavior of global tables in Amazon DynamoDB.

Global table concepts

A *global table* is a collection of one or more replica tables, all owned by a single AWS account.

A *replica table* (or *replica*, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per AWS Region. For more information about how to get started with global tables, see [Tutorial: Creating a global table](#).

When you create a DynamoDB global table, it consists of multiple replica tables (one per Region) that DynamoDB treats as a single unit. Every replica has the same table name and the same primary key schema. When an application writes data to a replica table in one Region, DynamoDB propagates the write to the other replica tables in the other AWS Regions automatically.

You can add replica tables to the global table so that it can be available in additional Regions.

With Version 2019.11.21 (Current), when you create a Global Secondary Index in one Region it is automatically replicated to the other Region(s) as well as automatically backfilled.

Common tasks

Common tasks for global tables work as follows.

You can delete a global table's replica table the same as a regular table. This will stop replication to that Region and delete the table copy kept in that Region. You cannot sever the replication and have copies of the table exist as independent entities. You can instead copy the global table to a local table in that Region, and then delete the global replica for that Region.

 **Note**

You won't be able to delete a source table until at least 24 hours after it's used to initiate a new Region. If you try to delete it too soon you will receive an error.

Conflicts can arise if applications update the same item in different Regions at about the same time. To help ensure eventual consistency, DynamoDB global tables use a "last writer wins" method to reconcile between concurrent updates. All the replicas will agree on the latest update and converge toward a state in which they all have identical data.

 **Note**

There are several ways to avoid conflicts, including:

- Only allowing writes to the table in one Region.
- Routing user traffic to different Regions according to your write policies, to ensure there are no conflicts.
- Avoiding the use of non-idempotent updates such as `Bookmark = Bookmark + 1`, in favor of static updates such as `Bookmark=25`.
- Keep in mind that when you need to route writes or reads to one Region, it's up to your application to ensure that flow is enforced.

Monitoring global tables

You can use CloudWatch to observe the metric `ReplicationLatency`. This tracks the elapsed time between when an item is written to a replica table, and when that item appears in another replica in the global table. It's expressed in milliseconds and is emitted for every source-Region

and destination-Region pair. This metric is kept at the source Region. This is the only CloudWatch metric provided by Global Tables v2.

The replication latency you will experience depends on the distance between your chosen AWS Regions, as well as other variables. If your original table was in the US West (N. California) (us-west-1) Region, a replica in a closer Region, such as the US West (Oregon) (us-west-2) Region, would have lower replication latency compared to a replica in a region which is much further away, such as the Africa (Cape Town) (af-south-1) Region. It is common to see 0.5 to 2.5 seconds of replication latency for AWS Regions within a hectometer from one another.

Note

Replication latency does not effect API latency. If you've a client and table in Region A and you add a global tables replica in Region B, the client and table in Region A will have the same latency as before adding Region B. If you call the [PutItem](#) API operation in Region B, it will eventually be available to read in Region A after a delay of approximately the ReplicationLatency statistic available in Amazon CloudWatch. Before it is replicated, you'd receive an empty response and after it's replicated, you'd receive the item; both calls would have approximately the same API latency.

Time To Live (TTL)

You can use Time To Live (TTL) to specify an attribute name whose value indicates the time of expiration for the item. This value is given as a number in seconds since the start of the Unix epoch. After that time DynamoDB can delete the item without incurring write costs.

With global tables you configure TTL in one Region, and that setting is auto replicated to the other Region(s). When an item is deleted via a TTL rule, that work is performed without consuming Write Units on the source table - but the target table(s) will incur Replicated Write Unit costs.

Be aware that if the source and target table have very low Provisioned write capacity, this may cause throttling as the TTL deletes require write capacity.

Streams and transactions with global tables

Each global table produces an independent stream based on all its writes, regardless of the origination point for those writes. You can choose to consume this DynamoDB stream in one Region or in all Regions independently.

If you want processed local writes but not replicated writes, you can add your own region attribute to each item. Then you can use a Lambda event filter to invoke only the Lambda for writes in the local Region.

Transactional operations provide ACID (Atomicity, Consistency, Isolation, and Durability) guarantees ONLY within the Region where the write is made originally. Transactions are not supported across Regions in global tables.

For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a `TransactWriteItems` operation in the US East (Ohio) Region, you may observe partially completed transactions in US West (Oregon) Region as changes are replicated. Changes will only be replicated to other Regions once they have been committed in the source Region.

 **Note**

- Global tables “write around” DynamoDB Accelerator by updating DynamoDB directly. As a result DAX will not be aware it is holding stale data. The DAX cache will only be refreshed when the cache’s TTL expires.
- Tags on global tables do not automatically propagate.

Read and write throughput

Global tables manage read and write throughput in the following ways.

- The write capacity must be the same on all table instances across Regions.
- With Version 2019.11.21 (Current), if the table is set to support autoscaling or is in on-demand mode then the write capacity is automatically kept in sync. This means that a write capacity change to one table replicates to the others.
- Read capacity can differ between Regions because reads may not be equal. When adding a global replica to a table, the capacity of the source Region is propagated. After creation you can adjust the read capacity for one replica, and this new setting is not transferred to the other side.

Consistency and conflict resolution

Any changes made to any item in any replica table are replicated to all the other replicas within the same global table. In a global table, a newly written item is usually propagated to all replica tables within a second.

With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

An application can read and write data to any replica table. If your application only uses eventually consistent reads and only issues reads against one AWS Region, it will work without any modification. However, if your application requires strongly consistent reads, it must perform all of its strongly consistent reads and writes in the same Region. DynamoDB does not support strongly consistent reads across Regions. Therefore, if you write to one Region and read from another Region, the read response might include stale data that doesn't reflect the results of recently completed writes in the other Region.

If applications update the same item in different Regions at about the same time, conflicts can arise. To help ensure eventual consistency, DynamoDB global tables use a *last writer wins* reconciliation between concurrent updates, in which DynamoDB makes a best effort to determine the last writer. This is performed at the item level. With this conflict resolution mechanism, all the replicas will agree on the latest update and converge toward a state in which they all have identical data.

Availability and durability

If a single AWS Region becomes isolated or degraded, your application can redirect to a different Region and perform reads and writes against a different replica table. You can apply custom business logic to determine when to redirect requests to other Regions.

If a Region becomes isolated or degraded, DynamoDB keeps track of any writes that have been performed but have not yet been propagated to all of the replica tables. When the Region comes back online, DynamoDB resumes propagating any pending writes from that Region to the replica tables in other Regions. It also resumes propagating writes from other replica tables to the Region that is now back online.

Best practices and requirements for managing global tables

Using Amazon DynamoDB global tables, you can replicate your table data across AWS Regions. It is important that the replica tables and secondary indexes in your global table have identical write capacity settings to ensure proper replication of data.

For future clarity, it can be useful not to put the Region in the name for any table that might someday be turned into a global table.

Warning

The table name for each global table must be unique within your AWS account.

Global tables version

To determine the version of the global table that you're using, see [Determining the global table version you are using](#).

Requirements for Managing Capacity

A global table must have throughput capacity configured one of two ways:

1. On-demand capacity mode, measured in replicated write request units (rWRUs)
2. Provisioned capacity mode with auto scaling, measured in replicated write capacity units (rWCUs)

Using provisioned capacity mode with auto scaling or on-demand capacity mode helps ensure a global table has sufficient capacity to perform replicated writes to all regions of the global table.

Note

Switching from one table capacity mode to the other capacity mode in any Region switches the mode for all replicas.

Deploying global tables

In AWS CloudFormation, each global table is controlled by a single stack in a single Region. This is regardless of the number of replicas. When you deploy your template, CloudFormation will create/update all replicas as part of a single stack operation. For this reason, you should not deploy the same AWS::DynamoDB::GlobalTable resource in multiple Regions. Doing so is unsupported and will result in errors.

If you deploy your application template in multiple Regions, you can use conditions to create the resource in only one Region. Alternatively, you can choose to define your AWS::DynamoDB::GlobalTable resources in a stack separate from your application stack and make sure it is only deployed to a single Region. For more information see [Global tables in CloudFormation](#)

A DynamoDB table is referred to by AWS::DynamoDB::Table, and a global table is AWS::DynamoDB::GlobalTable. As far as CloudFormation is concerned, this essentially makes them two different resources. As a result, one approach is to create all tables that might ever be global by using the GlobalTable construct. You can then keep them as standalone tables to start, and add them later to Regions if needed.

If you have a regular table and you want to convert it while using CloudFormation, a recommended method is to:

1. Set the deletion policy to retain.
2. Remove the table from the stack.
3. Convert the table to a Global Table in the console.
4. Import the global table as a new resource to the stack.

 **Note**

Cross-account replication is not supported at this time.

Using global tables to help handle a potential Region outage

Have or be able to quickly create independent copies of your execution stack in alternative Regions, each accessing its local DynamoDB endpoint.

Use Route53 or AWS Global Accelerator to route to the nearest healthy Region. Alternately, have the client aware of the multiple endpoints it might use.

Use health checks in each Region that will be able to determine reliably if there's any issue with the stack, including if DynamoDB is degraded. For example, don't just ping that the DynamoDB endpoint is up. Actually do a call that ensures a full successful database flow.

If the health check fail, traffic can route to other Regions (by updating the DNS entry with Route53, by having Global Accelerator route differently, or by having the client choose a different endpoint). Global tables have a good RPO (recovery point objective) because the data is continuously syncing and a good RTO (recovery time objective) because both Regions always keep a table ready for both read and write traffic.

For further information on health checks see [Health check types](#).

 **Note**

DynamoDB is a core service on which other services frequently build their control plane operations, so it's unlikely you'll encounter a scenario where DynamoDB has degraded service in a Region while other services are unimpacted.

Backing up global tables

When backing up global tables, a backup of tables in one Region should be sufficient and backing up all tables in all Regions shouldn't be required. If the purpose is to be able to recover erroneously deleted or modified data, then PITR in one Region should suffice. Similarly, when preserving a snapshot for historic purposes such as regulatory requirements then backing up in one Region should suffice. The backed up data can be replicated to multiple Regions via AWS Backup.

Replicas and calculating write units

For planning, you should take the number of writes that one Region will perform and add that to the number of writes happening for each other Region. This is critical as every write that is performed in one Region must also be performed in every replica Region. If you do not have enough capacity to handle all of the writes, capacity exceptions will occur. In addition, inter-regional replication wait times will rise.

For example, suppose that you expect 5 writes per second to your replica table in Ohio, 10 writes per second to your replica table in N. Virginia, and 5 writes per second to your replica table in

Ireland. In this case, you should expect to consume 20 rWCUs or rWRUs in each Region: Ohio, N. Virginia, and Ireland. In other words, you should expect to consume 60 rWCUs total across all three Regions.

For details about provisioned capacity with auto scaling and DynamoDB, see [Managing throughput capacity automatically with DynamoDB auto scaling](#).

Note

If a table is running in provisioned capacity mode with auto scaling, the provisioned write capacity is allowed to float within those autoscaling settings for each Region.

Tutorial: Creating a global table

This section describes how to create a global table using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Topics

- [Creating a global table \(console\)](#)
- [Creating a global table \(AWS CLI\)](#)
- [Creating a global table \(Java\)](#)

Creating a global table (console)

Follow these steps to create a global table using the console. The following example creates a global table with replica tables in United States and Europe.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>. For this example, choose the US East (Ohio) Region.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**.
 - a. For **Table name**, enter **Music**.
 - b. For **Partition key** enter **Artist**. For **Sort key** enter **SongTitle**. (**Artist** and **SongTitle** should both be strings.)

To create the table, choose **Create table**. This table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.

4. Choose the **Global Tables** tab, and then choose **Create replica**.
5. From the **Available replication Regions** dropdown, choose **US West (Oregon)**.

The console checks to ensure that a table with the same name doesn't exist in the selected Region. If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.

6. Choose **Create Replica**. This starts the table creation process in US West (Oregon);

The **Global Table** tab for the selected table (and for any other replica tables) shows that the table has been replicated in multiple Regions.

7. Now add another Region so that your global table is replicated and synchronized across the United States and Europe. To do this, repeat step 5, but this time, specify **Europe (Frankfurt)** instead of **US West (Oregon)**.
8. You should still be using the AWS Management Console in the US East (Ohio) Region. Select **Items** in the left navigation menu, select the **Music** table, then choose **Create Item**.
 - a. For **Artist**, enter **item_1**.
 - b. For **SongTitle**, enter **Song Value 1**.
 - c. To write the item, choose **Create item**.
9. After a short time, the item is replicated across all three Regions of your global table. To verify this, in the console, on the Region selector in the upper-right corner, choose **Europe (Frankfurt)**. The Music table in Europe (Frankfurt) should contain the new item.
10. Repeat step 9 and choose **US West (Oregon)** to verify replication in that region.

Creating a global table (AWS CLI)

Follow these steps to create a global table `Music` using the AWS CLI. The following example creates a global table, with replica tables in the United States and in Europe.

1. Create a new table (`Music`) in US East (Ohio), with DynamoDB Streams enabled (`NEW_AND_OLD_IMAGES`).

```
aws dynamodb create-table \
```

```
--table-name Music \
--attribute-definitions \
    AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
--key-schema \
    AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
--billing-mode PAY_PER_REQUEST \
--stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
--region us-east-2
```

2. Create an identical Music table in US East (N. Virginia).

```
aws dynamodb update-table --table-name Music --cli-input-json \
'{
  "ReplicaUpdates": [
    {
      "Create": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region=us-east-2
```

3. Repeat step 2 to create a table in Europe (Ireland) (eu-west-1).

4. You can view the list of replicas created using describe-table.

```
aws dynamodb describe-table --table-name Music --region us-east-2
```

5. To verify that replication is working, add a new item to the Music table in US East (Ohio).

```
aws dynamodb put-item \
--table-name Music \
--item '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region us-east-2
```

6. Wait for a few seconds, and then check to see whether the item has been successfully replicated to US East (N. Virginia) and Europe (Ireland).

```
aws dynamodb get-item \
```

```
--table-name Music \
--key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region us-east-1
```

```
aws dynamodb get-item \
--table-name Music \
--key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region eu-west-1
```

7. Delete the replica table in Europe (Ireland) Region.

```
aws dynamodb update-table --table-name Music --cli-input-json \
'{
  "ReplicaUpdates": [
    {
      "Delete": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}'
```

Creating a global table (Java)

The following java code sample, create a Music table in Europe (Ireland) region then creates a replica in Asia Pacific (Seoul) region.

```
package com.amazonaws.codesamples.gtv2
import java.util.logging.Logger;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AmazonDynamoDBException;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.BillingMode;
import com.amazonaws.services.dynamodbv2.model.CreateReplicationGroupMemberAction;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableRequest;
import com.amazonaws.services.dynamodbv2.model.GlobalSecondaryIndex;
```

```
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughputOverride;
import com.amazonaws.services.dynamodbv2.model.ReplicaGlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.ReplicationGroupUpdate;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
import com.amazonaws.services.dynamodbv2.model.StreamSpecification;
import com.amazonaws.services.dynamodbv2.model.StreamViewType;
import com.amazonaws.services.dynamodbv2.model.UpdateTableRequest;
import com.amazonaws.waiters.WaiterParameters;

public class App
{
    private final static Logger LOGGER = Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);

    public static void main( String[] args )
    {

        String tableName = "Music";
        String indexName = "index1";

        Regions calledRegion = Regions.EU_WEST_1;
        Regions destRegion = Regions.AP_NORTHEAST_2;

        AmazonDynamoDB ddbClient = AmazonDynamoDBClientBuilder.standard()
            .withCredentials(new ProfileCredentialsProvider("default"))
            .withRegion(calledRegion)
            .build();

        LOGGER.info("Creating a regional table - TableName: " + tableName +",
        IndexName: " + indexName + " ....");
        ddbClient.createTable(new CreateTableRequest()
            .withTableName(tableName)
            .withAttributeDefinitions(
                new AttributeDefinition()

            .withAttributeName("Artist").withAttributeType(ScalarAttributeType.S),
            new AttributeDefinition())
    }
```

```
.withAttributeName("SongTitle").withAttributeType(ScalarAttributeType.S))
    .withKeySchema(
        new
KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH),
        new
KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE))
    .withBillingMode(BillingMode.PAY_PER_REQUEST)
    .withGlobalSecondaryIndexes(new GlobalSecondaryIndex()
        .withIndexName(indexName)
        .withKeySchema(new KeySchemaElement()
            .withAttributeName("SongTitle")
            .withKeyType(KeyType.HASH)))
    .withProjection(new
Projection().withProjectionType(ProjectionType.ALL)))
    .withStreamSpecification(new StreamSpecification()
        .withStreamEnabled(true)
        .withStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES)));

    LOGGER.info("Waiting for ACTIVE table status ....");
    ddbClient.waiters().tableExists().run(new WaiterParameters<>(new
DescribeTableRequest(tableName)));

    LOGGER.info("Testing parameters for adding a new Replica in " + destRegion +
" ....");

    CreateReplicationGroupMemberAction createReplicaAction = new
CreateReplicationGroupMemberAction()
    .withRegionName(destRegion.getName())
    .withGlobalSecondaryIndexes(new ReplicaGlobalSecondaryIndex()
        .withIndexName(indexName)
        .withProvisionedThroughputOverride(new
ProvisionedThroughputOverride()
            .withReadCapacityUnits(15L))));

    ddbClient.updateTable(new UpdateTableRequest()
        .withTableName(tableName)
        .withReplicaUpdates(new ReplicationGroupUpdate()
            .withCreate(createReplicaAction.withKMSMasterKeyId(null))));
```

```
    }  
}
```

Monitoring global tables

You can use Amazon CloudWatch to monitor the behavior and performance of a global table. Amazon DynamoDB publishes `ReplicationLatency` metric for each replica in the global table.

- **ReplicationLatency**—The elapsed time between when an item is written to a replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source- and destination-Region pair.

During normal operation, `ReplicationLatency` should be fairly constant. An elevated value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. Over time, this could result in other replica tables *falling behind* because they no longer receive updates consistently. In this case, you should verify that the read capacity units (RCUs) and write capacity units (WCUs) are identical for each of the replica tables. In addition, when choosing WCU settings, follow the recommendations in [Global tables version](#).

`ReplicationLatency` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

For more information, see [DynamoDB Metrics and dimensions](#).

Using IAM with global tables

When you create a global table for the first time, Amazon DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role is named [AWSServiceRoleForDynamoDBReplication](#), and it allows DynamoDB to manage cross-Region replication for global tables on your behalf. Don't delete this service-linked role. If you do, all of your global tables will no longer function.

For more information about service-linked roles, see [Using service-linked roles in the IAM User Guide](#).

To create replica tables in DynamoDB, you must have the following permissions in the source region.

- dynamodb:UpdateTable

To create replica tables in DynamoDB, you must have the following permissions in destination regions.

- dynamodb CreateTable
- dynamodb CreateTableReplica
- dynamodb Scan
- dynamodb Query
- dynamodb UpdateItem
- dynamodb PutItem
- dynamodb GetItem
- dynamodb DeleteItem
- dynamodb BatchWriteItem

To delete replica tables in DynamoDB, you must have the following permissions in the destination regions.

- dynamodb DeleteTable
- dynamodb DeleteTableReplica

To update replica auto scaling policy through `UpdateTableReplicaAutoScaling`, you must have the following permissions in all Regions where table replicas exist

- application-autoscaling>DeleteScalingPolicy
- application-autoscaling>DeleteScheduledAction
- application-autoscaling>DeregisterScalableTarget
- application-autoscaling>DescribeScalableTargets
- application-autoscaling>DescribeScalingActivities
- application-autoscaling>DescribeScalingPolicies

- application-autoscaling:DescribeScheduledActions
- application-autoscaling:PutScalingPolicy
- application-autoscaling:PutScheduledAction
- application-autoscaling:RegisterScalableTarget

To use UpdateTimeToLive you must have permission for dynamodb:UpdateTimeToLive in all Regions where table replicas exist.

Example: Add replica

The following IAM policy grants permissions to allow you to add replicas to a global table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:CreateTable",  
                "dynamodb:DescribeTable",  
                "dynamodb:UpdateTable",  
                "dynamodb:CreateTableReplica",  
                "iam:CreateServiceLinkedRole"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example: Update AutoScaling policy

The following IAM policy grants permissions to allow you to update replica auto scaling policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": [  
                "application-autoscaling:DescribeScheduledActions",  
                "application-autoscaling:PutScalingPolicy",  
                "application-autoscaling:PutScheduledAction",  
                "application-autoscaling:RegisterScalableTarget"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

```
        "Action": [
            "application-autoscaling:RegisterScalableTarget",
            "application-autoscaling>DeleteScheduledAction",
            "application-autoscaling:DescribeScalableTargets",
            "application-autoscaling:DescribeScalingActivities",
            "application-autoscaling:DescribeScalingPolicies",
            "application-autoscaling:PutScalingPolicy",
            "application-autoscaling:DescribeScheduledActions",
            "application-autoscaling:DeleteScalingPolicy",
            "application-autoscaling:PutScheduledAction",
            "application-autoscaling:DeregisterScalableTarget"
        ],
        "Resource": "*"
    }
]
}
```

Example: Allow replica creations for a specific table name and regions

The following IAM policy grants permissions to allow table and replica creation for `Customers` table with replicas in three Regions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "dynamodb>CreateTable",
                "dynamodb>DescribeTable",
                "dynamodb>UpdateTable"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/Customers",
                "arn:aws:dynamodb:us-west-1:123456789012:table/Customers",
                "arn:aws:dynamodb:eu-east-2:123456789012:table/Customers"
            ]
        }
    ]
}
```

Determining the global table version you are using

There are two versions of DynamoDB global tables available: [Global Tables version 2019.11.21 \(Current\)](#) and [Global tables version 2017.11.29 \(Legacy\)](#). We recommend using [Global Tables version 2019.11.21 \(Current\)](#). It is more efficient and consumes less write capacity than [Global tables version 2017.11.29 \(Legacy\)](#). The advantages of the current version include:

- The source and target tables are maintained together and kept aligned automatically for throughput, TTL settings, auto scaling settings, and other useful attributes.
- Global secondary indexes are also kept aligned.
- You can dynamically add new replica tables from a table populated with data
- The metadata attributes required to control replication are hidden which helps prevent writing of them which would cause issues with replication.
- The current version supports more Regions than the legacy version, and lets you add or remove Regions to an existing table while the legacy version does not.
- [Global Tables version 2019.11.21 \(Current\)](#) is more efficient and consumes less write capacity than [Global tables version 2017.11.29 \(Legacy\)](#), and therefore is more cost effective. In specifics:
 - Inserting a new item in one Region and then replicating to other Regions requires 2 rWCUs per region for Version 2017.11.29 (Legacy), but only 1 for Version 2019.11.21 (Current).
 - Updating an item requires 2 rWCUs in the source Region and 1 then rWCUs per destination Region in Version 2017.11.29 (Legacy), but only 1 rWCUs per source or destination in Version 2019.11.21 (Current).
 - Deleting an item requires 1 rWCUs in the source Region and then 2 rWCUs per destination Region in Version 2017.11.29 (Legacy), but only 1 rWCUs per source or destination in Version 2019.11.21 (Current).

For more information see [Amazon DynamoDB Pricing](#).

Determining the version through the CLI

To find out which version of global tables you are using through the AWS CLI, check `DescribeTable` and `DescribeGlobalTable`. `DescribeTable` will show the table version if it is Version 2019.11.21 (Current), and the `DescribeGlobalTable` property will show the table version if it is Version 2017.11.29 (Legacy).

Determining the version through the console

Finding the version through the console

To find out which version of global tables you are using through the console, do the following:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table you want to use.
4. Choose the **Global Tables** tab.
5. The **Global table version** displays the version of global tables in use:

 You are using global tables version 2019.11.21. If you need to use version 2017.11.29 instead, choose "Create version 2017.11.29 replica." You can create a version 2017.11.29 replica only if you have an empty table.

[Create a version 2017.11.29 replica.](#)

To upgrade from global tables Version 2017.11.29 (Legacy) to Version 2019.11.21 (Current), follow these steps [here](#). The overall upgrade process will work without disrupting live tables, and should finish in less than an hour. For more information, see [Updating to Version 2019.11.21 \(Current\)](#)

Note

- If the **Global table version** message does not appear in the console, that means there is another table in a different Region with the same name. In this case, the current table can't be made into a global table. Either the current table must be copied to a new table with a unique name, or all other tables with the same name must be removed.
- If you are using [Global Tables version 2019.11.21 \(Current\)](#) of global tables and you also use the [Time to Live](#) feature, DynamoDB replicates TTL deletes to all replica tables. The initial TTL delete does not consume write capacity in the region in which the TTL expiry occurs. However, the replicated TTL delete to the replica table(s) consumes a replicated write capacity unit when using provisioned capacity, or replicated write when using on-demand capacity mode, in each of the replica regions and applicable charges will apply.

- In [Global Tables version 2019.11.21 \(Current\)](#), when a TTL delete occurs it is replicated to all replica regions. These replicated writes do not contain type or principalID properties. This can make it difficult to distinguish a TTL delete from a user delete in the replicated tables.

Upgrading global tables to Current (2019.11.21) version from Legacy(2017.11.29) version

There are two versions of DynamoDB global tables available: [Global Tables version 2019.11.21 \(Current\)](#) and [Global tables version 2017.11.29 \(Legacy\)](#). Customers should use version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy). To determine which version you're using, see [Determining the global table version you are using](#).

This section describes how to upgrade your global tables to *Version 2019.11.21 (Current)* using the DynamoDB console. Upgrading from Version 2017.11.29 (Legacy) to Version 2019.11.21 (Current) is a one-time action and you cannot reverse it. Currently, you can upgrade global tables using the console only.

Topics

- [Differences in behavior between Legacy and Current versions](#)
- [Upgrade prerequisites](#)
- [Required permissions for global tables upgrade](#)
- [What to expect during the upgrade](#)
- [DynamoDB Streams behavior before, during, and after upgrade](#)
- [Upgrading to Version 2019.11.21 \(Current\)](#)

Differences in behavior between Legacy and Current versions

The following list describes the differences in behavior between the Legacy and Current versions of global tables.

- Version 2019.11.21 (Current) consumes less write capacity for several DynamoDB operations compared to Version 2017.11.29 (Legacy), and therefore, is more cost-effective for most customers. The differences for these DynamoDB operations are as follows:
 - Invoking [PutItem](#) for a 1KB item in a Region and replicating to other Regions requires 2 rWRUs per region for 2017.11.29 (Legacy), but only 1 rWRU for 2019.11.21 (Current).
 - Invoking [UpdateItem](#) for a 1KB item requires 2 rWRUs in the source Region and 1 rWRU per destination Region for 2017.11.29 (Legacy), but only 1 rWRU for both source and destination Regions for 2019.11.21 (Current).
 - Invoking [DeleteItem](#) for a 1KB item requires 1 rWRU in the source Region and 2 rWRUs per destination Region for 2017.11.29 (Legacy), but only 1 rWRU for both source or destination Region for 2019.11.21 (Current).

The following table shows the rWRU consumption for 2017.11.29 (Legacy) and 2019.11.21 (Current) tables.

rWRU consumption of 2017.11.29 (Legacy) and 2019.11.21 (Current) tables for a 1KB item in two Regions

Operation	2017.11.29 (Legacy)	2019.11.21 (Current)	Savings
PutItem	4 rWRUs	2 rWRUs	50%
UpdateItem	3 rWRUs	2 rWRUs	33%
DeleteItem	3 rWRUs	2 rWRUs	33%

- Version 2017.11.29 (Legacy) is available in only 11 AWS Regions. However, Version 2019.11.21 (Current) is available in all the AWS Regions.
- You create Version 2017.11.29 (Legacy) global tables by first creating a set of empty Regional tables, then invoking the [CreateGlobalTable](#) API to form the global table. You create Version 2019.11.21 (Current) global tables by invoking the [UpdateTable](#) API to add a replica to an existing Regional table.
- Version 2017.11.29 (Legacy) requires you to empty all replicas in the table before adding a replica in a new Region (including during creation). Version 2019.11.21 (Current) supports you to add and remove replicas to Regions on a table that already contains data.
- Version 2017.11.29 (Legacy) uses the following dedicated set of control plane APIs for managing replicas:

- [CreateGlobalTable](#)
- [DescribeGlobalTable](#)
- [DescribeGlobalTableSettings](#)
- [ListGlobalTables](#)
- [UpdateGlobalTable](#)
- [UpdateGlobalTableSettings](#)

Version 2019.11.21 (Current) uses the [DescribeTable](#) and [UpdateTable](#) APIs to manage replicas.

- Version 2017.11.29 (Legacy) publishes two DynamoDB Streams records for each write. Version 2019.11.21 (Current) only publishes one DynamoDB Streams record for each write.
- Version 2017.11.29 (Legacy) populates and updates the aws:rep:deleting, aws:rep:updateregion, and aws:rep:updatetime attributes. Version 2019.11.21 (Current) does not populate or update these attributes.
- Version 2017.11.29 (Legacy) does not synchronize [Time to Live \(TTL\)](#) settings across replicas. Version 2019.11.21 (Current) synchronizes TTL settings across replicas.
- Version 2017.11.29 (Legacy) does not replicate TTL deletes to other replicas. Version 2019.11.21 (Current) replicates TTL deletes to all replicas.
- Version 2017.11.29 (Legacy) does not synchronize [auto scaling](#) settings across replicas. Version 2019.11.21 (Current) synchronizes auto scaling settings across replicas.
- Version 2017.11.29 (Legacy) does not synchronize [global secondary index \(GSI\)](#) settings across replicas. Version 2019.11.21 (Current) synchronizes GSI settings across replicas.
- Version 2017.11.29 (Legacy) does not synchronize [encryption at rest](#) settings across replicas. Version 2019.11.21 (Current) synchronizes encryption at rest settings across replicas.
- Version 2017.11.29 (Legacy) publishes the PendingReplicationCount metric. Version 2019.11.21 (Current) does not publish this metric.

Upgrade prerequisites

Before you start upgrading to Version 2019.11.21 (Current) global tables, you must fulfill the following prerequisites:

- [Time to Live \(TTL\)](#) settings on replicas are consistent across Regions.
- [Global secondary index \(GSI\)](#) definitions on replicas are consistent across Regions.
- [Encryption at rest](#) settings on replicas are consistent across Regions.

- DynamoDB auto scaling is enabled for WCUs for all replicas, or [on-demand](#) capacity mode is enabled for all replicas.
- Applications don't require the presence of the `aws:rep:deleting`, `aws:rep:updateregion`, and `aws:rep:updatetime` attributes in table items.

Required permissions for global tables upgrade

To upgrade to Version 2019.11.21 (Current), you must have

`dynamodb:UpdateGlobalTableVersion` permissions in all Regions with replicas. These permissions are required in addition to the permissions needed for accessing the DynamoDB console and viewing tables.

The following IAM policy grants permissions to upgrade any global table to Version 2019.11.21 (Current).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "dynamodb:UpdateGlobalTableVersion",  
            "Resource": "*"  
        }  
    ]  
}
```

The following IAM policy grants permissions to upgrade only the Music global table with replicas in two Regions to Version 2019.11.21 (Current).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "dynamodb:UpdateGlobalTableVersion",  
            "Resource": [  
                "arn:aws:dynamodb::123456789012:global-table/Music",  
                "arn:aws:dynamodb:ap-southeast-1:123456789012:table/Music",  
                "arn:aws:dynamodb:us-east-2:123456789012:table/Music"  
            ]  
        }  
    ]  
}
```

```
    ]  
}
```

What to expect during the upgrade

- All global table replicas will continue to process read and write traffic while upgrading.
- The upgrade process requires between a few minutes to several hours depending on the table size and number of replicas.
- During the upgrade process, the value of [TableStatus](#) will change from ACTIVE to UPDATING. You can view the status of the table by invoking the [DescribeTable](#) API, or with the **Tables** view in the [DynamoDB console](#).
- Auto scaling will not adjust the provisioned capacity settings for a global table while the table is being upgraded. We strongly recommend that you set the table to [on-demand](#) capacity mode during the upgrade.
- If you choose to use [provisioned](#) capacity mode with auto scaling during the upgrade, you must increase the minimum read and write throughput on your policies to accommodate any expected increases in traffic to avoid throttling during the upgrade.
- When the upgrade process is complete, your table status will change to ACTIVE.

DynamoDB Streams behavior before, during, and after upgrade

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
Put or Update	Source	Timestamp population happens using UpdateItem .	Timestamp population happens using PutItem .	No customer visible timestamp is generated.
		Two Streams records are generated. The first record contains the customer written attribute	Two Streams records are generated. The first record contains the customer written attribute	A single Streams record is generated containing the customer-written attributes.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
Write		s. The second record contains the aws:rep:* attributes.	s. The second record contains the aws:rep:* attributes.	
		Two rWCUs are consumed for each customer write.	Two rWCUs are consumed for each customer write.	One rWCU is consumed for each customer write.
		ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.	ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.	ReplicationLatency metric is published in CloudWatch.
Destination		Replication happens using PutItem.	Replication happens using PutItem.	Replication happens using PutItem.
		A single Streams record is generated, which contains both the customer-written attributes and the aws:rep:* attributes.	A single Streams record is generated, which contains both the customer-written attributes and the aws:rep:* attributes.	A single Streams record is generated, which contains the customer-written attributes only and no replication attributes.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		<p>One rWCU is consumed if the item exists in the destination Region.</p> <p>Two rWCUs are consumed if the item doesn't exist in the destination Region.</p>	<p>One rWCU is consumed if the item exists in the destination Region.</p> <p>Two rWCUs are consumed if the item doesn't exist in the destination Region.</p>	<p>One rWCU is consumed for each customer write.</p>
Delete	Source	<p>ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.</p>	<p>ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.</p>	<p>ReplicationLatency metric is published in CloudWatch.</p>

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		A single Streams record is generated, which contains both the customer-written attributes and the aws:rep:* attributes.	A single Streams record is generated, which contains both the customer-written attributes and the aws:rep:* attributes.	A single Streams record is generated, which contains the customer-written attributes.
		One rWCU is consumed for each customer delete.	One rWCU is consumed for each customer delete.	One rWCU is consumed for each customer delete.
		ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.	ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.	ReplicationLatency metric is published in CloudWatch.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
Destination	Two-phase deletes take place:	Deletes the item using DeleteItem.	Deletes the item using DeleteItem.	
	<ul style="list-style-type: none"> In Phase 1, UpdateItem sets the deleting flag. In Phase 2, DeleteItem deletes the item. 			
	<p>Two Streams records are generated. The first record contains the change to the aws:rep:deleting field. The second record contains the customer-written attributes and the aws:rep:* attributes.</p>	A single Stream record is generated, which contains the customer-written attributes.	A single Stream record is generated, which contains the customer-written attributes.	
	Two rWCUs are consumed for each customer delete.	One rWCU is consumed for each customer delete.	One rWCU is consumed for each customer delete.	

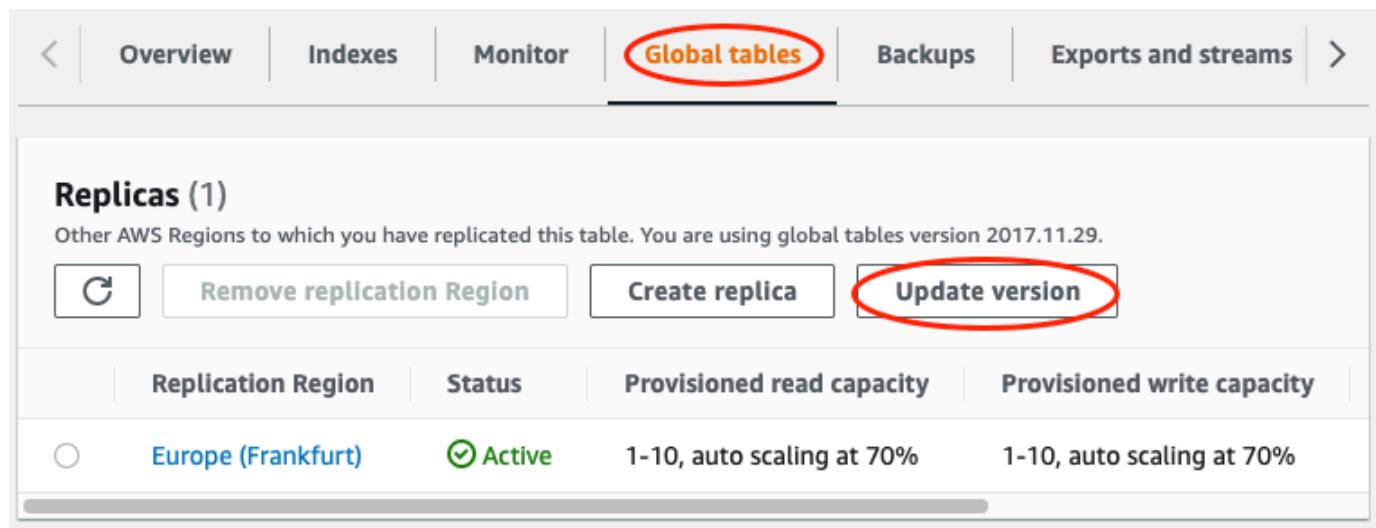
Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		ReplicationLatency and PendingReplicationCount metrics are published in CloudWatch.	ReplicationLatency metric is published in CloudWatch.	ReplicationLatency metric is published in CloudWatch.

Upgrading to Version 2019.11.21 (Current)

Perform the following steps to upgrade your version of DynamoDB global tables using the AWS Management Console.

To upgrade global tables to Version 2019.11.21 (Current)

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>.
2. In the navigation pane on the left side of the console, choose **Tables**, and then select the global table that you want to upgrade to Version 2019.11.21 (Current).
3. Choose the **Global Tables** tab.
4. Choose **Update version**.



5. Read and agree to the new requirements, and then choose **Update version**.

6. After the upgrade process is complete, the global tables version that appears on the console changes to **2019.11.21**.

Working with read and write operations

You can perform read and write operations with either the DynamoDB API or PartiQL for DynamoDB. These operations will allow you to interact with the items in your table to perform basic create, read, update, and delete (CRUD) functionality.

The following sections go more in depth on this topic.

Topics

- [DynamoDB API](#)
- [PartiQL - a SQL-compatible query language for Amazon DynamoDB](#)

DynamoDB API

Topics

- [Working with items and attributes](#)
- [Item collections - how to model one-to-many relationships in DynamoDB](#)
- [Working with scans in DynamoDB](#)

Working with items and attributes

In Amazon DynamoDB, an *item* is a collection of attributes. Each attribute has a name and a value. An attribute value can be a scalar, a set, or a document type. For more information, see [Amazon DynamoDB: How it works](#).

DynamoDB provides four operations for basic create, read, update, and delete (CRUD) functionality. All these operations are atomic.

- PutItem — Create an item.
- GetItem — Read an item.
- UpdateItem — Update an item.
- DeleteItem — Delete an item.

Each of these operations requires that you specify the primary key of the item that you want to work with. For example, to read an item using `GetItem`, you must specify the partition key and sort key (if applicable) for that item.

In addition to the four basic CRUD operations, DynamoDB also provides the following:

- `BatchGetItem` — Read up to 100 items from one or more tables.
- `BatchWriteItem` — Create or delete up to 25 items in one or more tables.

These batch operations combine multiple CRUD operations into a single request. In addition, the batch operations read and write items in parallel to minimize response latencies.

This section describes how to use these operations and includes related topics, such as conditional updates and atomic counters. This section also includes example code that uses the AWS SDKs.

Topics

- [Reading an item](#)
- [Writing an item](#)
- [Return values](#)
- [Batch operations](#)
- [Atomic counters](#)
- [Conditional writes](#)
- [Using expressions in DynamoDB](#)
- [Time to Live \(TTL\)](#)
- [Working with items: Java](#)
- [Working with items: .NET](#)

Reading an item

To read an item from a DynamoDB table, use the `GetItem` operation. You must provide the name of the table, along with the primary key of the item you want.

Example

The following AWS CLI example shows how to read an item from the `ProductCatalog` table.

```
aws dynamodb get-item \
```

```
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}'
```

Note

With `GetItem`, you must specify the *entire* primary key, not just part of it. For example, if a table has a composite primary key (partition key and sort key), you must supply a value for the partition key and a value for the sort key.

A `GetItem` request performs an eventually consistent read by default. You can use the `ConsistentRead` parameter to request a strongly consistent read instead. (This consumes additional read capacity units, but it returns the most up-to-date version of the item.)

`GetItem` returns all of the item's attributes. You can use a *projection expression* to return only some of the attributes. For more information, see [Projection expressions](#).

To return the number of read capacity units consumed by `GetItem`, set the `ReturnConsumedCapacity` parameter to `TOTAL`.

Example

The following AWS Command Line Interface (AWS CLI) example shows some of the optional `GetItem` parameters.

```
aws dynamodb get-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"1"}}' \
  --consistent-read \
  --projection-expression "Description, Price, RelatedItems" \
  --return-consumed-capacity TOTAL
```

Writing an item

To create, update, or delete an item in a DynamoDB table, use one of the following operations:

- `PutItem`
- `UpdateItem`
- `DeleteItem`

For each of these operations, you must specify the entire primary key, not just part of it. For example, if a table has a composite primary key (partition key and sort key), you must provide a value for the partition key and a value for the sort key.

To return the number of write capacity units consumed by any of these operations, set the `ReturnConsumedCapacity` parameter to one of the following:

- `TOTAL` — Returns the total number of write capacity units consumed.
- `INDEXES` — Returns the total number of write capacity units consumed, with subtotals for the table and any secondary indexes that were affected by the operation.
- `NONE` — No write capacity details are returned. (This is the default.)

PutItem

`PutItem` creates a new item. If an item with the same key already exists in the table, it is replaced with the new item.

Example

Write a new item to the `Thread` table. The primary key for `Thread` consists of `ForumName` (partition key) and `Subject` (sort key).

```
aws dynamodb put-item \
  --table-name Thread \
  --item file://item.json
```

The arguments for `--item` are stored in the `item.json` file.

```
{
  "ForumName": {"S": "Amazon DynamoDB"},
  "Subject": {"S": "New discussion thread"},
  "Message": {"S": "First post in this thread"},
  "LastPostedBy": {"S": "fred@example.com"},
  "LastPostDateTime": {"S": "201603190422"}
}
```

UpdateItem

If an item with the specified key does not exist, `UpdateItem` creates a new item. Otherwise, it modifies an existing item's attributes.

You use an *update expression* to specify the attributes that you want to modify and their new values. For more information, see [Update expressions](#).

Within the update expression, you use expression attribute values as placeholders for the actual values. For more information, see [Expression attribute values](#).

Example

Modify various attributes in the Thread item. The optional `ReturnValues` parameter shows the item as it appears after the update. For more information, see [Return values](#).

```
aws dynamodb update-item \  
  --table-name Thread \  
  --key file://key.json \  
  --update-expression "SET Answered = :zero, Replies = :zero, LastPostedBy  
= :lastpostedby" \  
  --expression-attribute-values file://expression-attribute-values.json \  
  --return-values ALL_NEW
```

The arguments for `--key` are stored in the `key.json` file.

```
{  
    "ForumName": {"S": "Amazon DynamoDB"},  
    "Subject": {"S": "New discussion thread"}  
}
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{  
    ":zero": {"N": "0"},  
    ":lastpostedby": {"S": "barney@example.com"}  
}
```

DeleteItem

`DeleteItem` deletes the item with the specified key.

Example

The following AWS CLI example shows how to delete the Thread item.

```
aws dynamodb delete-item \
    --table-name Thread \
    --key file://key.json
```

Return values

In some cases, you might want DynamoDB to return certain attribute values as they appeared before or after you modified them. The PutItem, UpdateItem, and DeleteItem operations have a `ReturnValues` parameter that you can use to return the attribute values before or after they are modified.

The default value for `ReturnValues` is `NONE`, meaning that DynamoDB does not return any information about attributes that were modified.

The following are the other valid settings for `ReturnValues`, organized by DynamoDB API operation.

PutItem

- `ReturnValues: ALL_OLD`
 - If you overwrite an existing item, `ALL_OLD` returns the entire item as it appeared before the overwrite.
 - If you write a nonexistent item, `ALL_OLD` has no effect.

UpdateItem

The most common usage for `UpdateItem` is to update an existing item. However, `UpdateItem` actually performs an *upsert*, meaning that it automatically creates the item if it doesn't already exist.

- `ReturnValues: ALL_OLD`
 - If you update an existing item, `ALL_OLD` returns the entire item as it appeared before the update.
 - If you update a nonexistent item (upsert), `ALL_OLD` has no effect.
- `ReturnValues: ALL_NEW`
 - If you update an existing item, `ALL_NEW` returns the entire item as it appeared after the update.

- If you update a nonexistent item (upsert), ALL_NEW returns the entire item.
- ReturnValues: UPDATED_OLD
 - If you update an existing item, UPDATED_OLD returns only the updated attributes, as they appeared before the update.
 - If you update a nonexistent item (upsert), UPDATED_OLD has no effect.
- ReturnValues: UPDATED_NEW
 - If you update an existing item, UPDATED_NEW returns only the affected attributes, as they appeared after the update.
 - If you update a nonexistent item (upsert), UPDATED_NEW returns only the updated attributes, as they appear after the update.

DeleteItem

- ReturnValues: ALL_OLD
 - If you delete an existing item, ALL_OLD returns the entire item as it appeared before you deleted it.
 - If you delete a nonexistent item, ALL_OLD doesn't return any data.

Batch operations

For applications that need to read or write multiple items, DynamoDB provides the BatchGetItem and BatchWriteItem operations. Using these operations can reduce the number of network round trips from your application to DynamoDB. In addition, DynamoDB performs the individual read or write operations in parallel. Your applications benefit from this parallelism without having to manage concurrency or threading.

The batch operations are essentially wrappers around multiple read or write requests. For example, if a BatchGetItem request contains five items, DynamoDB performs five GetItem operations on your behalf. Similarly, if a BatchWriteItem request contains two put requests and four delete requests, DynamoDB performs two PutItem and four DeleteItem requests.

In general, a batch operation does not fail unless *all* the requests in the batch fail. For example, suppose that you perform a BatchGetItem operation, but one of the individual GetItem requests in the batch fails. In this case, BatchGetItem returns the keys and data from the GetItem request that failed. The other GetItem requests in the batch are not affected.

BatchGetItem

A single BatchGetItem operation can contain up to 100 individual GetItem requests and can retrieve up to 16 MB of data. In addition, a BatchGetItem operation can retrieve items from multiple tables.

Example

Retrieve two items from the Thread table, using a projection expression to return only some of the attributes.

```
aws dynamodb batch-get-item \
--request-items file://request-items.json
```

The arguments for --request-items are stored in the request-items.json file.

```
{
    "Thread": {
        "Keys": [
            {
                "ForumName": {"S": "Amazon DynamoDB"},
                "Subject": {"S": "DynamoDB Thread 1"}
            },
            {
                "ForumName": {"S": "Amazon S3"},
                "Subject": {"S": "S3 Thread 1"}
            }
        ],
        "ProjectionExpression": "ForumName, Subject, LastPostedDateTime, Replies"
    }
}
```

BatchWriteItem

The BatchWriteItem operation can contain up to 25 individual PutItem and DeleteItem requests and can write up to 16 MB of data. (The maximum size of an individual item is 400 KB.) In addition, a BatchWriteItem operation can put or delete items in multiple tables.

Note

BatchWriteItem does not support UpdateItem requests.

Example

Write two items to the ProductCatalog table.

```
aws dynamodb batch-write-item \
--request-items file://request-items.json
```

The arguments for `--request-items` are stored in the `request-items.json` file.

```
{
    "ProductCatalog": [
        {
            "PutRequest": {
                "Item": {
                    "Id": { "N": "601" },
                    "Description": { "S": "Snowboard" },
                    "QuantityOnHand": { "N": "5" },
                    "Price": { "N": "100" }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": { "N": "602" },
                    "Description": { "S": "Snow shovel" }
                }
            }
        }
    ]
}
```

Atomic counters

You can use the `UpdateItem` operation to implement an *atomic counter*—a numeric attribute that is incremented, unconditionally, without interfering with other write requests. (All write requests are applied in the order in which they were received.) With an atomic counter, the updates are not idempotent. In other words, the numeric value increments or decrements each time you call `UpdateItem`. If the increment value used to update the atomic counter is positive, then it can cause overcounting. If the increment value is negative, then it can cause undercounting.

You might use an atomic counter to track the number of visitors to a website. In this case, your application would increment a numeric value, regardless of its current value. If an `UpdateItem` operation fails, the application could simply retry the operation. This would risk updating the counter twice, but you could probably tolerate a slight overcounting or undercounting of website visitors.

An atomic counter would not be appropriate where overcounting or undercounting can't be tolerated (for example, in a banking application). In this case, it is safer to use a conditional update instead of an atomic counter.

For more information, see [Incrementing and decrementing numeric attributes](#).

Example

The following AWS CLI example increments the `Price` of a product by 5. For this example, the item was known to exist before the counter is updated. Because `UpdateItem` is not idempotent, the `Price` increases every time you run this code.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id": { "N": "601" }}' \  
  --update-expression "SET Price = Price + :incr" \  
  --expression-attribute-values '{":incr":{"N":"5"}}' \  
  --return-values UPDATED_NEW
```

Conditional writes

By default, the DynamoDB write operations (`PutItem`, `UpdateItem`, `DeleteItem`) are *unconditional*: Each operation overwrites an existing item that has the specified primary key.

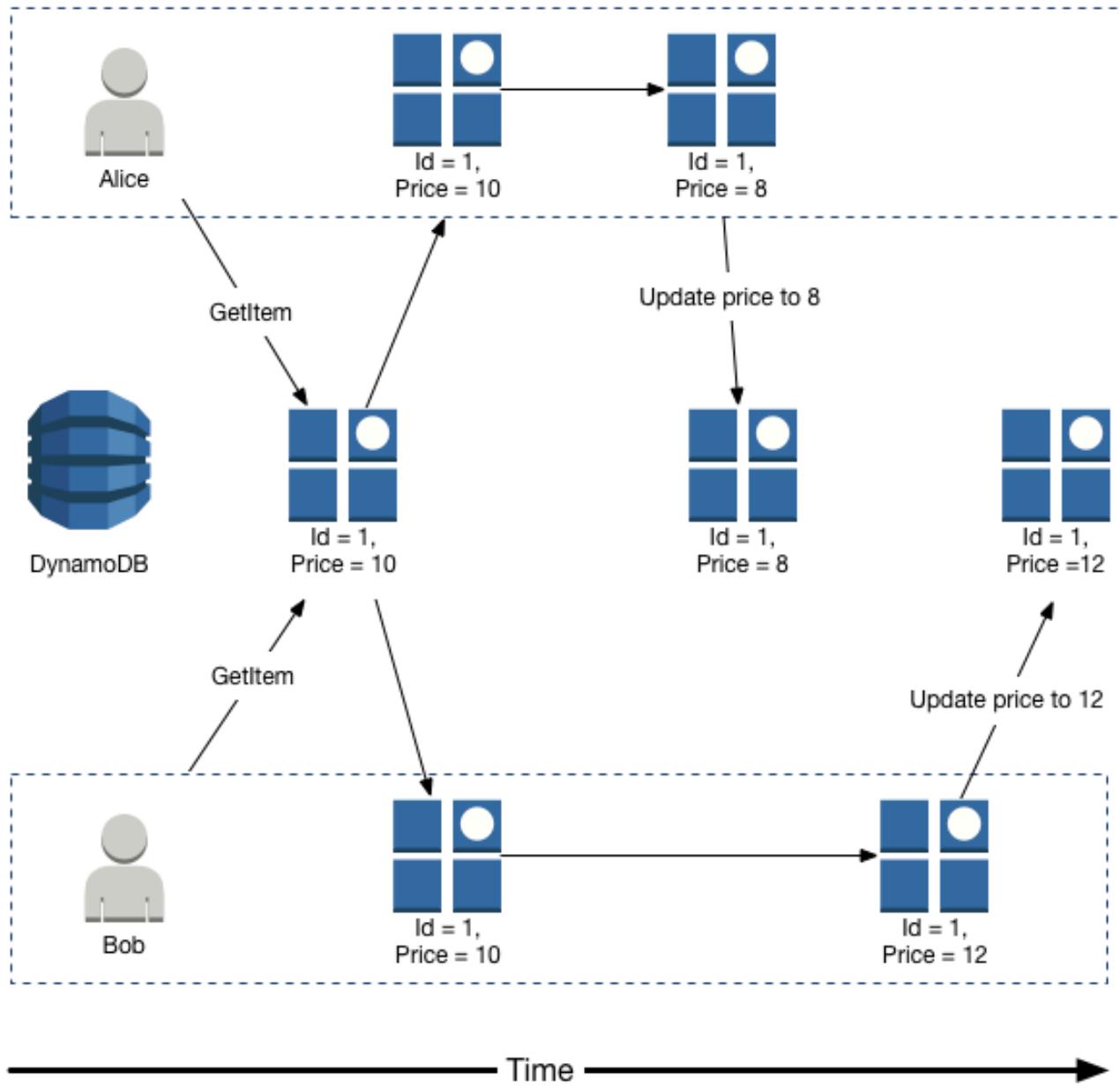
DynamoDB optionally supports conditional writes for these operations. A conditional write succeeds only if the item attributes meet one or more expected conditions. Otherwise, it returns an error.

Conditional writes check their conditions against the most recently updated version of the item. Note that if the item did not previously exist or if the most recent successful operation against that item was a delete, then the conditional write will find no previous item.

Conditional writes are helpful in many situations. For example, you might want a `PutItem` operation to succeed only if there is not already an item with the same primary key. Or you could

prevent an `UpdateItem` operation from modifying an item if one of its attributes has a certain value.

Conditional writes are helpful in cases where multiple users attempt to modify the same item. Consider the following diagram, in which two users (Alice and Bob) are working with the same item from a DynamoDB table.



Suppose that Alice uses the AWS CLI to update the Price attribute to 8.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"1"}}' \  
  --update-expression "SET Price = :newval" \  
  --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the file `expression-attribute-values.json`:

```
{  
  ":newval":{"N":"8"}  
}
```

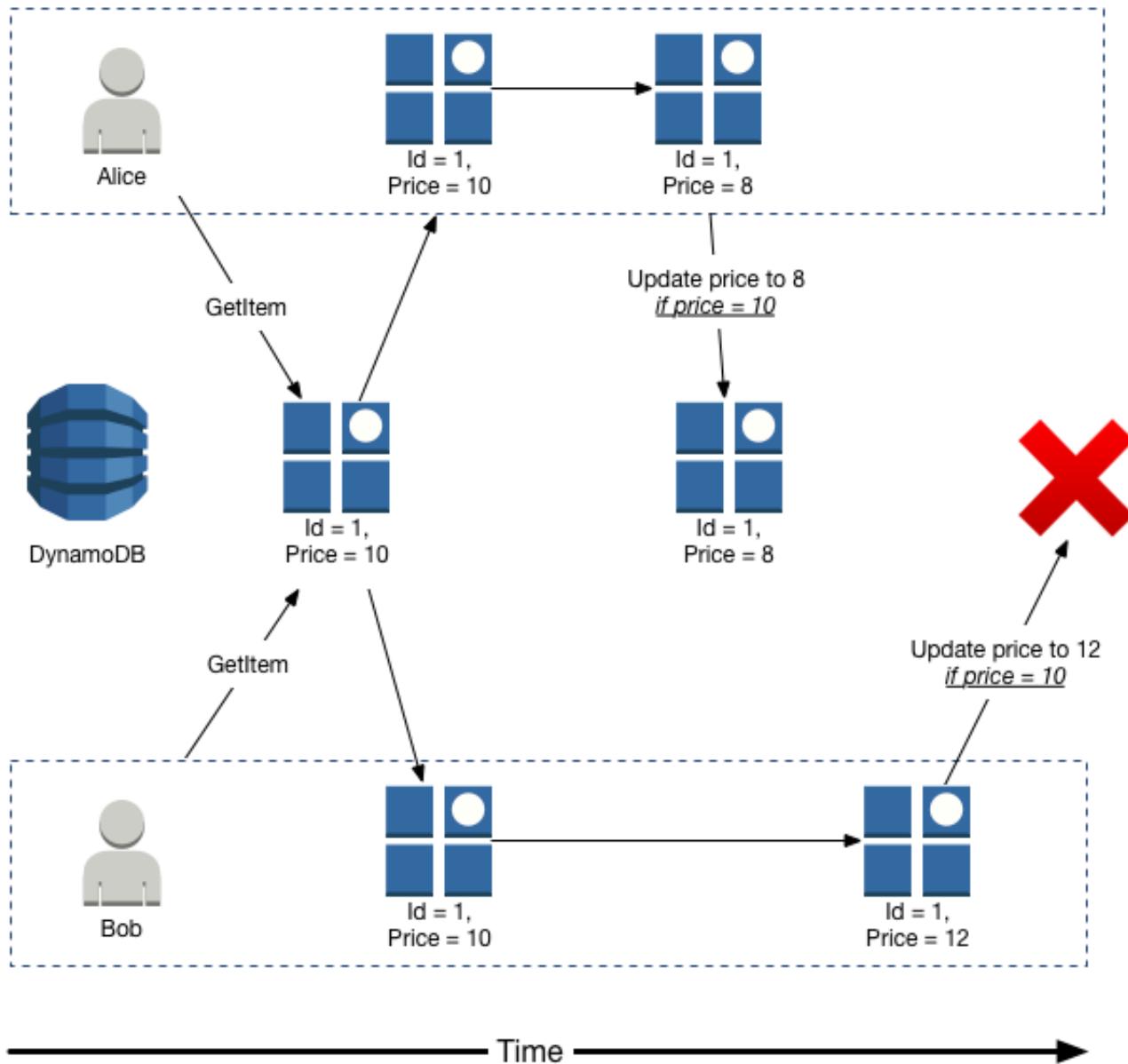
Now suppose that Bob issues a similar `UpdateItem` request later, but changes the Price to 12. For Bob, the `--expression-attribute-values` parameter looks like the following.

```
{  
  ":newval":{"N":"12"}  
}
```

Bob's request succeeds, but Alice's earlier update is lost.

To request a conditional `PutItem`, `DeleteItem`, or `UpdateItem`, you specify a condition expression. A *condition expression* is a string containing attribute names, conditional operators, and built-in functions. The entire expression must evaluate to true. Otherwise, the operation fails.

Now consider the following diagram, showing how conditional writes would prevent Alice's update from being overwritten.



Alice first tries to update Price to 8, but only if the current Price is 10.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--update-expression "SET Price = :newval" \
--condition-expression "Price = :currval" \
```

```
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{  
    ":newval": {"N": "8"},  
    ":currval": {"N": "10"}  
}
```

Alice's update succeeds because the condition evaluates to true.

Next, Bob attempts to update the Price to 12, but only if the current Price is 10. For Bob, the `--expression-attribute-values` parameter looks like the following.

```
{  
    ":newval": {"N": "12"},  
    ":currval": {"N": "10"}  
}
```

Because Alice has previously changed the Price to 8, the condition expression evaluates to false, and Bob's update fails.

For more information, see [Condition expressions](#).

Conditional write idempotence

Conditional writes can be *idempotent* if the conditional check is on the same attribute that is being updated. This means that DynamoDB performs a given write request only if certain attribute values in the item match what you expect them to be at the time of the request.

For example, suppose that you issue an `UpdateItem` request to increase the Price of an item by 3, but only if the Price is currently 20. After you send the request, but before you get the results back, a network error occurs, and you don't know whether the request was successful. Because this conditional write is idempotent, you can retry the same `UpdateItem` request, and DynamoDB updates the item only if the Price is currently 20.

Capacity units consumed by conditional writes

If a `ConditionExpression` evaluates to false during a conditional write, DynamoDB still consumes write capacity from the table. The amount consumed is dependent on the size of the

existing item (or a minimum of 1). For example, if an existing item is 300kb and the new item you are trying to create or update is 310kb, the write capacity units consumed will be the 300 if the condition fails, and 310 if the condition succeeds. If this is a new item (no existing item), then the write capacity units consumed will be 1 if the condition fails and 310 if the condition succeeds.

 **Note**

Write operations consume *write* capacity units only. They never consume *read* capacity units.

A failed conditional write returns a `ConditionalCheckFailedException`. When this occurs, you don't receive any information in the response about the write capacity that was consumed. However, you can view the `ConsumedWriteCapacityUnits` metric for the table in Amazon CloudWatch. For more information, see [DynamoDB metrics](#) in [Logging and monitoring in DynamoDB](#).

To return the number of write capacity units consumed during a conditional write, you use the `ReturnConsumedCapacity` parameter:

- `TOTAL` — Returns the total number of write capacity units consumed.
- `INDEXES` — Returns the total number of write capacity units consumed, with subtotals for the table and any secondary indexes that were affected by the operation.
- `NONE` — No write capacity details are returned. (This is the default.)

 **Note**

Unlike a global secondary index, a local secondary index shares its provisioned throughput capacity with its table. Read and write activity on a local secondary index consumes provisioned throughput capacity from the table.

Using expressions in DynamoDB

In Amazon DynamoDB, you use *expressions* to denote the attributes that you want to read from an item. You also use expressions when writing an item to indicate any conditions that must be met

(also known as a conditional update), and to indicate how the attributes are to be updated. This section describes the basic expression grammar and the available kinds of expressions.

Note

For backward compatibility, DynamoDB also supports conditional parameters that do not use expressions. For more information, see [Legacy conditional parameters](#).

New applications should use expressions rather than the legacy parameters.

Topics

- [Specifying item attributes when using expressions](#)
- [Projection expressions](#)
- [Expression attribute names in DynamoDB](#)
- [Expression attribute values](#)
- [Condition expressions](#)
- [Update expressions](#)

Specifying item attributes when using expressions

This section describes how to refer to item attributes in an expression in Amazon DynamoDB. You can work with any attribute, even if it is deeply nested within multiple lists and maps.

Topics

- [Top-level attributes](#)
- [Nested attributes](#)
- [Document paths](#)

A Sample Item: ProductCatalog

The following is a representation of an item in the ProductCatalog table. (This table is described in [Example tables and data](#).)

```
{  
  "Id": 123,
```

```
"Title": "Bicycle 123",
"Description": "123 description",
"BicycleType": "Hybrid",
"Brand": "Brand-Company C",
"Price": 500,
"Color": ["Red", "Black"],
"ProductCategory": "Bicycle",
"InStock": true,
"QuantityOnHand": null,
"RelatedItems": [
    341,
    472,
    649
],
"Pictures": {
    "FrontView": "http://example.com/products/123_front.jpg",
    "RearView": "http://example.com/products/123_rear.jpg",
    "SideView": "http://example.com/products/123_left_side.jpg"
},
"ProductReviews": {
    "FiveStar": [
        "Excellent! Can't recommend it highly enough! Buy it!",
        "Do yourself a favor and buy this."
    ],
    "OneStar": [
        "Terrible product! Do not buy this."
    ]
},
"Comment": "This product sells out quickly during the summer",
"Safety.Warning": "Always wear a helmet"
}
```

Note the following:

- The partition key value (**Id**) is 123. There is no sort key.
- Most of the attributes have scalar data types, such as **String**, **Number**, **Boolean**, and **Null**.
- One attribute (**Color**) is a **String Set**.
- The following attributes are document data types:
 - A list of **RelatedItems**. Each element is an **Id** for a related product.
 - A map of **Pictures**. Each element is a short description of a picture, along with a URL for the corresponding image file.

- A map of `ProductReviews`. Each element represents a rating and a list of reviews corresponding to that rating. Initially, this map is populated with five-star and one-star reviews.

Top-level attributes

An attribute is said to be *top level* if it is not embedded within another attribute. For the `ProductCatalog` item, the top-level attributes are as follows:

- `Id`
- `Title`
- `Description`
- `BicycleType`
- `Brand`
- `Price`
- `Color`
- `ProductCategory`
- `InStock`
- `QuantityOnHand`
- `RelatedItems`
- `Pictures`
- `ProductReviews`
- `Comment`
- `Safety.Warning`

All of these top-level attributes are scalars, except for `Color` (list), `RelatedItems` (list), `Pictures` (map), and `ProductReviews` (map).

Nested attributes

An attribute is said to be *nested* if it is embedded within another attribute. To access a nested attribute, you use *dereference operators*:

- `[n]` — for list elements

- . (dot) — for map elements

Accessing list elements

The dereference operator for a list element is `[N]`, where n is the element number. List elements are zero-based, so `[0]` represents the first element in the list, `[1]` represents the second, and so on. Here are some examples:

- `MyList[0]`
- `AnotherList[12]`
- `ThisList[5][11]`

The element `ThisList[5]` is itself a nested list. Therefore, `ThisList[5][11]` refers to the 12th element in that list.

The number within the square brackets must be a non-negative integer. Therefore, the following expressions are not valid:

- `MyList[-1]`
- `MyList[0.4]`

Accessing map elements

The dereference operator for a map element is `.` (a dot). Use a dot as a separator between elements in a map:

- `MyMap.nestedField`
- `MyMap.nestedField.deeplyNestedField`

Document paths

In an expression, you use a *document path* to tell DynamoDB where to find an attribute. For a top-level attribute, the document path is simply the attribute name. For a nested attribute, you construct the document path using dereference operators.

The following are some examples of document paths. (Refer to the item shown in [Specifying item attributes when using expressions](#).)

- A top-level scalar attribute.

Description

- A top-level list attribute. (This returns the entire list, not just some of the elements.)

RelatedItems

- The third element from the RelatedItems list. (Remember that list elements are zero-based.)

RelatedItems[2]

- The front-view picture of the product.

Pictures.FrontView

- All of the five-star reviews.

ProductReviews.FiveStar

- The first of the five-star reviews.

ProductReviews.FiveStar[0]

Note

The maximum depth for a document path is 32. Therefore, the number of dereferences operators in a path cannot exceed this limit.

You can use any attribute name in a document path as long as they meet these requirements:

- The attribute name must begin with a pound sign (#)
- The first character is a-z or A-Z and or 0-9
- The second character (if present) is a-z, A-Z

Note

If an attribute name does not meet this requirement, you must define an expression attribute name as a placeholder.

For more information, see [Expression attribute names in DynamoDB](#).

Projection expressions

To read data from a table, you use operations such as GetItem, Query, or Scan. Amazon DynamoDB returns all the item attributes by default. To get only some, rather than all of the attributes, use a projection expression.

A *projection expression* is a string that identifies the attributes that you want. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated.

The following are some examples of projection expressions, based on the ProductCatalog item from [Specifying item attributes when using expressions](#):

- A single top-level attribute.

Title

- Three top-level attributes. DynamoDB retrieves the entire Color set.

Title, Price, Color

- Four top-level attributes. DynamoDB returns the entire contents of RelatedItems and ProductReviews.

Title, Description, RelatedItems, ProductReviews

DynamoDB has a list of reserved words and special characters. You can use any attribute name in a projection expression, provided that the first character is a-z or A-Z and the second character (if present) is a-z, A-Z, or 0-9. If an attribute name doesn't meet this requirement, you must define an expression attribute name as a placeholder. For a complete list, see [Reserved words in DynamoDB](#). Also, the following characters have special meaning in DynamoDB: # (hash) and : (colon).

Although DynamoDB allows you to use these reserved words and special characters for names, we recommend that you avoid doing so because you have to define placeholder variables whenever you use these names in an expression. For more information, see [Expression attribute names in DynamoDB](#).

The following AWS CLI example shows how to use a projection expression with a GetItem operation. This projection expression retrieves a top-level scalar attribute

(Description), the first element in a list (RelatedItems[0]), and a list nested within a map (ProductReviews.FiveStar).

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key file://key.json \
--projection-expression "Description, RelatedItems[0], ProductReviews.FiveStar"
```

The following JSON would be returned for this example.

```
{
  "Item": {
    "Description": {
      "S": "123 description"
    },
    "ProductReviews": {
      "M": {
        "FiveStar": {
          "L": [
            {
              "S": "Excellent! Can't recommend it highly enough! Buy it!"
            },
            {
              "S": "Do yourself a favor and buy this."
            }
          ]
        }
      }
    },
    "RelatedItems": {
      "L": [
        {
          "N": "341"
        }
      ]
    }
  }
}
```

The arguments for --key are stored in the key.json file.

```
{
```

```
"Id": { "N": "123" }  
}
```

For programming language-specific code examples, see [Getting started with DynamoDB and the AWS SDKs](#).

Expression attribute names in DynamoDB

An *expression attribute name* is a placeholder that you use in an Amazon DynamoDB expression as an alternative to an actual attribute name. An expression attribute name must begin with a pound sign (#), and be followed by one or more alphanumeric characters and the underscore (_) character.

This section describes several situations in which you must use expression attribute names.

Note

The examples in this section use the AWS Command Line Interface (AWS CLI). For programming language-specific code examples, see [Getting started with DynamoDB and the AWS SDKs](#).

Topics

- [Reserved words](#)
- [Attribute names containing special characters](#)
- [Nested attributes](#)
- [Repeating attribute names](#)

Reserved words

Sometimes you might need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word. (For a complete list of reserved words, see [Reserved words in DynamoDB](#).)

For example, the following AWS CLI example would fail because COMMENT is a reserved word.

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key-value { "id": "123" }
```

```
--key '{"Id":{"N":"123"}}' \
--projection-expression "Comment"
```

To work around this, you can replace `Comment` with an expression attribute name such as `#c`. The `#` (pound sign) is required and indicates that this is a placeholder for an attribute name. The AWS CLI example would now look like the following.

```
aws dynamodb get-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"123"}}' \
  --projection-expression "#c" \
  --expression-attribute-names '{"#c":"Comment"}'
```

Note

If an attribute name begins with a number, contains a space or contains a reserved word, you *must* use an expression attribute name to replace that attribute's name in the expression.

Attribute names containing special characters

In an expression, a dot (".") is interpreted as a separator character in a document path. However, DynamoDB also allows you to use a dot character and other special characters, such as a hyphen ("-") as part of an attribute name. This can be ambiguous in some cases. To illustrate, suppose that you wanted to retrieve the `Safety.Warning` attribute from a `ProductCatalog` item (see [Specifying item attributes when using expressions](#)).

Suppose that you wanted to access `Safety.Warning` using a projection expression.

```
aws dynamodb get-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"123"}}' \
  --projection-expression "Safety.Warning"
```

DynamoDB would return an empty result, rather than the expected string ("Always wear a helmet"). This is because DynamoDB interprets a dot in an expression as a document path separator. In this case, you must define an expression attribute name (such as `#sw`) as a substitute for `Safety.Warning`. You could then use the following projection expression.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "#sw" \
--expression-attribute-names '{"#sw":"Safety.Warning"}'
```

DynamoDB would then return the correct result.

 **Note**

If an attribute name contains a dot (".") or a hyphen ("-"), you *must* use an expression attribute name to replace that attribute's name in the expression.

Nested attributes

Suppose that you wanted to access the nested attribute `ProductReviews.OneStar`, using the following projection expression.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "ProductReviews.OneStar"
```

The result would contain all of the one-star product reviews, which is expected.

But what if you decided to use an expression attribute name instead? For example, what would happen if you were to define `#pr1star` as a substitute for `ProductReviews.OneStar`?

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "#pr1star" \
--expression-attribute-names '{"#pr1star":"ProductReviews.OneStar"}'
```

DynamoDB would return an empty result instead of the expected map of one-star reviews. This is because DynamoDB interprets a dot in an expression attribute name as a character within an attribute's name. When DynamoDB evaluates the expression attribute name `#pr1star`, it determines that `ProductReviews.OneStar` refers to a scalar attribute—which is not what was intended.

The correct approach would be to define an expression attribute name for each element in the document path:

- `#pr` – `ProductReviews`
- `#1star` – `OneStar`

You could then use `#pr.#1star` for the projection expression.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "#pr.#1star" \
--expression-attribute-names '{"#pr":"ProductReviews", "#1star": "OneStar"}'
```

DynamoDB would then return the correct result.

Repeating attribute names

Expression attribute names are helpful when you need to refer to the same attribute name repeatedly. For example, consider the following expression for retrieving some of the reviews from a `ProductCatalog` item.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "ProductReviews.FiveStar, ProductReviews.ThreeStar,
ProductReviews.OneStar"
```

To make this more concise, you can replace `ProductReviews` with an expression attribute name such as `#pr`. The revised expression would now look like the following.

- `#pr.FiveStar, #pr.ThreeStar, #pr.OneStar`

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "#pr.FiveStar, #pr.ThreeStar, #pr.OneStar" \
--expression-attribute-names '{"#pr":"ProductReviews"}'
```

If you define an expression attribute name, you must use it consistently throughout the entire expression. Also, you cannot omit the # symbol.

Expression attribute values

If you need to compare an attribute with a value, define an expression attribute value as a placeholder. *Expression attribute values* in Amazon DynamoDB are substitutes for the actual values that you want to compare—values that you might not know until runtime. An expression attribute value must begin with a colon (:) and be followed by one or more alphanumeric characters.

For example, suppose that you wanted to return all of the ProductCatalog items that are available in Black and cost 500 or less. You could use a Scan operation with a filter expression, as in this AWS Command Line Interface (AWS CLI) example.

```
aws dynamodb scan \
--table-name ProductCatalog \
--filter-expression "contains(Color, :c) and Price <= :p" \
--expression-attribute-values file://values.json
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{
  ":c": { "S": "Black" },
  ":p": { "N": "500" }
}
```

Note

A Scan operation reads every item in a table. So you should avoid using Scan with large tables.

The filter expression is applied to the Scan results, and items that don't match the filter expression are discarded.

If you define an expression attribute value, you must use it consistently throughout the entire expression. Also, you can't omit the : symbol.

Expression attribute values are used with key condition expressions, condition expressions, update expressions, and filter expressions.

Note

For programming language-specific code examples, see [Getting started with DynamoDB and the AWS SDKs.](#)

Condition expressions

To manipulate data in an Amazon DynamoDB table, you use the PutItem, UpdateItem, and DeleteItem operations. (You can also use BatchWriteItem to perform multiple PutItem or DeleteItem operations in a single call.)

For these data manipulation operations, you can specify a *condition expression* to determine which items should be modified. If the condition expression evaluates to true, the operation succeeds; otherwise, the operation fails.

The PutItem, UpdateItem, and DeleteItem operations have a ReturnValues parameter you can use to return attribute values as they appeared before or after you modified them. For more information, see [ReturnValue](#).

The following are some AWS Command Line Interface (AWS CLI) examples of using condition expressions. These examples are based on the ProductCatalog table, which was introduced in [Specifying item attributes when using expressions](#). The partition key for this table is Id; there is no sort key. The following PutItem operation creates a sample ProductCatalog item that the examples refer to.

```
aws dynamodb put-item \
  --table-name ProductCatalog \
  --item file://item.json
```

The arguments for --item are stored in the item.json file. (For simplicity, only a few item attributes are used.)

```
{
  "Id": {"N": "456" },
  "ProductCategory": {"S": "Sporting Goods" },
  "Price": {"N": "650" }
}
```

Topics

- [Conditional put](#)
- [Conditional deletes](#)
- [Conditional updates](#)
- [Conditional expression examples](#)
- [Comparison operator and function reference](#)

Conditional put

The PutItem operation overwrites an item with the same primary key (if it exists). If you want to avoid this, use a condition expression. This allows the write to proceed only if the item in question does not already have the same primary key.

The following example uses `attribute_not_exists()` to check whether the primary key exists in the table before attempting the write operation.

Note

If your primary key consists of both a partition key(pk) and a sort key(sk), the parameter will check whether `attribute_not_exists(pk)` AND `attribute_not_exists(sk)` evaluate to true or false before attempting the write operation.

```
aws dynamodb put-item \  
  --table-name ProductCatalog \  
  --item file://item.json \  
  --condition-expression "attribute_not_exists(Id)"
```

If the condition expression evaluates to false, DynamoDB returns the following error message: The conditional request failed.

Note

For more information about `attribute_not_exists` and other functions, see [Comparison operator and function reference](#).

Conditional deletes

To perform a conditional delete, you use a `DeleteItem` operation with a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

Consider the item from [Condition expressions](#).

```
{  
    "Id": {  
        "N": "456"  
    },  
    "Price": {  
        "N": "650"  
    },  
    "ProductCategory": {  
        "S": "Sporting Goods"  
    }  
}
```

Suppose that you wanted to delete the item, but only under the following conditions:

- The `ProductCategory` is either "Sporting Goods" or "Gardening Supplies."
- The `Price` is between 500 and 600.

The following example tries to delete the item.

```
aws dynamodb delete-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"456"}}' \  
  --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (Price between :lo  
  and :hi)" \  
  --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
    ":cat1": {"S": "Sporting Goods"},  
    ":cat2": {"S": "Gardening Supplies"},  
    ":lo": {"N": "500"},  
    ":hi": {"N": "600"}}
```

```
}
```

Note

In the condition expression, the : (colon character) indicates an *expression attribute value*—a placeholder for an actual value. For more information, see [Expression attribute values](#). For more information about IN, AND, and other keywords, see [Comparison operator and function reference](#).

In this example, the ProductCategory comparison evaluates to true, but the Price comparison evaluates to false. This causes the condition expression to evaluate to false and the DeleteItem operation to fail.

Conditional updates

To perform a conditional update, you use an UpdateItem operation with a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

Note

UpdateItem also supports *update expressions*, where you specify the modifications you want to make to an item. For more information, see [Update expressions](#).

Suppose that you started with the item shown in [Condition expressions](#).

```
{
    "Id": { "N": "456" },
    "Price": { "N": "650" },
    "ProductCategory": { "S": "Sporting Goods" }
}
```

The following example performs an UpdateItem operation. It tries to reduce the Price of a product by 75—but the condition expression prevents the update if the current Price is less than or equal to 500.

```
aws dynamodb update-item \
```

```
--table-name ProductCatalog \
--key '{"Id": {"N": "456"}}' \
--update-expression "SET Price = Price - :discount" \
--condition-expression "Price > :limit" \
--expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
  ":discount": { "N": "75" },
  ":limit": {"N": "500"}
}
```

If the starting Price is 650, the `UpdateItem` operation reduces the Price to 575. If you run the `UpdateItem` operation again, the Price is reduced to 500. If you run it a third time, the condition expression evaluates to false, and the update fails.

Note

In the condition expression, the `:` (colon character) indicates an *expression attribute value*—a placeholder for an actual value. For more information, see [Expression attribute values](#). For more information about `>` and other operators, see [Comparison operator and function reference](#).

Conditional expression examples

For more information about the functions used in the following examples, see [Comparison operator and function reference](#). If you want to know more about how to specify different attribute types in an expression, see [Specifying item attributes when using expressions](#).

Checking for attributes in an item

You can check for the existence (or nonexistence) of any attribute. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `attribute_not_exists` to delete a product only if it does not have a `Price` attribute.

```
aws dynamodb delete-item \
```

```
--table-name ProductCatalog \
--key '{"Id": {"N": "456"}}' \
--condition-expression "attribute_not_exists(Price)"
```

DynamoDB also provides an `attribute_exists` function. The following example deletes a product only if it has received poor reviews.

```
aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{"Id": {"N": "456"}}' \
--condition-expression "attribute_exists(ProductReviews.OneStar)"
```

Checking for attribute type

You can check the data type of an attribute value by using the `attribute_type` function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `attribute_type` to delete a product only if it has a `Color` attribute of type String Set.

```
aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{"Id": {"N": "456"}}' \
--condition-expression "attribute_type(Color, :v_sub)" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{
  ":v_sub": {"S": "SS"}
}
```

Checking string starting value

You can check if a String attribute value begins with a particular substring by using the `begins_with` function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `begins_with` to delete a product only if the `FrontView` element of the `Pictures` map starts with a specific value.

```
aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{"Id": {"N": "456"}}' \
--condition-expression "begins_with(Pictures.FrontView, :v_sub)" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for --expression-attribute-values are stored in the expression-attribute-values.json file.

```
{  
  ":v_sub":{"S":"http://"}  
}
```

Checking for an element in a set

You can check for an element in a set or look for a substring within a string by using the contains function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses contains to delete a product only if the Color String Set has an element with a specific value.

```
aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{"Id": {"N": "456"}}' \
--condition-expression "contains(Color, :v_sub)" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for --expression-attribute-values are stored in the expression-attribute-values.json file.

```
{  
  ":v_sub":{"S":"Red"}  
}
```

Checking the size of an attribute value

You can check for the size of an attribute value by using the size function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses size to delete a product only if the size of the VideoClip Binary attribute is greater than 64000 bytes.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "size(VideoClip) > :v_sub" \
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{
  ":v_sub":{"N":"64000"}
}
```

Comparison operator and function reference

This section covers the built-in functions and keywords for writing filter expressions and condition expressions in Amazon DynamoDB. For more detailed information on functions and programming with DynamoDB, see [Programming with DynamoDB and the AWS SDKs](#) and the [DynamoDB API Reference](#).

Topics

- [Syntax for filter and condition expressions](#)
- [Making comparisons](#)
- [Functions](#)
- [Logical evaluations](#)
- [Parentheses](#)
- [Precedence in conditions](#)

Syntax for filter and condition expressions

In the following syntax summary, an *operand* can be the following:

- A top-level attribute name, such as `Id`, `Title`, `Description`, or `ProductCategory`
- A document path that references a nested attribute

```
condition-expression ::=  
  operand comparator operand
```

```
| operand BETWEEN operand AND operand
| operand IN ( operand (',', operand (, ...)))
| function
| condition AND condition
| condition OR condition
| NOT condition
| ( condition )

comparator ::=
=
|
| <>
| <
| <=
| >
| >=>

function ::=
attribute_exists (path)
| attribute_not_exists (path)
| attribute_type (path, type)
| begins_with (path, substr)
| contains (path, operand)
| size (path)
```

Making comparisons

Use these comparators to compare an operand against a range of values or an enumerated list of values:

- *a* = *b* – True if *a* is equal to *b*.
- *a* <*>* *b* – True if *a* is not equal to *b*.
- *a* < *b* – True if *a* is less than *b*.
- *a* <= *b* – True if *a* is less than or equal to *b*.
- *a* > *b* – True if *a* is greater than *b*.
- *a* >= *b* – True if *a* is greater than or equal to *b*.

Use the BETWEEN and IN keywords to compare an operand against a range of values or an enumerated list of values:

- *a* BETWEEN *b* AND *c* – True if *a* is greater than or equal to *b*, and less than or equal to *c*.

- **a IN (b, c, d)** – True if **a** is equal to any value in the list—for example, any of **b**, **c**, or **d**. The list can contain up to 100 values, separated by commas.

Functions

Use the following functions to determine whether an attribute exists in an item, or to evaluate the value of an attribute. These function names are case sensitive. For a nested attribute, you must provide its full document path.

Function	Description
<code>attribute_exists (path)</code>	<p>True if the item contains the attribute specified by path.</p> <p>Example: Check whether an item in the Product table has a side view picture.</p> <ul style="list-style-type: none">• <code>attribute_exists (#Picture s.#SideView)</code>
<code>attribute_not_exists (path)</code>	<p>True if the attribute specified by path does not exist in the item.</p> <p>Example: Check whether an item has a Manufacturer attribute.</p> <ul style="list-style-type: none">• <code>attribute_not_exists (Manufacturer)</code>
<code>attribute_type (path, type)</code>	<p>True if the attribute at the specified path is of a particular data type. The type parameter must be one of the following:</p> <ul style="list-style-type: none">• S – String• SS – String set•

Function	Description
	<p>N – Number</p> <ul style="list-style-type: none">• NS – Number set• B – Binary• BS – Binary set• BOOL – Boolean• NULL – Null• L – List• M – Map
	<p>You must use an expression attribute value for the type parameter.</p> <p>Example: Check whether the <code>Quantity0nHand</code> attribute is of type List. In this example, <code>:v_sub</code> is a placeholder for the string L.</p> <ul style="list-style-type: none">• <code>attribute_type (ProductReviews.FiveStar, :v_sub)</code>
	<p>You must use an expression attribute value for the type parameter.</p>

Function	Description
<code>begins_with (<i>path</i>, <i>substr</i>)</code>	<p>True if the attribute specified by path begins with a particular substring.</p> <p>Example: Check whether the first few characters of the front view picture URL are <code>http://</code>.</p> <ul style="list-style-type: none">• <code>begins_with (Pictures.FrontView, :v_sub)</code> <p>The expression attribute value <code>:v_sub</code> is a placeholder for <code>http://</code>.</p>

Function	Description
<code>contains (path, operand)</code>	<p>True if the attribute specified by path is one of the following:</p> <ul style="list-style-type: none">• A String that contains a particular substring.• A Set that contains a particular element within the set.• A List that contains a particular element within the list. <p>If the attribute specified by path is a String, the operand must be a String. If the attribute specified by path is a Set, the operand must be the set's element type.</p> <p>The path and the operand must be distinct. That is, <code>contains (a, a)</code> returns an error.</p> <p>Example: Check whether the Brand attribute contains the substring Company.</p> <ul style="list-style-type: none">• <code>contains (Brand, :v_sub)</code> <p>The expression attribute value <code>:v_sub</code> is a placeholder for Company.</p> <p>Example: Check whether the product is available in red.</p> <ul style="list-style-type: none">• <code>contains (Color, :v_sub)</code>

Function	Description
	The expression attribute value :v_sub is a placeholder for Red.

Function	Description
<code>size (path)</code>	<p>Returns a number that represents an attribute's size. The following are valid data types for use with <code>size</code>.</p> <p>If the attribute is of type <code>String</code>, <code>size</code> returns the length of the string.</p> <p>Example: Check whether the string <code>Brand</code> is less than or equal to 20 characters. The expression attribute value <code>:v_sub</code> is a placeholder for 20.</p> <ul style="list-style-type: none">• <code>size (Brand) <= :v_sub</code>
	<p>If the attribute is of type <code>Binary</code>, <code>size</code> returns the number of bytes in the attribute value.</p> <p>Example: Suppose that the <code>ProductCatalog</code> item has a binary attribute named <code>VideoClip</code> that contains a short video of the product in use. The following expression checks whether <code>VideoClip</code> exceeds 64,000 bytes. The expression attribute value <code>:v_sub</code> is a placeholder for 64000.</p> <ul style="list-style-type: none">• <code>size(VideoClip) > :v_sub</code> <p>If the attribute is a <code>Set</code> data type, <code>size</code> returns the number of elements in the set.</p>

Function	Description
	<p>Example: Check whether the product is available in more than one color. The expression attribute value <code>:v_sub</code> is a placeholder for 1.</p> <ul style="list-style-type: none"> <li data-bbox="833 445 1274 508">• <code>size (Color) < :v_sub</code>
	<p>If the attribute is of type List or Map, <code>size</code> returns the number of child elements.</p> <p>Example: Check whether the number of OneStar reviews has exceeded a certain threshold. The expression attribute value <code>:v_sub</code> is a placeholder for 3.</p> <ul style="list-style-type: none"> <li data-bbox="833 994 1400 1100">• <code>size(ProductReviews.OneStar) > :v_sub</code>

Logical evaluations

Use the AND, OR, and NOT keywords to perform logical evaluations. In the following list, *a* and *b* represent conditions to be evaluated.

- *a* AND *b* – True if *a* and *b* are both true.
- *a* OR *b* – True if either *a* or *b* (or both) are true.
- NOT *a* – True if *a* is false. False if *a* is true.

The following is a code example of AND in an operation.

```
dynamodb-local (*)> select * from exprtest where a > 3 and a < 5;
```

Parentheses

Use parentheses to change the precedence of a logical evaluation. For example, suppose that conditions *a* and *b* are true, and that condition *c* is false. The following expression evaluates to true:

- *a* OR *b* AND *c*

However, if you enclose a condition in parentheses, it is evaluated first. For example, the following evaluates to false:

- (*a* OR *b*) AND *c*

 **Note**

You can nest parentheses in an expression. The innermost ones are evaluated first.

The following is a code example with parentheses in a logical evaluation.

```
dynamodb-local (*)> select * from exprtest where attribute_type(b, string)  
or ( a = 5 and c = "coffee");
```

Precedence in conditions

DynamoDB evaluates conditions from left to right using the following precedence rules:

- = <> < <= > >=
- IN
- BETWEEN
- attribute_exists attribute_not_exists begins_with contains
- Parentheses
- NOT
- AND
- OR

Update expressions

The `UpdateItem` operation updates an existing item, or adds a new item to the table if it does not already exist. You must provide the key of the item that you want to update. You must also provide an update expression, indicating the attributes that you want to modify and the values that you want to assign to them.

An *update expression* specifies how `UpdateItem` will modify the attributes of an item—for example, setting a scalar value or removing elements from a list or a map.

The following is a syntax summary for update expressions.

```
update-expression ::=  
  [ SET action [, action] ... ]  
  [ REMOVE action [, action] ... ]  
  [ ADD action [, action] ... ]  
  [ DELETE action [, action] ... ]
```

An update expression consists of one or more clauses. Each clause begins with a `SET`, `REMOVE`, `ADD`, or `DELETE` keyword. You can include any of these clauses in an update expression, in any order. However, each action keyword can appear only once.

Within each clause, there are one or more actions separated by commas. Each action represents a data modification.

The examples in this section are based on the `ProductCatalog` item shown in [Projection expressions](#).

The topics below cover some different use cases for the `SET` action.

Topics

- [SET — modifying or adding item attributes](#)
- [REMOVE — deleting attributes from an item](#)
- [ADD — updating numbers and sets](#)
- [DELETE — removing elements from a set](#)
- [Using multiple update expressions](#)

SET — modifying or adding item attributes

Use the SET action in an update expression to add one or more attributes to an item. If any of these attributes already exists, they are overwritten by the new values.

You can also use SET to add or subtract from an attribute that is of type Number. To perform multiple SET actions, separate them with commas.

In the following syntax summary:

- The *path* element is the document path to the item.
- An *operand* element can be either a document path to an item or a function.

```
set-action ::=  
    path = value  
  
value ::=  
    operand  
    | operand '+' operand  
    | operand '-' operand  
  
operand ::=  
    path | function
```

The following PutItem operation creates a sample item that the examples refer to.

```
aws dynamodb put-item \  
  --table-name ProductCatalog \  
  --item file://item.json
```

The arguments for --item are stored in the `item.json` file. (For simplicity, only a few item attributes are used.)

```
{  
    "Id": {"N": "789"},  
    "ProductCategory": {"S": "Home Improvement"},  
    "Price": {"N": "52"},  
    "InStock": {"BOOL": true},  
    "Brand": {"S": "Acme"}  
}
```

Topics

- [Modifying attributes](#)
- [Adding lists and maps](#)
- [Adding elements to a list](#)
- [Adding nested map attributes](#)
- [Incrementing and decrementing numeric attributes](#)
- [Appending elements to a list](#)
- [Preventing overwrites of an existing attribute](#)

Modifying attributes

Example

Update the ProductCategory and Price attributes.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "SET ProductCategory = :c, Price = :p" \  
  --expression-attribute-values file://values.json \  
  --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
  ":c": { "S": "Hardware" },  
  ":p": { "N": "60" }  
}
```

Note

In the `UpdateItem` operation, `--return-values ALL_NEW` causes DynamoDB to return the item as it appears after the update.

Adding lists and maps

Example

Add a new list and a new map.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "SET RelatedItems = :ri, ProductReviews = :pr" \  
  --expression-attribute-values file://values.json \  
  --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
  ":ri": {  
    "L": [  
      { "S": "Hammer" }  
    ]  
  },  
  ":pr": {  
    "M": {  
      "FiveStar": {  
        "L": [  
          { "S": "Best product ever!" }  
        ]  
      }  
    }  
  }  
}
```

Adding elements to a list

Example

Add a new attribute to the `RelatedItems` list. (Remember that list elements are zero-based, so `[0]` represents the first element in the list, `[1]` represents the second, and so on.)

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "SET RelatedItems[1] = :ri" \  
  --expression-attribute-values file://values.json \  
  --return-values ALL_NEW
```

```
--expression-attribute-values file://values.json \
--return-values ALL_NEW
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{
  ":ri": { "S": "Nails" }
}
```

Note

When you use SET to update a list element, the contents of that element are replaced with the new data that you specify. If the element doesn't already exist, SET appends the new element at the end of the list.

If you add multiple elements in a single SET operation, the elements are sorted in order by element number.

Adding nested map attributes

Example

Add some nested map attributes.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "SET #pr.#5star[1] = :r5, #pr.#3star = :r3" \
--expression-attribute-names file://names.json \
--expression-attribute-values file://values.json \
--return-values ALL_NEW
```

The arguments for --expression-attribute-names are stored in the names.json file.

```
{
  "#pr": "ProductReviews",
  "#5star": "FiveStar",
  "#3star": "ThreeStar"
}
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{  
    ":r5": { "S": "Very happy with my purchase" },  
    ":r3": {  
        "L": [  
            { "S": "Just OK - not that great" }  
        ]  
    }  
}
```

Incrementing and decrementing numeric attributes

You can add to or subtract from an existing numeric attribute. To do this, use the + (plus) and - (minus) operators.

Example

Decrease the Price of an item.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "SET Price = Price - :p" \  
  --expression-attribute-values '{":p": {"N":"15"}}' \  
  --return-values ALL_NEW
```

To increase the Price, you would use the + operator in the update expression.

Appending elements to a list

You can add elements to the end of a list. To do this, use SET with the `list_append` function. (The function name is case sensitive.) The `list_append` function is specific to the SET action and can only be used in an update expression. The syntax is as follows.

- `list_append (list1, list2)`

The function takes two lists as input and appends all elements from `list2` to `list1`.

Example

In [Adding elements to a list](#), you create the `RelatedItems` list and populate it with two elements: Hammer and Nails. Now you append two more elements to the end of `RelatedItems`.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "SET #ri = list_append(#ri, :vals)" \
--expression-attribute-names '{"#ri": "RelatedItems"}' \
--expression-attribute-values file://values.json \
--return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
  ":vals": {
    "L": [
      { "S": "Screwdriver" },
      {"S": "Hacksaw" }
    ]
  }
}
```

Finally, you append one more element to the *beginning* of `RelatedItems`. To do this, swap the order of the `list_append` elements. (Remember that `list_append` takes two lists as input and appends the second list to the first.)

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "SET #ri = list_append(:vals, #ri)" \
--expression-attribute-names '{"#ri": "RelatedItems"}' \
--expression-attribute-values '{":vals": {"L": [ { "S": "Chisel" } ]}}' \
--return-values ALL_NEW
```

The resulting `RelatedItems` attribute now contains five elements, in the following order: Chisel, Hammer, Nails, Screwdriver, Hacksaw.

Preventing overwrites of an existing attribute

If you want to avoid overwriting an existing attribute, you can use `SET` with the `if_not_exists` function. (The function name is case sensitive.) The `if_not_exists` function is specific to the `SET` action and can only be used in an update expression. The syntax is as follows.

- `if_not_exists (path, value)`

If the item does not contain an attribute at the specified *path*, `if_not_exists` evaluates to *value*; otherwise, it evaluates to *path*.

Example

Set the Price of an item, but only if the item does not already have a Price attribute. (If Price already exists, nothing happens.)

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "SET Price = if_not_exists(Price, :p)" \  
  --expression-attribute-values '{":p": {"N": "100"}}' \  
  --return-values ALL_NEW
```

REMOVE — deleting attributes from an item

Use the REMOVE action in an update expression to remove one or more attributes from an item in Amazon DynamoDB. To perform multiple REMOVE actions, separate them with commas.

The following is a syntax summary for REMOVE in an update expression. The only operand is the document path for the attribute that you want to remove.

```
remove-action ::=  
  path
```

Example

Remove some attributes from an item. (If the attributes don't exist, nothing happens.)

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "REMOVE Brand, InStock, QuantityOnHand" \  
  --return-values ALL_NEW
```

Removing elements from a list

You can use REMOVE to delete individual elements from a list.

Example

In [Appending elements to a list](#), you modify a list attribute (`RelatedItems`) so that it contained five elements:

- [0]—Chisel
- [1]—Hammer
- [2]—Nails
- [3]—Screwdriver
- [4]—Hacksaw

The following AWS Command Line Interface (AWS CLI) example deletes Hammer and Nails from the list.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "REMOVE RelatedItems[1], RelatedItems[2]" \  
  --return-values ALL_NEW
```

After Hammer and Nails are removed, the remaining elements are shifted. The list now contains the following:

- [0]—Chisel
- [1]—Screwdriver
- [2]—Hacksaw

ADD — updating numbers and sets

Note

In general, we recommend using SET rather than ADD.

Use the ADD action in an update expression to add a new attribute and its values to an item.

If the attribute already exists, the behavior of ADD depends on the attribute's data type:

- If the attribute is a number, and the value you are adding is also a number, the value is mathematically added to the existing attribute. (If the value is a negative number, it is subtracted from the existing attribute.)
- If the attribute is a set, and the value you are adding is also a set, the value is appended to the existing set.

 **Note**

The ADD action supports only number and set data types.

To perform multiple ADD actions, separate them with commas.

In the following syntax summary:

- The *path* element is the document path to an attribute. The attribute must be either a Number or a set data type.
- The *value* element is a number that you want to add to the attribute (for Number data types), or a set to append to the attribute (for set types).

```
add-action ::=  
    path value
```

The topics below cover some different use cases for the ADD action.

Topics

- [Adding a number](#)
- [Adding elements to a set](#)

Adding a number

Assume that the QuantityOnHand attribute does not exist. The following AWS CLI example sets QuantityOnHand to 5.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --attribute-value-list '{  
        "QuantityOnHand": {"N": "5"}  
    }'
```

```
--update-expression "ADD QuantityOnHand :q" \
--expression-attribute-values '{":q": {"N": "5"}}' \
--return-values ALL_NEW
```

Now that `QuantityOnHand` exists, you can rerun the example to increment `QuantityOnHand` by 5 each time.

Adding elements to a set

Assume that the `Color` attribute does not exist. The following AWS CLI example sets `Color` to a string set with two elements.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "ADD Color :c" \
--expression-attribute-values '{":c": {"SS":["Orange", "Purple"]}}' \
--return-values ALL_NEW
```

Now that `Color` exists, you can add more elements to it.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "ADD Color :c" \
--expression-attribute-values '{":c": {"SS":["Yellow", "Green", "Blue"]}}' \
--return-values ALL_NEW
```

DELETE — removing elements from a set

Important

The `DELETE` action supports only Set data types.

Use the `DELETE` action in an update expression to remove one or more elements from a set. To perform multiple `DELETE` actions, separate them with commas.

In the following syntax summary:

- The `path` element is the document path to an attribute. The attribute must be a set data type.

- The **subset** is one or more elements that you want to delete from **path**. You must specify **subset** as a set type.

```
delete-action ::=  
    path subset
```

Example

In [Adding elements to a set](#), you create the Color string set. This example removes some of the elements from that set.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "DELETE Color :p" \  
    --expression-attribute-values '{":p": {"SS": ["Yellow", "Purple"]}}' \  
    --return-values ALL_NEW
```

Using multiple update expressions

You can use multiple update expressions in a single statement.

Example

If you want to modify an attribute's value and completely remove another attribute, you could use a SET and a REMOVE action in a single statement. This operation would reduce the Price value to 15 while also removing the InStock attribute from the item.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "SET Price = Price - :p REMOVE InStock" \  
    --expression-attribute-values '{":p": {"N":"15"}}' \  
    --return-values ALL_NEW
```

Example

If you want to add to a list while also changing another attribute's value, you could use two SET actions in a single statement. This operation would add "Nails" to the RelatedItems list attribute and also set the Price value to 21.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems[1] = :newValue, Price = :newPrice" \
    --expression-attribute-values '{":newValue": {"S":"Nails"}, ":newPrice": \
    {"N":"21"} }' \
    --return-values ALL_NEW
```

Time to Live (TTL)

Time To Live (TTL) for DynamoDB is a cost-effective method for deleting items that are no longer relevant. TTL allows you to define a per-item expiration timestamp that indicates when an item is no longer needed. DynamoDB automatically deletes expired items within a few days of their expiration time, without consuming write throughput.

To use TTL, first enable it on a table and then define a specific attribute to store the TTL expiration timestamp. The timestamp must be stored in [Unix epoch time format](#) at the seconds granularity. Each time an item is created or updated, you can compute the expiration time and save it in the TTL attribute.

Items with valid, expired TTL attributes may be deleted by the system at any time, typically within a few days of their expiration. You can still update the expired items that are pending deletion, including changing or removing their TTL attributes. While updating an expired item, we recommended that you use a condition expression to make sure the item has not been subsequently deleted. Use filter expressions to remove expired items from [Scan](#) and [Query](#) results.

Deleted items work similarly to those deleted through typical delete operations. Once deleted, items go into DynamoDB Streams as service deletions instead of user deletes, and are removed from local secondary indexes and global secondary indexes just like other delete operations.

If you are using [Global Tables version 2019.11.21 \(Current\)](#) of global tables and you also use the TTL feature, DynamoDB replicates TTL deletes to all replica tables. The initial TTL delete does not consume Write Capacity Units (WCUs) in the region in which the TTL expiry occurs. However, the replicated TTL delete to the replica table(s) consumes a replicated Write Capacity Unit when using provisioned capacity, or Replicated Write Unit when using on-demand capacity mode, in each of the replica regions and applicable charges will apply.

For more information about TTL, see these topics:

Topics

- [Enabling time to live \(TTL\)](#)
- [Computing time to live \(TTL\)](#)
- [Working with expired items](#)

Enabling time to live (TTL)

TTL can be enabled in the Amazon DynamoDB Console, the AWS Command Line Interface (AWS CLI), or using the [Amazon DynamoDB API Reference](#) with any of the supposed AWS SDKs. It takes approximately one hour for TTL to be enabled across all partitions.

Enable DynamoDB TTL using the AWS console

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Tables**, and then choose the table that you want to modify.
3. In the **Additional settings** tab, in the **Time to Live (TTL)** section, choose **Turn on** to enable TTL.

The screenshot shows the AWS Management Console interface for Amazon DynamoDB. The top navigation bar includes links for Overview, Indexes, Monitor, Global tables, Backups, Exports and streams, and Additional settings. The Additional settings tab is currently selected. Below the tabs, there are several sections: **Read/write capacity** (with an Edit button), **Auto scaling activities (0)** (with a Find events search bar and a C refresh button), and **Time to Live (TTL)**. The **Time to Live (TTL)** section contains a sub-section titled "Automatically delete expired items from a table." It features a "Run preview" button and a "Turn on" button, which is highlighted with a red rectangle. Other options in this section include "TTL status" (set to Off) and "Capacity unit" (set to 1). A note at the bottom of the TTL section states: "There are no auto scaling activities for the table or its global secondary indexes."

4. When enabling TTL on a table, DynamoDB requires you to identify a specific attribute name that the service will look for when determining if an item is eligible for expiration. The TTL attribute name, shown below, is case sensitive and must match the attribute defined in your read and write operations. A mismatch will cause expired items to go undeleted. Renaming the TTL attribute requires you to disable TTL and then reenable it with the new attribute going forward. TTL will continue to process deletions for approximately 30 minutes once it is disabled. TTL must be reconfigured on restored tables.

The screenshot shows the 'Turn on Time to Live (TTL)' configuration page. At the top, the navigation path is: DynamoDB > Tables > Music > Turn on Time to Live (TTL). The main section is titled 'TTL settings' and contains a 'TTL attribute name' field where 'expireAt' is entered. Below this, a note says 'Between 1 and 255 characters.' In the 'Preview' section, there's a 'Simulated date and time' field with an epoch time value of 1694640532, showing a timestamp of September 13, 2023, 15:28:52 (UTC-06:00), and a 'Run preview' button. A note at the bottom left states: 'Activating TTL can take up to one hour to be applied across all partitions. You will not be able to make additional TTL changes until this update is complete.' At the bottom right are 'Cancel' and 'Turn on TTL' buttons.

5. (Optional) You can perform a test by simulating the date and time of the expiration and matching a few items. This provides you with a sample list of items and confirms that there are items containing the TTL attribute name provided along with the expiration time.

Turn on Time to Live (TTL) Info

TTL settings

TTL attribute name

The name of the attribute that will be stored in the TTL timestamp.

Between 1 and 255 characters.

Preview

Confirm that your TTL attribute and values are working properly by specifying a date and time, and reviewing a sample of the items that will be deleted by then. Note that preview may show only some of the relevant items.

Simulated date and time

Specify the date and time to simulate which items would be expired.

December 11, 2023, 16:58:01 (UTC-07:00)

Items to be deleted (32)

artist	album	createdAt	expireAt (TTL)
f91897e5-0...	72499653-...	1694559481	1702339081
7d38838f-e...	64b6999b-...	1694559479	1702339079
6734d779-...	52d667bd-...	1694559481	1702339081
4553fb30-...	bb2cc547-e...	1694559481	1702339081
ea7c0eeb-5...	840b3c7b-...	1694559478	1702339078

After TTL is enabled, the TTL attribute is marked **TTL** when you view items on the DynamoDB console. You can view the date and time that an item expires by hovering your pointer over the attribute.

Items returned (100)

	artist (String)	album (String)	createdAt	expireAt (TTL)	
<input type="checkbox"/>	f91897e5-0a7e-4ee8-a9be-561ec...	72499653-50fd-454f-9ed0-496...	1694559481	1702339081	
<input type="checkbox"/>	7d38838f-e904-4673-96ba-ab29c...	64b6999b-80aa-46d6-b567-c6f...	1694559479	1702339079	
<input type="checkbox"/>	9da8f8a1-d920-41e2-8469-88fa8...	e8cb4ef3-8d22-4f5b-96f3-e79c...	1694559481	UTC December 11, 2023 23:58:06 UTC	X
<input type="checkbox"/>	6734d779-5d71-47f3-ae4a-4b617...	52d667bd-cd9d-48a4-9a66-3bf...	1694559481	Local December 11, 2023 16:58:06 MST	
<input type="checkbox"/>	cdb74466-0b36-41cd-9b39-cbe41...	52965e04-cb1a-4089-b891-9a1...	1694559481	Region (N. Virginia)	
<input type="checkbox"/>	70aba065-a9d3-40f3-bd64-0d34c...	3272c168-4de2-4edf-a253-e02...	1694559481	December 11, 2023 18:58:06 EST	
<input type="checkbox"/>	54caf925-843f-4966-b1e3-95530...	5e723d06-877d-4572-808b-e8d...	1694559481	1702339086	
<input type="checkbox"/>	4af50ef7-8c8e-4cc3-ad61-9eb3b5...	8c3dfc04-7091-4557-b287-67ca...	1694559486	1702339087	

Enable DynamoDB TTL using the API

Python

You can enable TTL with code, using the [UpdateTimeToLive](#) operation.

```
import boto3

def enable_ttl(table_name, ttl_attribute_name):
    """
    Enables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table
    :param ttl_attribute_name: The name of the TTL attribute being provided to the
        table.
    """
    try:
        dynamodb = boto3.client('dynamodb')

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={
                'Enabled': True,
                'AttributeName': ttl_attribute_name
            }
        )
    
```

```
# In the returned response, check for a successful status code.  
if response['ResponseMetadata']['HTTPStatusCode'] == 200:  
    print("TTL has been enabled successfully.")  
else:  
    print(f"Failed to enable TTL, status code {response['ResponseMetadata']['HTTPStatusCode']}")  
except Exception as ex:  
    print("Couldn't enable TTL in table %s. Here's why: %s" % (table_name, ex))  
    raise  
  
# your values  
enable_ttl('your-table-name', 'expirationDate')
```

You can confirm TTL is enabled by using the [DescribeTimeToLive](#) operation, which describes the TTL status on a table. The TimeToLive status is either ENABLED or DISABLED.

```
# create a DynamoDB client  
dynamodb = boto3.client('dynamodb')  
  
# set the table name  
table_name = 'YourTable'  
  
# describe TTL  
response = dynamodb.describe_time_to_live(TableName=table_name)
```

Javascript

You can enable TTL with code, using the [UpdateTimeToLiveCommand](#) operation.

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-dynamodb";  
  
const enableTTL = async (tableName, ttlAttribute) => {  
  
    const client = new DynamoDBClient({});  
  
    const params = {  
        TableName: tableName,  
        TimeToLiveSpecification: {  
            Enabled: true,  
            AttributeName: ttlAttribute  
        }  
    };
```

```
try {
    const response = await client.send(new UpdateTimeToLiveCommand(params));
    if (response.$metadata.httpStatusCode === 200) {
        console.log(`TTL enabled successfully for table ${tableName}, using
attribute name ${ttlAttribute}.`);
    } else {
        console.log(`Failed to enable TTL for table ${tableName}, response
object: ${response}`);
    }
    return response;
} catch (e) {
    console.error(`Error enabling TTL: ${e}`);
    throw e;
}
};

// call with your own values
enableTTL('ExampleTable', 'exampleTtlAttribute');
```

Enable Time to Live using the AWS CLI

1. Enable TTL on the TTLExample table.

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-
specification "Enabled=true, AttributeName=ttl"
```

2. Describe TTL on the TTLExample table.

```
aws dynamodb describe-time-to-live --table-name TTLExample
{
    "TimeToLiveDescription": {
        "AttributeName": "ttl",
        "TimeToLiveStatus": "ENABLED"
    }
}
```

3. Add an item to the TTLExample table with the Time to Live attribute set using the BASH shell and the AWS CLI.

```
EXP=`date -d '+5 days' +%s`
```

```
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl": {"N": "'$EXP'"}}'
```

This example starts with the current date and adds 5 days to it to create an expiration time. Then, it converts the expiration time to epoch time format to finally add an item to the "TTLExample" table.

 **Note**

One way to set expiration values for Time to Live is to calculate the number of seconds to add to the expiration time. For example, 5 days is 432,000 seconds. However, it is often preferable to start with a date and work from there.

It is fairly simple to get the current time in epoch time format, as in the following examples.

- Linux Terminal: `date +%s`
- Python: `import time; int(time.time())`
- Java: `System.currentTimeMillis() / 1000L`
- JavaScript: `Math.floor(Date.now() / 1000)`

Enable DynamoDB TTL using AWS CloudFormation

1. Enable TTL on the TTLExample table.

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-specification "Enabled=true, AttributeName=ttl"
```

2. Describe TTL on the TTLExample table.

```
aws dynamodb describe-time-to-live --table-name TTLExample
{
    "TimeToLiveDescription": {
        "AttributeName": "ttl",
        "TimeToLiveStatus": "ENABLED"
    }
}
```

3. Add an item to the TTLExample table with the Time to Live attribute set using the BASH shell and the AWS CLI.

```
EXP=`date -d '+5 days' +%s`  
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl": {"N": "'$EXP'"}}'
```

This example starts with the current date and adds 5 days to it to create an expiration time. Then, it converts the expiration time to epoch time format to finally add an item to the "TTLExample" table.

 **Note**

One way to set expiration values for Time to Live is to calculate the number of seconds to add to the expiration time. For example, 5 days is 432,000 seconds. However, it is often preferable to start with a date and work from there.

It is fairly simple to get the current time in epoch time format, as in the following examples.

- Linux Terminal: `date +%s`
- Python: `import time; int(time.time())`
- Java: `System.currentTimeMillis() / 1000L`
- JavaScript: `Math.floor(Date.now() / 1000)`

Computing time to live (TTL)

A common way to implement TTL is to set an expiration time for items based on when they were created or last updated. This can be done by adding time to the `createdAt` and `updatedAt` timestamps. For example, the TTL for newly created items can be set to `createdAt + 90 days`. When the item is updated the TTL can be recalculated to `updatedAt + 90 days`.

The computed expiration time must be in epoch format, in seconds. To be considered for expiry and deletion, the TTL cannot be more than five years in the past. If you use any other format, the TTL processes ignore the item. If you set the expiration date to sometime in the future when you want the item to expire, the item will be expired after that time. For example, say that you set the

expiration date to 1724241326 (which is Monday, August 21st, 2024 11:55:26 (GMT)). The item will be expired after the specified time.

Topics

- [Create an item and set the Time to Live](#)
- [Update an item and refresh the Time to Live](#)

Create an item and set the Time to Live

The following example demonstrates how to calculate the expiration time when creating a new item, using `expireAt` as the TTL attribute name. An assignment statement obtains the current time as a variable. In the example, the expiration time is calculated as 90 days from the current time. The time is then converted to epoch format and saved as an integer data type in the TTL attribute.

Python

```
import boto3
from datetime import datetime, timedelta

def create_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Creates a DynamoDB item with an attached expiry attribute.

    :param table_name: Table name for the boto3 resource to target when creating an item
    :param region: string representing the AWS region. Example: `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """

    try:
        dynamodb = boto3.resource('dynamodb', region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expiration time (90 days from now) in epoch second format
        expiration_time = int((datetime.now() + timedelta(days=90)).timestamp())
    
```

```
item = {
    'primaryKey': primary_key,
    'sortKey': sort_key,
    'createdAt': current_time,
    'expiredAt': expiration_time
}

table.put_item(Item=item)

print("Item created successfully.")
except Exception as e:
    print(f"Error creating item: {e}")
    raise

# Use your own values
create_dynamodb_item('your-table-name', 'us-west-2', 'your-partition-key-value',
    'your-sort-key-value')
```

Javascript

In this request we add logic to compute the expiration time of the newly created item:

```
import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";

function createDynamoDBItem(table_name, region, partition_key, sort_key) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    // Get the current time in epoch second format
    const current_time = Math.floor(new Date().getTime() / 1000);

    // Calculate the expireAt time (90 days from now) in epoch second format
    const expire_at = Math.floor((new Date().getTime() + 90 * 24 * 60 * 60 * 1000) / 1000);

    // Create DynamoDB item
    const item = {
        'partitionKey': {'S': partition_key},
        'sortKey': {'S': sort_key},
        'createdAt': {'N': current_time.toString()},
        'expiredAt': {'N': (current_time + 90 * 24 * 60 * 60 * 1000).toString()}
    };

    const command = new PutItemCommand({
        TableName: table_name,
        Item: item
    });

    client.send(command);
}
```

```
'expireAt': {'N': expire_at.toString()}

};

const putItemCommand = new PutItemCommand({
    TableName: table_name,
    Item: item,
    ProvisionedThroughput: {
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1,
    },
});

client.send(putItemCommand, function(err, data) {
    if (err) {
        console.log("Exception encountered when creating item %s, here's what
happened: ", data, ex);
        throw err;
    } else {
        console.log("Item created successfully: %s.", data);
        return data;
    }
});
});

// use your own values
createDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
'your-sort-key-value');
```

Update an item and refresh the Time to Live

This example is a continuation of the one from the [previous section](#). The expiration time can be recomputed if the item is updated. The following example recomputes the `expireAt` timestamp to be 90 days from the current time.

Python

```
import boto3
from datetime import datetime, timedelta

def update_dynamodb_item(table_name, region, primary_key, sort_key):
    """
```

```
Update an existing DynamoDB item with a TTL.  
:param table_name: Name of the DynamoDB table  
:param region: AWS Region of the table - example `us-east-1`  
:param primary_key: one attribute known as the partition key.  
:param sort_key: Also known as a range attribute.  
:return: Void (nothing)  
"""  
  
try:  
    # Create the DynamoDB resource.  
    dynamodb = boto3.resource('dynamodb', region_name=region)  
    table = dynamodb.Table(table_name)  
  
    # Get the current time in epoch second format  
    current_time = int(datetime.now().timestamp())  
  
    # Calculate the expireAt time (90 days from now) in epoch second format  
    expire_at = int((datetime.now() + timedelta(days=90)).timestamp())  
  
    table.update_item(  
        Key={  
            'partitionKey': primary_key,  
            'sortKey': sort_key  
        },  
        UpdateExpression="set updatedAt=:c, expireAt=:e",  
        ExpressionAttributeValues={  
            ':c': current_time,  
            ':e': expire_at  
        },  
    )  
  
    print("Item updated successfully.")  
except Exception as e:  
    print(f"Error updating item: {e}")  
  
# Replace with your own values  
update_dynamodb_item('your-table-name', 'us-west-2', 'your-partition-key-value',  
    'your-sort-key-value')
```

The output from the update operation shows that, while the `createdAt` time is unchanged, the `updatedAt` and `expireAt` times have been updated. The `expireAt` time is now set to 90 days from the time of the last update, which is Thursday, October 19, 2023 at 1:27:15 PM.

partition_key	createdAt	updatedAt	expireAt	attribute_1	attribute_2
some_value	2023-07-1 7 14:11:05. 322323	2023-07-1 9 13:27:15. 213423	1697722035	new_value	some_value

Javascript

```

import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

async function updateDynamoDBItem(tableName, region, partitionKey, sortKey) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);
    const expireAt = Math.floor((Date.now() + 90 * 24 * 60 * 60 * 1000) /
1000); //is there a better way to do this?

    const params = {
        TableName: tableName,
        Key: marshall({
            partitionKey: partitionKey,
            sortKey: sortKey
        }),
        UpdateExpression: "SET updatedAt = :c, expireAt = :e",
        ExpressionAttributeValues: marshall({
            ":c": currentTime,
            ":e": expireAt
        }),
    };
}

try {
    const data = await client.send(new UpdateItemCommand(params));
    const responseData = unmarshall(data.Attributes);
    console.log("Item updated successfully: %s", responseData);
    return responseData;
} catch (err) {
}

```

```
        console.error("Error updating item:", err);
        throw err;
    }

//enter your values here
updateDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
    'your-sort-key-value');
```

The TTL examples discussed in this introduction demonstrate a method to ensure only recently updated items are kept in a table. Updated items have their lifespan extended, whereas items not updated post-creation expire and are deleted at no cost, reducing storage and maintaining clean tables.

Working with expired items

Expired items that are pending deletion can be filtered from read and write operations. This is useful in scenarios when expired data is no longer valid and should not be used. If they are not filtered, they'll continue to show in read and write operations until they are deleted by the background process.

Note

These items still count towards storage and read costs until they are deleted.

TTL deletions can be identified in DynamoDB Streams, but only in the Region where the deletion occurred. TTL deletions that are replicated to global table regions are not identifiable in DynamoDB streams in the regions the deletion is replicated to.

Filter expired items from read operations

For read operations such as [Scan](#) and [Query](#), a filter expression can filter out expired items that are pending deletion. As shown in the code snippet below, the filter expression can filter out items where the TTL time is equal to or less than the current time. This is done with an assignment statement that obtains the current time as a variable (now), which is converted to int for epoch time format.

Python

```
import boto3
from datetime import datetime

def query_dynamodb_items(table_name, partition_key):
    """
    :param table_name: Name of the DynamoDB table
    :param partition_key:
    :return:
    """
    try:
        # Initialize a DynamoDB resource
        dynamodb = boto3.resource('dynamodb',
                                  region_name='us-east-1')

        # Specify your table
        table = dynamodb.Table(table_name)

        # Get the current time in epoch format
        current_time = int(datetime.now().timestamp())

        # Perform the query operation with a filter expression to exclude expired
        items
        # response = table.query(
        #
        KeyConditionExpression=boto3.dynamodb.conditions.Key('partitionKey').eq(partition_key),
        #
        FilterExpression=boto3.dynamodb.conditions.Attr('expireAt').gt(current_time)
        #
        response = table.query(
            KeyConditionExpression=dynamodb.conditions.Key('partitionKey').eq(partition_key),
            FilterExpression=dynamodb.conditions.Attr('expireAt').gt(current_time)
        )

        # Print the items that are not expired
        for item in response['Items']:
            print(item)

    except Exception as e:
        print(f"Error querying items: {e}")
```

```
# Call the function with your values
query_dynamodb_items('Music', 'your-partition-key-value')
```

The output from the update operation shows that, while the `createdAt` time is unchanged, the `updatedAt` and `expireAt` times have been updated. The `expireAt` time is now set to 90 days from the time of the last update, which is Thursday, October 19, 2023 at 1:27:15 PM.

partition_key	createdAt	updatedAt	expireAt	attribute_1	attribute_2
some_value	2023-07-1 7 14:11:05. 322323	2023-07-1 9 13:27:15. 213423	1697722035	new_value	some_value

Javascript

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

async function queryDynamoDBItems(tableName, region, primaryKey) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pk",
        FilterExpression: "#ea > :ea",
        ExpressionAttributeNames: {
            "#pk": "primaryKey",
            "#ea": "expireAt"
        },
        ExpressionAttributeValues: marshall({
            ":pk": primaryKey,
            ":ea": currentTime
        })
    };

    const result = await client.send(new QueryCommand(params));
    return result.Items;
}
```

```
        })
    };

    try {
        const { Items } = await client.send(new QueryCommand(params));
        Items.forEach(item => {
            console.log(unmarshall(item))
        });
        return Items;
    } catch (err) {
        console.error(`Error querying items: ${err}`);
        throw err;
    }
}

//enter your own values here
queryDynamoDBItems('your-table-name', 'your-partition-key-value');
```

Conditionally write to expired items

A condition expression can be used to avoid writes against expired items. The code snippet below is a conditional update that checks whether the expiration time is greater than the current time. If true, the write operation will continue.

Python

```
import boto3
from datetime import datetime, timedelta
from botocore.exceptions import ClientError


def update_dynamodb_item(table_name, region, primary_key, sort_key, ttl_attribute):
    """
    Updates an existing record in a DynamoDB table with a new or updated TTL
    attribute.

    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :param ttl_attribute: name of the TTL attribute in the target DynamoDB table
    :return:
    """

    dynamodb = boto3.client('dynamodb', region_name=region)
```

```
"""
try:
    dynamodb = boto3.resource('dynamodb', region_name=region)
    table = dynamodb.Table(table_name)

    # Generate updated TTL in epoch second format
    updated_expiration_time = int((datetime.now() +
timedelta(days=90)).timestamp())

    # Define the update expression for adding/updating a new attribute
    update_expression = "SET newAttribute = :val1"

    # Define the condition expression for checking if 'ttlExpirationDate' is not
    # expired
    condition_expression = "ttlExpirationDate > :val2"

    # Define the expression attribute values
    expression_attribute_values = {
        ':val1': ttl_attribute,
        ':val2': updated_expiration_time
    }

    response = table.update_item(
        Key={
            'primaryKey': primary_key,
            'sortKey': sort_key
        },
        UpdateExpression=update_expression,
        ConditionExpression=condition_expression,
        ExpressionAttributeValues=expression_attribute_values
    )

    print("Item updated successfully.")
    return response['ResponseMetadata']['HTTPStatusCode'] # Ideally a 200 OK
except ClientError as e:
    if e.response['Error']['Code'] == "ConditionalCheckFailedException":
        print("Condition check failed: Item's 'ttlExpirationDate' is expired.")
    else:
        print(f"Error updating item: {e}")
except Exception as e:
    print(f"Error updating item: {e}")

# replace with your values
```

```
update_dynamodb_item('your-table-name', 'us-east-1', 'your-partition-key-value',
    'your-sort-key-value',
        'your-ttl-attribute-value')
```

The output from the update operation shows that, while the `createdAt` time is unchanged, the `updatedAt` and `expireAt` times have been updated. The `expireAt` time is now set to 90 days from the time of the last update, which is Thursday, October 19, 2023 at 1:27:15 PM.

partition_key	createdAt	updatedAt	expireAt	attribute_1	attribute_2
some_value	2023-07-1 7 14:11:05. 322323	2023-07-1 9 13:27:15. 213423	1697722035	new_value	some_value

Javascript

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

// Example function to update an item in a DynamoDB table.
// The function should take the table name, region, partition key, sort key, and
// new attribute as arguments.
// The function should use the DynamoDB client to update the item.
// The function should return the updated item.
// The function should handle errors and exceptions.
const updateDynamoDBItem = async (tableName, region, partitionKey, sortKey,
    newAttribute) => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            artist: partitionKey,
```

```
        album: sortKey
    },
    UpdateExpression: "SET newAttribute = :newAttribute",
    ConditionExpression: "expireAt > :expiration",
    ExpressionAttributeValues: marshall({
        ':newAttribute': newAttribute,
        ':expiration': currentTime
    }),
    ReturnValues: "ALL_NEW"
};

try {
    const response = await client.send(new UpdateItemCommand(params));
    const responseData = unmarshall(response.Attributes);
    console.log("Item updated successfully: ", responseData);
    return responseData;
} catch (error) {
    if (error.name === "ConditionalCheckFailedException") {
        console.log("Condition check failed: Item's 'expireAt' is expired.");
    } else {
        console.error("Error updating item: ", error);
    }
    throw error;
}
};

// Enter your values here
updateDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value', 'your-
sort-key-value', 'your-new-attribute-value');
```

Identifying deleted items in DynamoDB Streams

The streams record contains a user identity field `Records[<index>].userIdentity`. Items that are deleted by the TTL process have the following fields:

```
Records[<index>].userIdentity.type
"Service"
```

```
Records[<index>].userIdentity.principalId
"dynamodb.amazonaws.com"
```

The following JSON shows the relevant portion of a single streams record:

```
"Records": [  
    {  
        ...  
        "userIdentity": {  
            "type": "Service",  
            "principalId": "dynamodb.amazonaws.com"  
        }  
        ...  
    }  
]
```

Working with items: Java

You can use the AWS SDK for Java Document API to perform typical create, read, update, and delete (CRUD) operations on Amazon DynamoDB items in a table.

Note

The SDK for Java also provides an object persistence model, allowing you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

This section contains Java examples to perform several Java Document API item actions and several complete working examples.

Topics

- [Putting an item](#)
- [Getting an item](#)
- [Batch write: Putting and deleting multiple items](#)
- [Batch get: Getting multiple items](#)
- [Updating an item](#)
- [Deleting an item](#)
- [Example: CRUD operations using the AWS SDK for Java document API](#)
- [Example: Batch operations using AWS SDK for Java document API](#)
- [Example: Handling binary type attributes using the AWS SDK for Java document API](#)

Putting an item

The `putItem` method stores an item in a table. If the item exists, it replaces the entire item. Instead of replacing the entire item, if you want to update only specific attributes, you can use the `updateItem` method. For more information, see [Updating an item](#).

Follow these steps:

1. Create an instance of the DynamoDB class.
2. Create an instance of the Table class to represent the table you want to work with.
3. Create an instance of the Item class to represent the new item. You must specify the new item's primary key and its attributes.
4. Call the `putItem` method of the Table object, using the Item that you created in the preceding step.

The following Java code example demonstrates the preceding tasks. The code writes a new item to the `ProductCatalog` table.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

// Build a list of related items
List<Number> relatedItems = new ArrayList<Number>();
relatedItems.add(341);
relatedItems.add(472);
relatedItems.add(649);

//Build a map of product pictures
Map<String, String> pictures = new HashMap<String, String>();
pictures.put("FrontView", "http://example.com/products/123_front.jpg");
pictures.put("RearView", "http://example.com/products/123_rear.jpg");
pictures.put("SideView", "http://example.com/products/123_left_side.jpg");

//Build a map of product reviews
Map<String, List<String>> reviews = new HashMap<String, List<String>>();
```

```
List<String> fiveStarReviews = new ArrayList<String>();
fiveStarReviews.add("Excellent! Can't recommend it highly enough! Buy it!");
fiveStarReviews.add("Do yourself a favor and buy this");
reviews.put("FiveStar", fiveStarReviews);

List<String> oneStarReviews = new ArrayList<String>();
oneStarReviews.add("Terrible product! Do not buy this.");
reviews.put("OneStar", oneStarReviews);

// Build the item
Item item = new Item()
    .withPrimaryKey("Id", 123)
    .withString("Title", "Bicycle 123")
    .withString("Description", "123 description")
    .withString("BicycleType", "Hybrid")
    .withString("Brand", "Brand-Company C")
    .withNumber("Price", 500)
    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red", "Black")))
    .withString("ProductCategory", "Bicycle")
    .withBoolean("InStock", true)
    .withNull("QuantityOnHand")
    .withList("RelatedItems", relatedItems)
    .withMap("Pictures", pictures)
    .withMap("Reviews", reviews);

// Write the item to the table
PutItemOutcome outcome = table.putItem(item);
```

In the preceding example, the item has attributes that are scalars (String, Number, Boolean, Null), sets (String Set), and document types (List, Map).

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters to the `putItem` method. For example, the following Java code example uses an optional parameter to specify a condition for uploading the item. If the condition you specify is not met, the AWS SDK for Java throws a `ConditionalCheckFailedException`. The code example specifies the following optional parameters in the `putItem` method:

- A `ConditionExpression` that defines the conditions for the request. The code defines the condition that the existing item with the same primary key is replaced only if it has an ISBN attribute that equals a specific value.

- A map for ExpressionAttributeValues that is used in the condition. In this case, there is only one substitution required: The placeholder :val in the condition expression is replaced at runtime with the actual ISBN value to be checked.

The following example adds a new book item using these optional parameters.

Example

```
Item item = new Item()
    .withPrimaryKey("Id", 104)
    .withString("Title", "Book 104 Title")
    .withString("ISBN", "444-4444444444")
    .withNumber("Price", 20)
    .withStringSet("Authors",
        new HashSet<String>(Arrays.asList("Author1", "Author2")));
    
Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", "444-4444444444");

PutItemOutcome outcome = table.putItem(
    item,
    "ISBN = :val", // ConditionExpression parameter
    null,           // ExpressionAttributeNames parameter - we're not using it for this
example
    expressionAttributeValues);
```

PutItem and JSON documents

You can store a JSON document as an attribute in a DynamoDB table. To do this, use the withJSON method of Item. This method parses the JSON document and maps each element to a native DynamoDB data type.

Suppose that you wanted to store the following JSON document, containing vendors that can fulfill orders for a particular product.

Example

```
{
    "V01": {
        "Name": "Acme Books",
        "Offices": [ "Seattle" ]
    },
}
```

```
"V02": {  
    "Name": "New Publishers, Inc.",  
    "Offices": ["London", "New York"  
    ]  
,  
"V03": {  
    "Name": "Better Buy Books",  
    "Offices": [ "Tokyo", "Los Angeles", "Sydney"  
    ]  
}  
}
```

You can use the `withJSON` method to store this in the `ProductCatalog` table, in a `Map` attribute named `VendorInfo`. The following Java code example demonstrates how to do this.

```
// Convert the document into a String. Must escape all double-quotes.  
String vendorDocument = "{"  
+ "    \"V01\": {"  
+ "        \"Name\": \"Acme Books\",  
+ "        \"Offices\": [ \"Seattle\" ]"  
+ "    },"  
+ "    \"V02\": {"  
+ "        \"Name\": \"New Publishers, Inc.\",  
+ "        \"Offices\": [ \"London\", \"New York\" + \"\"]" + "},"  
+ "    \"V03\": {"  
+ "        \"Name\": \"Better Buy Books\",  
+ "        \"Offices\": [ \"Tokyo\", \"Los Angeles\", \"Sydney\""  
+ "            ]"  
+ "        }"  
+ "    }";  
  
Item item = new Item()  
.withPrimaryKey("Id", 210)  
.withString("Title", "Book 210 Title")  
.withString("ISBN", "210-2102102102")  
.withNumber("Price", 30)  
.withJSON("VendorInfo", vendorDocument);  
  
PutItemOutcome outcome = table.putItem(item);
```

Getting an item

To retrieve a single item, use the `getItem` method of a `Table` object. Follow these steps:

1. Create an instance of the DynamoDB class.
2. Create an instance of the Table class to represent the table you want to work with.
3. Call the getItem method of the Table instance. You must specify the primary key of the item that you want to retrieve.

The following Java code example demonstrates the preceding steps. The code gets the item that has the specified partition key.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Item item = table.getItem("Id", 210);
```

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters for the getItem method. For example, the following Java code example uses an optional method to retrieve only a specific list of attributes and to specify strongly consistent reads. (To learn more about read consistency, see [Read consistency](#).)

You can use a ProjectionExpression to retrieve only specific attributes or elements, rather than an entire item. A ProjectionExpression can specify top-level or nested attributes using document paths. For more information, see [Projection expressions](#).

The parameters of the getItem method don't let you specify read consistency. However, you can create a GetItemSpec, which provides full access to all of the inputs to the low-level GetItem operation. The following code example creates a GetItemSpec and uses that spec as input to the getItem method.

Example

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 206)
    .withProjectionExpression("Id, Title, RelatedItems[0], Reviews.FiveStar")
    .withConsistentRead(true);
```

```
Item item = table.getItem(spec);

System.out.println(item.toJSONString());
```

To print an Item in a human-readable format, use the `toJSONPretty` method. The output from the previous example looks like the following.

```
{
    "RelatedItems" : [ 341 ],
    "Reviews" : {
        "FiveStar" : [ "Excellent! Can't recommend it highly enough! Buy it!", "Do yourself
a favor and buy this" ]
    },
    "Id" : 123,
    "Title" : "20-Bicycle 123"
}
```

GetItem and JSON documents

In the [PutItem and JSON documents](#) section, you store a JSON document in a Map attribute named `VendorInfo`. You can use the `getItem` method to retrieve the entire document in JSON format. Or you can use document path notation to retrieve only some of the elements in the document. The following Java code example demonstrates these techniques.

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210);

System.out.println("All vendor info:");
spec.withProjectionExpression("VendorInfo");
System.out.println(table.getItem(spec).toJSON());

System.out.println("A single vendor:");
spec.withProjectionExpression("VendorInfo.V03");
System.out.println(table.getItem(spec).toJSON());

System.out.println("First office location for this vendor:");
spec.withProjectionExpression("VendorInfo.V03.Offices[0]");
System.out.println(table.getItem(spec).toJSON());
```

The output from the previous example looks like the following.

All vendor info:

```
{"VendorInfo":{"V03":{"Name":"Better Buy Books","Offices":["Tokyo","Los Angeles","Sydney"]}, "V02":{"Name":"New Publishers, Inc.","Offices":["London","New York"]}, "V01":{"Name":"Acme Books","Offices":["Seattle"]}}}
```

A single vendor:

```
{"VendorInfo":{"V03":{"Name":"Better Buy Books","Offices":["Tokyo","Los Angeles","Sydney"]}}}
```

First office location for a single vendor:

```
{"VendorInfo":{"V03":{"Offices":["Tokyo"]}}}
```

Note

You can use the `toJSON` method to convert any item (or its attributes) to a JSON-formatted string. The following code retrieves several top-level and nested attributes and prints the results as JSON.

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210)
    .withProjectionExpression("VendorInfo.V01, Title, Price");

Item item = table.getItem(spec);
System.out.println(item.toJSON());
```

The output looks like the following.

```
{"VendorInfo":{"V01":{"Name":"Acme Books","Offices":["Seattle"]}}, "Price":30, "Title":"Book 210 Title"}
```

Batch write: Putting and deleting multiple items

Batch write refers to putting and deleting multiple items in a batch. The `batchWriteItem` method enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to put or delete multiple items using the AWS SDK for Java Document API.

1. Create an instance of the DynamoDB class.
2. Create an instance of the `TableWriteItems` class that describes all the put and delete operations for a table. If you want to write to multiple tables in a single batch write operation, you must create one `TableWriteItems` instance per table.

3. Call the `batchWriteItem` method by providing the `TableWriteItems` objects that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed request items returned in the response. This could happen if you reach the provisioned throughput quota or some other transient error. Also, DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, DynamoDB rejects the request. For more information, see [Service, account, and table quotas in Amazon DynamoDB](#).

The following Java code example demonstrates the preceding steps. The example performs a `batchWriteItem` operation on two tables: `Forum` and `Thread`. The corresponding `TableWriteItems` objects define the following actions:

- Put an item in the `Forum` table.
- Put and delete an item in the `Thread` table.

The code then calls `batchWriteItem` to perform the operation.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableWriteItems forumTableWriteItems = new TableWriteItems("Forum")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("Name", "Amazon RDS")
            .withNumber("Threads", 0));

TableWriteItems threadTableWriteItems = new TableWriteItems("Thread")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("ForumName", "Amazon RDS", "Subject", "Amazon RDS Thread 1")
            .withHashAndRangeKeysToDelete("ForumName", "Some partition key value", "Amazon S3",
                "Some sort key value"));

BatchWriteItemOutcome outcome = dynamoDB.batchWriteItem(forumTableWriteItems,
    threadTableWriteItems);

// Code for checking unprocessed items is omitted in this example
```

For a working example, see [Example: Batch write operation using the AWS SDK for Java document API](#).

Batch get: Getting multiple items

The `batchGetItem` method enables you to retrieve multiple items from one or more tables. To retrieve a single item, you can use the `getItem` method.

Follow these steps:

1. Create an instance of the DynamoDB class.
2. Create an instance of the `TableKeysAndAttributes` class that describes a list of primary key values to retrieve from a table. If you want to read from multiple tables in a single batch get operation, you must create one `TableKeysAndAttributes` instance per table.
3. Call the `batchGetItem` method by providing the `TableKeysAndAttributes` objects that you created in the preceding step.

The following Java code example demonstrates the preceding steps. The example retrieves two items from the `Forum` table and three items from the `Thread` table.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

    TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes(forumTableName);
    forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName", "Subject",
    "Amazon DynamoDB", "DynamoDB Thread 1",
    "Amazon DynamoDB", "DynamoDB Thread 2",
    "Amazon S3", "S3 Thread 1");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(
    forumTableKeysAndAttributes, threadTableKeysAndAttributes);

for (String tableName : outcome.getTableItems().keySet()) {
    System.out.println("Items in table " + tableName);
```

```
    List<Item> items = outcome.getTableItems().get(tableName);
    for (Item item : items) {
        System.out.println(item);
    }
}
```

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters when using `batchGetItem`. For example, you can provide a `ProjectionExpression` with each `TableKeysAndAttributes` you define. This allows you to specify the attributes that you want to retrieve from the table.

The following code example retrieves two items from the `Forum` table. The `withProjectionExpression` parameter specifies that only the `Threads` attribute is to be retrieved.

Example

```
TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes("Forum")
    .withProjectionExpression("Threads");

forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(forumTableKeysAndAttributes);
```

Updating an item

The `updateItem` method of a `Table` object can update existing attribute values, add new attributes, or delete attributes from an existing item.

The `updateItem` method behaves as follows:

- If an item does not exist (no item in the table with the specified primary key), `updateItem` adds a new item to the table.
- If an item exists, `updateItem` performs the update as specified by the `UpdateExpression` parameter.

Note

It is also possible to "update" an item using `putItem`. For example, if you call `putItem` to add an item to the table, but there is already an item with the specified primary key, `putItem` replaces the entire item. If there are attributes in the existing item that are not specified in the input, `putItem` removes those attributes from the item.

In general, we recommend that you use `updateItem` whenever you want to modify any item attributes. The `updateItem` method only modifies the item attributes that you specify in the input, and the other attributes in the item remain unchanged.

Follow these steps:

1. Create an instance of the `Table` class to represent the table that you want to work with.
2. Call the `updateTable` method of the `Table` instance. You must specify the primary key of the item that you want to retrieve, along with an `UpdateExpression` that describes the attributes to modify and how to modify them.

The following Java code example demonstrates the preceding tasks. The code updates a book item in the `ProductCatalog` table. It adds a new author to the set of `Authors` and deletes the existing `ISBN` attribute. It also reduces the price by one.

An `ExpressionAttributeValues` map is used in the `UpdateExpression`. The placeholders `:val1` and `:val2` are replaced at runtime with the actual values for `Authors` and `Price`.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#A", "Authors");
expressionAttributeNames.put("#P", "Price");
expressionAttributeNames.put("#I", "ISBN");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1",
    new HashSet<String>(Arrays.asList("Author YY","Author ZZ")));
expressionAttributeValues.put(":val2", 1); //Price
```

```
UpdateItemOutcome outcome = table.updateItem(  
    "Id",           // key attribute name  
    101,            // key attribute value  
    "add #A :val1 set #P = #P - :val2 remove #I", // UpdateExpression  
    expressionAttributeNames,  
    expressionAttributeValues);
```

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters for the updateItem method, including a condition that must be met in order for the update is to occur. If the condition you specify is not met, the AWS SDK for Java throws a ConditionalCheckFailedException. For example, the following Java code example conditionally updates a book item price to 25. It specifies a ConditionExpression stating that the price should be updated only if the existing price is 20.

Example

```
Table table = dynamoDB.getTable("ProductCatalog");  
  
Map<String, String> expressionAttributeNames = new HashMap<String, String>();  
expressionAttributeNames.put("#P", "Price");  
  
Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();  
expressionAttributeValues.put(":val1", 25); // update Price to 25...  
expressionAttributeValues.put(":val2", 20); //...but only if existing Price is 20  
  
UpdateItemOutcome outcome = table.updateItem(  
    new PrimaryKey("Id",101),  
    "set #P = :val1", // UpdateExpression  
    "#P = :val2",    // ConditionExpression  
    expressionAttributeNames,  
    expressionAttributeValues);
```

Atomic counter

You can use updateItem to implement an atomic counter, where you increment or decrement the value of an existing attribute without interfering with other write requests. To increment an atomic counter, use an UpdateExpression with a set action to add a numeric value to an existing attribute of type Number.

The following example demonstrates this, incrementing the `Quantity` attribute by one. It also demonstrates the use of the `ExpressionAttributeNames` parameter in an `UpdateExpression`.

```
Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#p", "PageCount");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", 1);

UpdateItemOutcome outcome = table.updateItem(
    "Id", 121,
    "set #p = #p + :val",
    expressionAttributeNames,
    expressionAttributeValues);
```

Deleting an item

The `deleteItem` method deletes an item from a table. You must provide the primary key of the item that you want to delete.

Follow these steps:

1. Create an instance of the DynamoDB client.
2. Call the `deleteItem` method by providing the key of the item you want to delete.

The following Java example demonstrates these tasks.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

DeleteItemOutcome outcome = table.deleteItem("Id", 101);
```

Specifying optional parameters

You can specify optional parameters for `deleteItem`. For example, the following Java code example specifies a `ConditionExpression`, stating that a book item in `ProductCatalog` can only be deleted if the book is no longer in publication (the `InPublication` attribute is false).

Example

```
Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", false);

DeleteItemOutcome outcome = table.deleteItem("Id", 103,
    "InPublication = :val",
    null, // ExpressionAttributeNames - not used in this example
    expressionAttributeValues);
```

Example: CRUD operations using the AWS SDK for Java document API

The following code example illustrates CRUD operations on an Amazon DynamoDB item. The example creates an item, retrieves it, performs various updates, and finally deletes the item.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
```

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DeleteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class DocumentAPIItemCRUDExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "ProductCatalog";

    public static void main(String[] args) throws IOException {

        createItems();

        retrieveItem();

        // Perform various updates.
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

        // Delete the item.
        deleteItem();

    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tableName);
```

```
try {

    Item item = new Item().withPrimaryKey("Id", 120).withString("Title", "Book
120 Title")
        .withString("ISBN", "120-1111111111")
        .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author12", "Author22")))
        .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
        .withBoolean("InPublication", false).withString("ProductCategory",
"Book");
    table.putItem(item);

    item = new Item().withPrimaryKey("Id", 121).withString("Title", "Book 121
Title")
        .withString("ISBN", "121-1111111111")
        .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author21", "Author 22")))
        .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
        .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
    table.putItem(item);

} catch (Exception e) {
    System.err.println("Create items failed.");
    System.err.println(e.getMessage());
}

}

private static void retrieveItem() {
    Table table = dynamoDB.getTable(tableName);

    try {

        Item item = table.getItem("Id", 120, "Id, ISBN, Title, Authors", null);

        System.out.println("Printing item after retrieving it....");
        System.out.println(item.toJSONString());

    } catch (Exception e) {
        System.err.println("GetItem failed.");
        System.err.println(e.getMessage());
    }
}
```

```
    }

}

private static void updateAddNewAttribute() {
    Table table = dynamoDB.getTable(tableName);

    try {

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
121)            .withUpdateExpression("set #na = :val1").withNameMap(new
NameMap().with("#na", "NewAttribute"))
            .WithValueMap(new ValueMap().withString(":val1", "Some value"))
            .withReturnValues(ReturnValue.ALL_NEW);

        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out.println("Printing item after adding new attribute...");
        System.out.println(outcome.getItem().toJSONPretty());

    } catch (Exception e) {
        System.err.println("Failed to add new attribute in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void updateMultipleAttributes() {

    Table table = dynamoDB.getTable(tableName);

    try {

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
120)            .withUpdateExpression("add #a :val1 set #na=:val2")
            .withNameMap(new NameMap().with("#a", "Authors").with("#na",
"NewAttribute"))
            .WithValueMap(
                new ValueMap().withStringSet(":val1", "Author YY", "Author
ZZ").withString(":val2",
                    "someValue"))
            .withReturnValues(ReturnValue.ALL_NEW);

    }
```

```
        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out.println("Printing item after multiple attribute update...");
        System.out.println(outcome.getItem().toJSONPretty());

    } catch (Exception e) {
        System.err.println("Failed to update multiple attributes in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void updateExistingAttributeConditionally() {

    Table table = dynamoDB.getTable(tableName);

    try {

        // Specify the desired price (25.00) and also the condition (price =
        // 20.00)

        120)           UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
                .withReturnValues(ReturnValue.ALL_NEW).withUpdateExpression("set #p
= :val1")
                .withConditionExpression("#p = :val2").withNameMap(new
NameMap().with("#p", "Price"))
                .WithValueMap(new ValueMap().withNumber(":val1",
25).withNumber(":val2", 20));

        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out.println("Printing item after conditional update to new
attribute...");
        System.out.println(outcome.getItem().toJSONPretty());

    } catch (Exception e) {
        System.err.println("Error updating item in " + tableName);
        System.err.println(e.getMessage());
    }
}
```

```
private static void deleteItem() {

    Table table = dynamoDB.getTable(tableName);

    try {

        DeleteItemSpec deleteItemSpec = new DeleteItemSpec().withPrimaryKey("Id",
120)
            .withConditionExpression("#ip = :val").withNameMap(new
NameMap().with("#ip", "InPublication"))
            .WithValueMap(new ValueMap().withBoolean(":val",
false)).withReturnValues(ReturnValue.ALL_OLD);

        DeleteItemOutcome outcome = table.deleteItem(deleteItemSpec);

        // Check the response.
        System.out.println("Printing item that was deleted...");
        System.out.println(outcome.getItem().toJSONPretty());

    } catch (Exception e) {
        System.err.println("Error deleting item in " + tableName);
        System.err.println(e.getMessage());
    }
}
}
```

Example: Batch operations using AWS SDK for Java document API

This section provides examples of batch write and batch get operations in Amazon DynamoDB using the AWS SDK for Java Document API.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

Topics

- [Example: Batch write operation using the AWS SDK for Java document API](#)

- [Example: Batch get operation using the AWS SDK for Java document API](#)

Example: Batch write operation using the AWS SDK for Java document API

The following Java code example uses the `batchWriteItem` method to perform the following put and delete operations:

- Put one item in the `Forum` table.
- Put one item and delete one item from the `Thread` table.

You can specify any number of put and delete requests against one or more tables when creating your batch write request. However, `batchWriteItem` limits the size of a batch write request and the number of put and delete operations in a single batch write operation. If your request exceeds these limits, your request is rejected. If your table does not have sufficient provisioned throughput to serve this request, the unprocessed request items are returned in the response.

The following example checks the response to see if it has any unprocessed request items. If it does, it loops back and resends the `batchWriteItem` request with unprocessed items in the request. If you followed the [Creating tables and loading data for code examples in DynamoDB](#) section, you should already have created the `Forum` and `Thread` tables. You can also create these tables and upload sample data programmatically. For more information, see [Creating example tables and uploading data using the AWS SDK for Java](#).

For step-by-step instructions for testing the following sample, see [Java code examples](#).

Example

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchWriteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
```

```
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableWriteItems;
import com.amazonaws.services.dynamodbv2.model.WriteRequest;

public class DocumentAPIBatchWrite {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {

        writeMultipleItemsBatchWrite();

    }

    private static void writeMultipleItemsBatchWrite() {
        try {

            // Add a new item to Forum
            TableWriteItems forumTableWriteItems = new
TableWriteItems(forumTableName) // Forum
                .withItemsToPut(new Item().withPrimaryKey("Name", "Amazon
RDS").withNumber("Threads", 0));

            // Add a new item, and delete an existing item, from Thread
            // This table has a partition key and range key, so need to specify
            // both of them
            TableWriteItems threadTableWriteItems = new
TableWriteItems(threadTableName)
                .withItemsToPut(
                    new Item().withPrimaryKey("ForumName", "Amazon RDS",
"Subject", "Amazon RDS Thread 1")
                        .withString("Message", "ElastiCache Thread 1
message")
                        .withStringSet("Tags", new
HashSet<String>(Arrays.asList("cache", "in-memory"))))
                .withHashAndRangeKeysToDelete("ForumName", "Subject", "Amazon S3",
"S3 Thread 100");

            System.out.println("Making the request.");
        }
    }
}
```

```
        BatchWriteItemOutcome outcome =
dynamoDB.batchWriteItem(forumTableWriteItems, threadTableWriteItems);

        do {

            // Check for unprocessed keys which could happen if you exceed
            // provisioned throughput

            Map<String, List<WriteRequest>> unprocessedItems =
outcome.getUnprocessedItems();

            if (outcome.getUnprocessedItems().size() == 0) {
                System.out.println("No unprocessed items found");
            } else {
                System.out.println("Retrieving the unprocessed items");
                outcome = dynamoDB.batchWriteItemUnprocessed(unprocessedItems);
            }

        } while (outcome.getUnprocessedItems().size() > 0);

    } catch (Exception e) {
        System.err.println("Failed to retrieve items: ");
        e.printStackTrace(System.err);
    }
}

}
```

Example: Batch get operation using the AWS SDK for Java document API

The following Java code example uses the batchGetItem method to retrieve multiple items from the Forum and the Thread tables. The BatchGetItemRequest specifies the table names and a list of keys for each item to get. The example processes the response by printing the items retrieved.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

Example

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchGetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableKeysAndAttributes;
import com.amazonaws.services.dynamodbv2.model.KeysAndAttributes;

public class DocumentAPIBatchGet {
    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {
        retrieveMultipleItemsBatchGet();
    }

    private static void retrieveMultipleItemsBatchGet() {

        try {

            TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes(forumTableName);
                // Add a partition key
                forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name", "Amazon S3",
"Amazon DynamoDB");

            TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
```

```
// Add a partition key and a sort key
threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName",
"Subject", "Amazon DynamoDB",
    "DynamoDB Thread 1", "Amazon DynamoDB", "DynamoDB Thread 2",
"Amazon S3", "S3 Thread 1");

System.out.println("Making the request.");

BatchGetItemOutcome outcome =
dynamoDB.batchGetItem(forumTableKeysAndAttributes,
    threadTableKeysAndAttributes);

Map<String, KeysAndAttributes> unprocessed = null;

do {
    for (String tableName : outcome.getTableItems().keySet()) {
        System.out.println("Items in table " + tableName);
        List<Item> items = outcome.getTableItems().get(tableName);
        for (Item item : items) {
            System.out.println(item.toJSONString());
        }
    }
}

// Check for unprocessed keys which could happen if you exceed
// provisioned
// throughput or reach the limit on response size.
unprocessed = outcome.getUnprocessedKeys();

if (unprocessed.isEmpty()) {
    System.out.println("No unprocessed keys found");
} else {
    System.out.println("Retrieving the unprocessed keys");
    outcome = dynamoDB.batchGetItemUnprocessed(unprocessed);
}

} while (!unprocessed.isEmpty());

} catch (Exception e) {
    System.err.println("Failed to retrieve items.");
    System.err.println(e.getMessage());
}

}
```

```
}
```

Example: Handling binary type attributes using the AWS SDK for Java document API

The following Java code example illustrates handling binary type attributes. The example adds an item to the Reply table. The item includes a binary type attribute (ExtendedMessage) that stores compressed data. The example then retrieves the item and prints all the attribute values. For illustration, the example uses the GZIPOutputStream class to compress a sample stream and assign it to the ExtendedMessage attribute. When the binary attribute is retrieved, it is decompressed using the GZIPInputStream class.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

If you followed the [Creating tables and loading data for code examples in DynamoDB](#) section, you should already have created the Reply table. You can also create this table programmatically. For more information, see [Creating example tables and uploading data using the AWS SDK for Java](#).

For step-by-step instructions for testing the following sample, see [Java code examples](#).

Example

```
package com.amazonaws.codesamples.document;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.TimeZone;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.GetItemSpec;

public class DocumentAPIItemBinaryExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "Reply";
    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    public static void main(String[] args) throws IOException {
        try {

            // Format the primary key values
            String threadId = "Amazon DynamoDB#DynamoDB Thread 2";

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
            String replyDateTime = dateFormatter.format(new Date());

            // Add a new reply with a binary attribute type
            createItem(threadId, replyDateTime);

            // Retrieve the reply with a binary attribute type
            retrieveItem(threadId, replyDateTime);

            // clean up by deleting the item
            deleteItem(threadId, replyDateTime);
        } catch (Exception e) {
            System.err.println("Error running the binary attribute type example: " +
e);
            e.printStackTrace(System.err);
        }
    }

    public static void createItem(String threadId, String replyDateTime) throws
IOException {

        Table table = dynamoDB.getTable(tableName);
```

```
// Craft a long message
String messageInput = "Long message to be compressed in a lengthy forum reply";

// Compress the long message
ByteBuffer compressedMessage = compressString(messageInput.toString());

table.putItem(new Item().withPrimaryKey("Id",
threadId).withString("ReplyDateTime", replyDateTime)
    .withString("Message", "Long message
follows").withBinary("ExtendedMessage", compressedMessage)
    .withString("PostedBy", "User A"));
}

public static void retrieveItem(String threadId, String replyDateTime) throws
IOException {

    Table table = dynamoDB.getTable(tableName);

    GetItemSpec spec = new GetItemSpec().withPrimaryKey("Id", threadId,
"ReplyDateTime", replyDateTime)
        .withConsistentRead(true);

    Item item = table.getItem(spec);

    // Uncompress the reply message and print
    String uncompressed =
uncompressString(ByteBuffer.wrap(item.getBinary("ExtendedMessage")));

    System.out.println("Reply message:\n" + " Id: " + item.getString("Id") + "\n" +
" ReplyDateTime: "
        + item.getString("ReplyDateTime") + "\n" + " PostedBy: " +
item.getString("PostedBy") + "\n"
        + " Message: "
        + item.getString("Message") + "\n" + " ExtendedMessage (uncompressed):
" + uncompressed + "\n");
}

public static void deleteItem(String threadId, String replyDateTime) {

    Table table = dynamoDB.getTable(tableName);
    table.deleteItem("Id", threadId, "ReplyDateTime", replyDateTime);
}

private static ByteBuffer compressString(String input) throws IOException {
```

```
// Compress the UTF-8 encoded String into a byte[]
ByteArrayOutputStream baos = new ByteArrayOutputStream();
GZIPOutputStream os = new GZIPOutputStream(baos);
os.write(input.getBytes("UTF-8"));
os.close();
baos.close();
byte[] compressedBytes = baos.toByteArray();

// The following code writes the compressed bytes to a ByteBuffer.
// A simpler way to do this is by simply calling
// ByteBuffer.wrap(compressedBytes);
// However, the longer form below shows the importance of resetting the
// position of the buffer
// back to the beginning of the buffer if you are writing bytes directly
// to it, since the SDK
// will consider only the bytes after the current position when sending
// data to DynamoDB.
// Using the "wrap" method automatically resets the position to zero.
ByteBuffer buffer = ByteBuffer.allocate(compressedBytes.length);
buffer.put(compressedBytes, 0, compressedBytes.length);
buffer.position(0); // Important: reset the position of the ByteBuffer
                  // to the beginning
return buffer;
}

private static String uncompressString(ByteBuffer input) throws IOException {
    byte[] bytes = input.array();
    ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPInputStream is = new GZIPInputStream(bais);

    int chunkSize = 1024;
    byte[] buffer = new byte[chunkSize];
    int length = 0;
    while ((length = is.read(buffer, 0, chunkSize)) != -1) {
        baos.write(buffer, 0, length);
    }

    String result = new String(baos.toByteArray(), "UTF-8");

    is.close();
    baos.close();
    bais.close();
}
```

```
        return result;
    }
}
```

Working with items: .NET

You can use the AWS SDK for .NET low-level API to perform typical create, read, update, and delete (CRUD) operations on an item in a table. The following are the common steps that you follow to perform data CRUD operations using the .NET low-level API:

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
2. Provide the operation-specific required parameters in a corresponding request object.

For example, use the `PutItemRequest` request object when uploading an item and use the `GetItemRequest` request object when retrieving an existing item.

You can use the request object to provide both the required and optional parameters.

3. Run the appropriate method provided by the client by passing in the request object that you created in the preceding step.

The `AmazonDynamoDBClient` client provides `PutItem`, `GetItem`, `UpdateItem`, and `DeleteItem` methods for the CRUD operations.

Topics

- [Putting an item](#)
- [Getting an item](#)
- [Updating an item](#)
- [Atomic counter](#)
- [Deleting an item](#)
- [Batch write: Putting and deleting multiple items](#)
- [Batch get: Getting multiple items](#)
- [Example: CRUD operations using the AWS SDK for .NET low-level API](#)
- [Example: Batch operations using the AWS SDK for .NET low-level API](#)
- [Example: Handling binary type attributes using the AWS SDK for .NET low-level API](#)

Putting an item

The PutItem method uploads an item to a table. If the item exists, it replaces the entire item.

Note

Instead of replacing the entire item, if you want to update only specific attributes, you can use the UpdateItem method. For more information, see [Updating an item](#).

The following are the steps to upload an item using the low-level .NET SDK API:

1. Create an instance of the AmazonDynamoDBClient class.
2. Provide the required parameters by creating an instance of the PutItemRequest class.

To put an item, you must provide the table name and the item.

3. Run the PutItem method by providing the PutItemRequest object that you created in the preceding step.

The following C# example demonstrates the preceding steps. The example uploads an item to the ProductCatalog table.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new PutItemRequest
{
    TableName = tableName,
    Item = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue { N = "201" } },
        { "Title", new AttributeValue { S = "Book 201 Title" } },
        { "ISBN", new AttributeValue { S = "11-11-11-11" } },
        { "Price", new AttributeValue { S = "20.00" } },
        {
            "Authors",
            new AttributeValue
            { SS = new List<string>{"Author1", "Author2"} }
        }
    }
}
```

```
    }
};

client.PutItem(request);
```

In the preceding example, you upload a book item that has the `Id`, `Title`, `ISBN`, and `Authors` attributes. Note that `Id` is a numeric type attribute, and all other attributes are of the string type. `Authors` is a `String` set.

Specifying optional parameters

You can also provide optional parameters using the `PutItemRequest` object as shown in the following C# example. The example specifies the following optional parameters:

- `ExpressionAttributeNames`, `ExpressionAttributeValues`, and `ConditionExpression` specify that the item can be replaced only if the existing item has the `ISBN` attribute with a specific value.
- `ReturnValues` parameter to request the old item in the response.

Example

```
var request = new PutItemRequest
{
    TableName = tableName,
    Item = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue { N = "104" } },
        { "Title", new AttributeValue { S = "Book 104 Title" } },
        { "ISBN", new AttributeValue { S = "444-4444444444" } },
        { "Authors",
            new AttributeValue { SS = new List<string>{"Author3"} }
        },
        // Optional parameters.
        ExpressionAttributeNames = new Dictionary<string, string>()
        {
            {"#I", "ISBN"}
        },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
        {
            {":isbn", new AttributeValue { S = "444-4444444444" } }
        },
        ConditionExpression = "#I = :isbn"
```

```
};  
var response = client.PutItem(request);
```

For more information, see [PutItem](#).

Getting an item

The GetItem method retrieves an item.

Note

To retrieve multiple items, you can use the BatchGetItem method. For more information, see [Batch get: Getting multiple items](#).

The following are the steps to retrieve an existing item using the low-level AWS SDK for .NET API.

1. Create an instance of the AmazonDynamoDBClient class.
2. Provide the required parameters by creating an instance of the GetItemRequest class.

To get an item, you must provide the table name and primary key of the item.

3. Run the GetItem method by providing the GetItemRequest object that you created in the preceding step.

The following C# example demonstrates the preceding steps. The example retrieves an item from the ProductCatalog table.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
string tableName = "ProductCatalog";  
  
var request = new GetItemRequest  
{  
    TableName = tableName,  
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N = "202" } } },  
};  
var response = client.GetItem(request);  
  
// Check the response.  
var result = response.GetItemResult;
```

```
var attributeMap = result.Item; // Attribute list in the response.
```

Specifying optional parameters

You can also provide optional parameters using the `GetItemRequest` object, as shown in the following C# example. The sample specifies the following optional parameters:

- `ProjectionExpression` parameter to specify the attributes to retrieve.
- `ConsistentRead` parameter to perform a strongly consistent read. To learn more read consistency, see [Read consistency](#).

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new GetItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
"202" } } },
    // Optional parameters.
    ProjectionExpression = "Id, ISBN, Title, Authors",
    ConsistentRead = true
};

var response = client.GetItem(request);

// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item;
```

For more information, see [GetItem](#).

Updating an item

The `UpdateItem` method updates an existing item if it is present. You can use the `UpdateItem` operation to update existing attribute values, add new attributes, or delete attributes from the existing collection. If the item that has the specified primary key is not found, it adds a new item.

The `UpdateItem` operation uses the following guidelines:

- If the item does not exist, `UpdateItem` adds a new item using the primary key that is specified in the input.
- If the item exists, `UpdateItem` applies the updates as follows:
 - Replaces the existing attribute values by the values in the update.
 - If the attribute that you provide in the input does not exist, it adds a new attribute to the item.
 - If the input attribute is null, it deletes the attribute, if it is present.
 - If you use ADD for the Action, you can add values to an existing set (string or number set), or mathematically add (use a positive number) or subtract (use a negative number) from the existing numeric attribute value.

 **Note**

The `PutItem` operation also can perform an update. For more information, see [Putting an item](#). For example, if you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. If there are attributes in the existing item and those attributes are not specified in the input, the `PutItem` operation deletes those attributes. However, `UpdateItem` updates only the specified input attributes. Any other existing attributes of that item remain unchanged.

The following are the steps to update an existing item using the low-level .NET SDK API:

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `UpdateItemRequest` class.

This is the request object in which you describe all the updates, such as add attributes, update existing attributes, or delete attributes. To delete an existing attribute, specify the attribute name with null value.

3. Run the `UpdateItem` method by providing the `UpdateItemRequest` object that you created in the preceding step.

The following C# code example demonstrates the preceding steps. The example updates a book item in the `ProductCatalog` table. It adds a new author to the `Authors` collection, and deletes the existing `ISBN` attribute. It also reduces the price by one.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
"202" } } },
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#A", "Authors"},
        {"#P", "Price"},
        {"#NA", "NewAttribute"},
        {"#I", "ISBN"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {"":auth",new AttributeValue { SS = {"Author YY","Author ZZ"} } },
        {"":p",new AttributeValue { N = "1" } },
        {"":newattr",new AttributeValue { S = "someValue" } },
    },
    // This expression does the following:
    // 1) Adds two new authors to the list
    // 2) Reduces the price
    // 3) Adds a new attribute to the item
    // 4) Removes the ISBN attribute from the item
    UpdateExpression = "ADD #A :auth SET #P = #P - :p, #NA = :newattr REMOVE #I"
};

var response = client.UpdateItem(request);
```

Specifying optional parameters

You can also provide optional parameters using the `UpdateItemRequest` object, as shown in the following C# example. It specifies the following optional parameters:

- `ExpressionAttributeValues` and `ConditionExpression` to specify that the price can be updated only if the existing price is 20.00.
- `ReturnValues` parameter to request the updated item in the response.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
"202" } } },
    // Update price only if the current price is 20.00.
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#P", "Price"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {"<:newprice",new AttributeValue {N = "22"}},
        {"<:currprice",new AttributeValue {N = "20"}}
    },
    UpdateExpression = "SET #P = :newprice",
    ConditionExpression = "#P = :currprice",
    TableName = tableName,
    ReturnValues = "ALL_NEW" // Return all the attributes of the updated item.
};

var response = client.UpdateItem(request);
```

For more information, see [UpdateItem](#).

Atomic counter

You can use `updateItem` to implement an atomic counter, where you increment or decrement the value of an existing attribute without interfering with other write requests. To update an atomic counter, use `updateItem` with an attribute of type `Number` in the `UpdateExpression` parameter, and `ADD` as the Action.

The following example demonstrates this, incrementing the `Quantity` attribute by one.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
```

```
{  
    Key = new Dictionary<string, AttributeValue>() { { "Id", new AttributeValue { N =  
"121" } } },  
    ExpressionAttributeNames = new Dictionary<string, string>()  
    {  
        {"#Q", "Quantity"}  
    },  
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()  
    {  
        {":incr", new AttributeValue { N = "1"} }  
    },  
    UpdateExpression = "SET #Q = #Q + :incr",  
    TableName = tableName  
};  
  
var response = client.UpdateItem(request);
```

Deleting an item

The `DeleteItem` method deletes an item from a table.

The following are the steps to delete an item using the low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `DeleteItemRequest` class.
To delete an item, the table name and item's primary key are required.
3. Run the `DeleteItem` method by providing the `DeleteItemRequest` object that you created in the preceding step.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
string tableName = "ProductCatalog";  
  
var request = new DeleteItemRequest  
{  
    TableName = tableName,  
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =  
"201" } } },  
};
```

```
var response = client.DeleteItem(request);
```

Specifying optional parameters

You can also provide optional parameters using the `DeleteItemRequest` object as shown in the following C# code example. It specifies the following optional parameters:

- `ExpressionAttributeValues` and `ConditionExpression` to specify that the book item can be deleted only if it is no longer in publication (the `InPublication` attribute value is false).
- `ReturnValues` parameter to request the deleted item in the response.

Example

```
var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
"201" } } },

    // Optional parameters.
    ReturnValues = "ALL_OLD",
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#IP", "InPublication"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":inpub",new AttributeValue {BOOL = false}}
    },
    ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);
```

For more information, see [DeleteItem](#).

Batch write: Putting and deleting multiple items

Batch write refers to putting and deleting multiple items in a batch. The `BatchWriteItem` method enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to retrieve multiple items using the low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Describe all the put and delete operations by creating an instance of the `BatchWriteItemRequest` class.
3. Run the `BatchWriteItem` method by providing the `BatchWriteItemRequest` object that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed request items returned in the response. This could happen if you reach the provisioned throughput quota or some other transient error. Also, DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, DynamoDB rejects the request. For more information, see [BatchWriteItem](#).

The following C# code example demonstrates the preceding steps. The example creates a `BatchWriteItemRequest` to perform the following write operations:

- Put an item in `Forum` table.
- Put and delete an item from `Thread` table.

The code runs `BatchWriteItem` to perform a batch operation.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchWriteItemRequest
{
    RequestItems = new Dictionary<string, List<WriteRequest>>
    {
        {
            table1Name, new List<WriteRequest>
            {
                new WriteRequest
                {
                    PutRequest = new PutRequest
                    {
                        Item = new Dictionary<string,AttributeValue>
                        {
                            { "Name", new AttributeValue { S = "Amazon S3 forum" } },
                            { "Threads", new AttributeValue { N = "0" } }
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}
},
{
    table2Name, new List<WriteRequest>
{
    new WriteRequest
    {
        PutRequest = new PutRequest
        {
            Item = new Dictionary<string,AttributeValue>
            {
                { "ForumName", new AttributeValue { S = "Amazon S3 forum" } },
                { "Subject", new AttributeValue { S = "My sample question" } },
                { "Message", new AttributeValue { S = "Message Text." } },
                { "KeywordTags", new AttributeValue { SS = new List<string> { "Amazon S3", "Bucket" } } }
            }
        }
    },
    new WriteRequest
    {
        DeleteRequest = new DeleteRequest
        {
            Key = new Dictionary<string,AttributeValue>()
            {
                { "ForumName", new AttributeValue { S = "Some forum name" } },
                { "Subject", new AttributeValue { S = "Some subject" } }
            }
        }
    }
}
};

response = client.BatchWriteItem(request);
```

For a working example, see [Example: Batch operations using the AWS SDK for .NET low-level API](#).

Batch get: Getting multiple items

The BatchGetItem method enables you to retrieve multiple items from one or more tables.

Note

To retrieve a single item, you can use the `GetItem` method.

The following are the steps to retrieve multiple items using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `BatchGetItemRequest` class.

To retrieve multiple items, the table name and a list of primary key values are required.

3. Run the `BatchGetItem` method by providing the `BatchGetItemRequest` object that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed keys, which could happen if you reach the provisioned throughput quota or some other transient error.

The following C# code example demonstrates the preceding steps. The example retrieves items from two tables, `Forum` and `Thread`. The request specifies two items in the `Forum` and three items in the `Thread` table. The response includes items from both of the tables. The code shows how you can process the response.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchGetItemRequest
{
    RequestItems = new Dictionary<string, KeysAndAttributes>()
    {
        { table1Name,
            new KeysAndAttributes
            {
                Keys = new List<Dictionary<string, AttributeValue>>()
                {
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "DynamoDB" } }
                    },
                }
            }
        }
    }
};
```

```
        new Dictionary<string, AttributeValue>()
    {
        { "Name", new AttributeValue { S = "Amazon S3" } }
    }
}
},
{
    table2Name,
    new KeysAndAttributes
    {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue { S = "DynamoDB" } },
                { "Subject", new AttributeValue { S = "DynamoDB Thread 1" } }
            },
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue { S = "DynamoDB" } },
                { "Subject", new AttributeValue { S = "DynamoDB Thread 2" } }
            },
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue { S = "Amazon S3" } },
                { "Subject", new AttributeValue { S = "Amazon S3 Thread 1" } }
            }
        }
    }
},
};

var response = client.BatchGetItem(request);

// Check the response.
var result = response.BatchGetItemResult;
var responses = result.Responses; // The attribute list in the response.

var table1Results = responses[table1Name];
Console.WriteLine("Items in table {0}" + table1Name);
foreach (var item1 in table1Results.Items)
{
```

```
    PrintItem(item1);
}

var table2Results = responses[table2Name];
Console.WriteLine("Items in table {1}" + table2Name);
foreach (var item2 in table2Results.Items)
{
    PrintItem(item2);
}
// Any unprocessed keys? could happen if you exceed ProvisionedThroughput or some other
// error.
Dictionary<string, KeysAndAttributes> unprocessedKeys = result.UnprocessedKeys;
foreach (KeyValuePair<string, KeysAndAttributes> pair in unprocessedKeys)
{
    Console.WriteLine(pair.Key, pair.Value);
}
```

Specifying optional parameters

You can also provide optional parameters using the `BatchGetItemRequest` object as shown in the following C# code example. The example retrieves two items from the `Forum` table. It specifies the following optional parameter:

- `ProjectionExpression` parameter to specify the attributes to retrieve.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";

var request = new BatchGetItemRequest
{
    RequestItems = new Dictionary<string, KeysAndAttributes>()
    {
        { table1Name,
            new KeysAndAttributes
            {
                Keys = new List<Dictionary<string, AttributeValue>>()
                {
                    new Dictionary<string, AttributeValue>()
                    {

```

```
        { "Name", new AttributeValue { S = "DynamoDB" } }
    },
    new Dictionary<string, AttributeValue>()
    {
        { "Name", new AttributeValue { S = "Amazon S3" } }
    }
},
// Optional - name of an attribute to retrieve.
ProjectionExpression = "Title"
}
}
};

var response = client.BatchGetItem(request);
```

For more information, see [BatchGetItem](#).

Example: CRUD operations using the AWS SDK for .NET low-level API

The following C# code example illustrates CRUD operations on an Amazon DynamoDB item. The example adds an item to the ProductCatalog table, retrieves it, performs various updates, and finally deletes the item. If you followed the steps in [Creating tables and loading data for code examples in DynamoDB](#), you already have the ProductCatalog table created. You can also create these sample tables programmatically. For more information, see [Creating example tables and uploading data using the AWS SDK for .NET](#).

For step-by-step instructions for testing the following sample, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelItemCRUDExample
    {
        private static string tableName = "ProductCatalog";
```

```
private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

static void Main(string[] args)
{
    try
    {
        CreateItem();
        RetrieveItem();

        // Perform various updates.
        UpdateMultipleAttributes();
        UpdateExistingAttributeConditionally();

        // Delete item.
        DeleteItem();
        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
    }
}

private static void CreateItem()
{
    var request = new PutItemRequest
    {
        TableName = tableName,
        Item = new Dictionary<string, AttributeValue>()
    };
    { "Id", new AttributeValue {
            N = "1000"
        }},
    { "Title", new AttributeValue {
            S = "Book 201 Title"
        }},
    { "ISBN", new AttributeValue {
            S = "11-11-11-11"
        }},
    { "Authors", new AttributeValue {
            SS = new List<string>{"Author1", "Author2" }
        }}
}
```

```
        },
        { "Price", new AttributeValue {
            N = "20.00"
        }},
        { "Dimensions", new AttributeValue {
            S = "8.5x11.0x.75"
        }},
        { "InPublication", new AttributeValue {
            BOOL = false
        } }
    }
};

client.PutItem(request);
}

private static void RetrieveItem()
{
    var request = new GetItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            N = "1000"
        } }
    },
        ProjectionExpression = "Id, ISBN, Title, Authors",
        ConsistentRead = true
    };
    var response = client.GetItem(request);

    // Check the response.
    var attributeList = response.Item; // attribute list in the response.
    Console.WriteLine("\nPrinting item after retrieving it .....");
    PrintItem(attributeList);
}

private static void UpdateMultipleAttributes()
{
    var request = new UpdateItemRequest
    {
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
```

```
        N = "1000"
    }
},
// Perform the following updates:
// 1) Add two new authors to the list
// 1) Set a new attribute
// 2) Remove the ISBN attribute
ExpressionAttributeNames = new Dictionary<string, string>()
{
    {"#A", "Authors"},
    {"#NA", "NewAttribute"},
    {"#I", "ISBN"}
},
ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
{
    {":auth", new AttributeValue {
        SS = {"Author YY", "Author ZZ"}
    }},
    {":new", new AttributeValue {
        S = "New Value"
    }}
},
UpdateExpression = "ADD #A :auth SET #NA = :new REMOVE #I",
TableName = tableName,
ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
};

var response = client.UpdateItem(request);

// Check the response.
var attributeList = response.Attributes; // attribute list in the response.
                                         // print attributeList.
Console.WriteLine("\nPrinting item after multiple attribute
update .....");
PrintItem(attributeList);
}

private static void UpdateExistingAttributeConditionally()
{
    var request = new UpdateItemRequest
    {
        Key = new Dictionary<string, AttributeValue>()
    }
```

```
        { "Id", new AttributeValue {
            N = "1000"
        } }
    },
    ExpressionAttributeNames = new Dictionary<string, string>()
{
    {"#P", "Price"}
},
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
{
    {"<:newprice", new AttributeValue {
        N = "22.00"
    }},
    {"<:currprice", new AttributeValue {
        N = "20.00"
    }}
},
// This updates price only if current price is 20.00.
UpdateExpression = "SET #P = :newprice",
ConditionExpression = "#P = :currprice",

TableName = tableName,
ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
};

var response = client.UpdateItem(request);

// Check the response.
var attributeList = response.Attributes; // attribute list in the response.
Console.WriteLine("\nPrinting item after updating price value
conditionally .....");
PrintItem(attributeList);
}

private static void DeleteItem()
{
    var request = new DeleteItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            N = "1000"
        } }
    },
}
```

```
// Return the entire item as it appeared before the update.
ReturnValues = "ALL_OLD",
ExpressionAttributeNames = new Dictionary<string, string>()
{
    {"#IP", "InPublication"}
},
ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
{
    {":inpub", new AttributeValue {
        BOOL = false
    }}
},
ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);

// Check the response.
var attributeList = response.Attributes; // Attribute list in the response.
                                            // Print item.
Console.WriteLine("\nPrinting item that was just deleted .....");
PrintItem(attributeList);
}

private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
{
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
    {
        string attributeName = kvp.Key;
        AttributeValue value = kvp.Value;

        Console.WriteLine(
            attributeName + " " +
            (value.S == null ? "" : "S=[" + value.S + "]") +
            (value.N == null ? "" : "N=[" + value.N + "]") +
            (value.SS == null ? "" : "SS=[" + string.Join(",", value.SS.ToArray()) + "]") +
            (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToArray()) + "]")
        );
    }
    Console.WriteLine("*****");
}
```

```
    }  
}
```

Example: Batch operations using the AWS SDK for .NET low-level API

Topics

- [Example: Batch write operation using the AWS SDK for .NET low-level API](#)
- [Example: Batch get operation using the AWS SDK for .NET low-level API](#)

This section provides examples of batch operations, *batch write* and *batch get*, that Amazon DynamoDB supports.

Example: Batch write operation using the AWS SDK for .NET low-level API

The following C# code example uses the `BatchWriteItem` method to perform the following put and delete operations:

- Put one item in the `Forum` table.
- Put one item and delete one item from the `Thread` table.

You can specify any number of put and delete requests against one or more tables when creating your batch write request. However, DynamoDB `BatchWriteItem` limits the size of a batch write request and the number of put and delete operations in a single batch write operation. For more information, see [BatchWriteItem](#). If your request exceeds these limits, your request is rejected. If your table does not have sufficient provisioned throughput to serve this request, the unprocessed request items are returned in the response.

The following example checks the response to see if it has any unprocessed request items. If it does, it loops back and resends the `BatchWriteItem` request with unprocessed items in the request. If you followed the steps in [Creating tables and loading data for code examples in DynamoDB](#), you already have the `Forum` and `Thread` tables created. You can also create these sample tables and upload sample data programmatically. For more information, see [Creating example tables and uploading data using the AWS SDK for .NET](#).

For step-by-step instructions for testing the following sample, see [.NET code examples](#).

Example

```
using System;
```

```
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchWrite
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                TestBatchWrite();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void TestBatchWrite()
        {
            var request = new BatchWriteItemRequest
            {
                ReturnConsumedCapacity = "TOTAL",
                RequestItems = new Dictionary<string, List<WriteRequest>>
            {
                {
                    table1Name, new List<WriteRequest>
                    {
                        new WriteRequest
                        {
                            PutRequest = new PutRequest
                            {
                                Item = new Dictionary<string, AttributeValue>
                                {
                                    { "Name", new AttributeValue { S = "S3 forum" }

```

```
        } },
        { "Threads", new AttributeValue {
            N = "0"
        }}
    }
}
},
{
    table2Name, new List<WriteRequest>
{
    new WriteRequest
{
    PutRequest = new PutRequest
{
    Item = new Dictionary<string, AttributeValue>
{
        { "ForumName", new AttributeValue {
            S = "S3 forum"
        }},
        { "Subject", new AttributeValue {
            S = "My sample question"
        }},
        { "Message", new AttributeValue {
            S = "Message Text."
        }},
        { "KeywordTags", new AttributeValue {
            SS = new List<string> { "S3", "Bucket" }
        }}
    }
},
    new WriteRequest
{
    // For the operation to delete an item, if you provide a
primary key value
    // that does not exist in the table, there is no error, it
is just a no-op.
    DeleteRequest = new DeleteRequest
{
    Key = new Dictionary<string, AttributeValue>()
{
        { "ForumName", new AttributeValue {
```

```
        S = "Some partition key value"
    } },
{ "Subject", new AttributeValue {
    S = "Some sort key value"
} }
}
}
}
}
}
};

CallBatchWriteTillCompletion(request);
}

private static void CallBatchWriteTillCompletion(BatchWriteItemRequest request)
{
    BatchWriteItemResponse response;

    int callCount = 0;
    do
    {
        Console.WriteLine("Making request");
        response = client.BatchWriteItem(request);
        callCount++;

        // Check the response.

        var tableConsumedCapacities = response.ConsumedCapacity;
        var unprocessed = response.UnprocessedItems;

        Console.WriteLine("Per-table consumed capacity");
        foreach (var tableConsumedCapacity in tableConsumedCapacities)
        {
            Console.WriteLine("{0} - {1}", tableConsumedCapacity.TableName,
tableConsumedCapacity.CapacityUnits);
        }

        Console.WriteLine("Unprocessed");
        foreach (var unp in unprocessed)
        {
            Console.WriteLine("{0} - {1}", unp.Key, unp.Value.Count);
        }
    }
}
```

```
        Console.WriteLine();

        // For the next iteration, the request will have unprocessed items.
        request.RequestItems = unprocessed;
    } while (response.UnprocessedItems.Count > 0);

    Console.WriteLine("Total # of batch write API calls made: {0}", callCount);
}
}
```

Example: Batch get operation using the AWS SDK for .NET low-level API

The following C# code example uses the BatchGetItem method to retrieve multiple items from the Forum and the Thread tables in Amazon DynamoDB. The BatchGetItemRequest specifies the table names and a list of primary keys for each table. The example processes the response by printing the items retrieved.

If you followed the steps in [Creating tables and loading data for code examples in DynamoDB](#), you already have these tables created with sample data. You can also create these sample tables and upload sample data programmatically. For more information, see [Creating example tables and uploading data using the AWS SDK for .NET](#).

For step-by-step instructions for testing the following sample, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchGet
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
```

```
try
{
    RetrieveMultipleItemsBatchGet();

    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}
catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
catch (Exception e) { Console.WriteLine(e.Message); }
}

private static void RetrieveMultipleItemsBatchGet()
{
    var request = new BatchGetItemRequest
    {
        RequestItems = new Dictionary<string, KeysAndAttributes>()
    }

    { tableName,
        new KeysAndAttributes
        {
            Keys = new List<Dictionary<string, AttributeValue>>()
            {
                new Dictionary<string, AttributeValue>()
                {
                    { "Name", new AttributeValue {
                        S = "Amazon DynamoDB"
                    } },
                    { "Name", new AttributeValue {
                        S = "Amazon S3"
                    } }
                }
            }
        }
    },
    {
        tableName,
        new KeysAndAttributes
        {
            Keys = new List<Dictionary<string, AttributeValue>>()
            {
                new Dictionary<string, AttributeValue>()
                {

```

```
        { "ForumName", new AttributeValue {
            S = "Amazon DynamoDB"
        } },
        { "Subject", new AttributeValue {
            S = "DynamoDB Thread 1"
        } }
    },
    new Dictionary<string, AttributeValue>()
{
    { "ForumName", new AttributeValue {
        S = "Amazon DynamoDB"
    } },
    { "Subject", new AttributeValue {
        S = "DynamoDB Thread 2"
    } }
},
new Dictionary<string, AttributeValue>()
{
    { "ForumName", new AttributeValue {
        S = "Amazon S3"
    } },
    { "Subject", new AttributeValue {
        S = "S3 Thread 1"
    } }
}
}
}
}
};

BatchGetItemResponse response;
do
{
    Console.WriteLine("Making request");
    response = client.BatchGetItem(request);

    // Check the response.
    var responses = response.Responses; // Attribute list in the response.

    foreach (var tableResponse in responses)
    {
        var tableResults = tableResponse.Value;
```

```
        Console.WriteLine("Items retrieved from table {0}",  
tableResponse.Key);  
        foreach (var item1 in tableResults)  
{  
            PrintItem(item1);  
        }  
    }  
  
    // Any unprocessed keys? could happen if you exceed  
ProvisionedThroughput or some other error.  
    Dictionary<string, KeysAndAttributes> unprocessedKeys =  
response.UnprocessedKeys;  
    foreach (var unprocessedTableKeys in unprocessedKeys)  
{  
        // Print table name.  
        Console.WriteLine(unprocessedTableKeys.Key);  
        // Print unprocessed primary keys.  
        foreach (var key in unprocessedTableKeys.Value.Keys)  
{  
            PrintItem(key);  
        }  
    }  
  
    request.RequestItems = unprocessedKeys;  
} while (response.UnprocessedKeys.Count > 0);  
}  
  
private static void PrintItem(Dictionary<string, AttributeValue> attributeList)  
{  
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)  
{  
        string attributeName = kvp.Key;  
        AttributeValue value = kvp.Value;  
  
        Console.WriteLine(  
            attributeName + " " +  
            (value.S == null ? "" : "S=[" + value.S + "]") +  
            (value.N == null ? "" : "N=[" + value.N + "]") +  
            (value.SS == null ? "" : "SS=[" + string.Join(",",  
value.SS.ToArray()) + "]") +  
            (value.NS == null ? "" : "NS=[" + string.Join(",",  
value.NS.ToArray()) + "]")  
        );  
    }  
}
```

```
        Console.WriteLine("*****");
    }
}
}
```

Example: Handling binary type attributes using the AWS SDK for .NET low-level API

The following C# code example illustrates the handling of binary type attributes. The example adds an item to the Reply table. The item includes a binary type attribute (ExtendedMessage) that stores compressed data. The example then retrieves the item and prints all the attribute values. For illustration, the example uses the GZipStream class to compress a sample stream and assigns it to the ExtendedMessage attribute, and decompresses it when printing the attribute value.

If you followed the steps in [Creating tables and loading data for code examples in DynamoDB](#), you already have the Reply table created. You can also create these sample tables programmatically. For more information, see [Creating example tables and uploading data using the AWS SDK for .NET](#).

For step-by-step instructions for testing the following example, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelItemBinaryExample
    {
        private static string tableName = "Reply";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            // Reply table primary key.
            string replyIdPartitionKey = "Amazon DynamoDB#DynamoDB Thread 1";
            string replyDateTimeSortKey = Convert.ToString(DateTime.UtcNow);
```

```
try
{
    CreateItem(replyIdPartitionKey, replyDateTimeSortKey);
    RetrieveItem(replyIdPartitionKey, replyDateTimeSortKey);
    // Delete item.
    DeleteItem(replyIdPartitionKey, replyDateTimeSortKey);
    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}
catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
catch (Exception e) { Console.WriteLine(e.Message); }

private static void CreateItem(string partitionKey, string sortKey)
{
    MemoryStream compressedMessage = ToGzipMemoryStream("Some long extended
message to compress.");
    var request = new PutItemRequest
    {
        TableName = tableName,
        Item = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            S = partitionKey
        }},
        { "ReplyDateTime", new AttributeValue {
            S = sortKey
        }},
        { "Subject", new AttributeValue {
            S = "Binary type "
        }},
        { "Message", new AttributeValue {
            S = "Some message about the binary type"
        }},
        { "ExtendedMessage", new AttributeValue {
            B = compressedMessage
        }}
    }
    };
    client.PutItem(request);
}

private static void RetrieveItem(string partitionKey, string sortKey)
```

```
{  
    var request = new GetItemRequest  
    {  
        TableName = tableName,  
        Key = new Dictionary<string, AttributeValue>()  
    {  
        { "Id", new AttributeValue {  
            S = partitionKey  
        } },  
        { "ReplyDateTime", new AttributeValue {  
            S = sortKey  
        } }  
    },  
    ConsistentRead = true  
};  
var response = client.GetItem(request);  
  
// Check the response.  
var attributeList = response.Item; // attribute list in the response.  
Console.WriteLine("\nPrinting item after retrieving it .....");  
  
PrintItem(attributeList);  
}  
  
private static void DeleteItem(string partitionKey, string sortKey)  
{  
    var request = new DeleteItemRequest  
    {  
        TableName = tableName,  
        Key = new Dictionary<string, AttributeValue>()  
    {  
        { "Id", new AttributeValue {  
            S = partitionKey  
        } },  
        { "ReplyDateTime", new AttributeValue {  
            S = sortKey  
        } }  
    }  
};  
var response = client.DeleteItem(request);  
}  
  
private static void PrintItem(Dictionary<string, AttributeValue> attributeList)  
{
```

```
foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
{
    string attributeName = kvp.Key;
    AttributeValue value = kvp.Value;

    Console.WriteLine(
        attributeName + " " +
        (value.S == null ? "" : "S=[" + value.S + "]") +
        (value.N == null ? "" : "N=[" + value.N + "]") +
        (value.SS == null ? "" : "SS=[" + string.Join(",", value.SS.ToArray()) + "]") +
        (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToArray()) + "]") +
        (value.B == null ? "" : "B=[" + FromGzipMemoryStream(value.B) + "]")
    );
}

Console.WriteLine("*****");
}

private static MemoryStream ToGzipMemoryStream(string value)
{
    MemoryStream output = new MemoryStream();
    using (GZipStream zipStream = new GZipStream(output,
CompressionMode.Compress, true))
        using (StreamWriter writer = new StreamWriter(zipStream))
    {
        writer.Write(value);
    }
    return output;
}

private static string FromGzipMemoryStream(MemoryStream stream)
{
    using (GZipStream zipStream = new GZipStream(stream,
CompressionMode.Decompress))
        using (StreamReader reader = new StreamReader(zipStream))
    {
        return reader.ReadToEnd();
    }
}
}
```

Item collections - how to model one-to-many relationships in DynamoDB

In DynamoDB, an *item collection* is a group of items that share the same partition key value, which means the items are related. Item collections are the primary mechanism to model one-to-many relationships in DynamoDB. Item collections can only exist on tables or indexes configured to use a [composite primary key](#).

 **Note**

Item collections can exist either in a base table or a secondary index. For more information specifically about how item collections interact with indexes, see [Item collections in Local Secondary Indexes](#).

Consider the following table showing three different users and their in-game inventories:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
account1234	inventory::armor	data {"armor": [{"name": "Pauldron of the Paladin", "type": "chest", "gear score": 545}, {"name": "Greaves of the Ranger", "type": "sword", "gear score": 382}]}		
	inventory::weapons	data {"weapons": [{"name": "Sword of the Ancients", "type": "sword", "gear score": 320}]}		
	login-data	pw d1e8a70b5ccab1dc2f56bbf7e99f064a660c08e361a35751b9c483c88943d082	state	last-login Active 1649276737
account1387	info	data {"email": "bot123@gmail.com"}		
	inventory::armor	data {"armor": [{"name": "Pauldron of the Paladin", "type": "chest", "gear score": 545}, {"name": "Greaves of the Ranger", "type": "sword", "gear score": 382}]}		
	login-data	pw k2g8jk0m5ppab1dc2f56bbf7e99f064a660c08e361a35751b9c464r23943l082	state	last-login Banned 1649456737
account1138	info	data {"email": "luh-3417@gmail.com" }		
	login-data	pw 88A41A9A62B11CCC8C120861928765A3EA41DEB9EAFE261D90F619473B89A2D4	state	last-login Active 14275516966

For some items in each collection, the sort key is a concatenation made up of information used to group data, such as `inventory::armor`, `inventory::weapon` or `info`. Each item collection can have a different combination of these attributes as the sort key. User account1234 has an `inventory::weapons` item, while user account1387 does not (because they have not found any yet). User account1138 only uses two items for their sort key (since they have no inventory yet) while the other users use three.

DynamoDB lets you selectively retrieve items from these item collections to do the following:

- Retrieve all items from a particular user
- Retrieve only one item from a particular user
- Retrieve all the items of a specific type belonging to a particular user

Speed up queries by organizing your data with item collections

In this example, each of the items in these three item collections represents a player and the data model we have chosen, based off the game's and player's access patterns. What data does the game need? When does it need it? How frequently does it need it? What's the cost of doing it this way? These data modeling decisions were made based off the answers to these questions.

In this game, there is a different page presented to the player for their inventory for weapons and another page for armor. When the player opens their inventory, weapons are shown first because we want that page to load extremely fast, while subsequent inventory pages can load after that. Since each of these item types can be quite large as the player acquires more in-game items, we decided that each inventory page would be its own item in the player's item collection in the database.

The following section talks more about how you can interact with item collections through the [Query](#) operation.

Topics

- [Query operations in DynamoDB](#)

Query operations in DynamoDB

You can use the [Query API](#) operation in Amazon DynamoDB to find items based on primary key values.

You must provide the name of the partition key attribute and a single value for that attribute. [Query](#) returns all items with that partition key value. Optionally, you can provide a sort key attribute and use a comparison operator to refine the search results.

For more information on how to use [Query](#), such as the request syntax, response parameters, and additional examples, see [Query](#) in the *Amazon DynamoDB API Reference*.

Topics

- [Key condition expressions for the Query operation](#)
- [Filter expressions for the Query operation](#)
- [Paginating table query results](#)
- [Other aspects of working with the Query operation](#)
- [Querying tables and indexes: Java](#)
- [Querying tables and indexes: .NET](#)

Key condition expressions for the Query operation

To specify the search criteria, you use a *key condition expression*—a string that determines the items to be read from the table or index.

You must specify the partition key name and value as an equality condition. You cannot use a non-key attribute in a key condition expression.

You can optionally provide a second condition for the sort key (if present). The sort key condition must use one of the following comparison operators:

- $a = b$ — true if the attribute a is equal to the value b
- $a < b$ — true if a is less than b
- $a \leq b$ — true if a is less than or equal to b
- $a > b$ — true if a is greater than b
- $a \geq b$ — true if a is greater than or equal to b
- $a \text{ BETWEEN } b \text{ AND } c$ — true if a is greater than or equal to b , and less than or equal to c .

The following function is also supported:

- `begins_with (a, substr)`— true if the value of attribute a begins with a particular substring.

The following AWS Command Line Interface (AWS CLI) examples demonstrate the use of key condition expressions. These expressions use placeholders (such as `:name` and `:sub`) instead of actual values. For more information, see [Expression attribute names in DynamoDB](#) and [Expression attribute values](#).

Example

Query the Thread table for a particular ForumName (partition key). All of the items with that ForumName value are read by the query because the sort key (Subject) is not included in KeyConditionExpression.

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :name" \
--expression-attribute-values '{":name":{"S":"Amazon DynamoDB"}}'
```

Example

Query the Thread table for a particular ForumName (partition key), but this time return only the items with a given Subject (sort key).

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :name and Subject = :sub" \
--expression-attribute-values file://values.json
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{
  ":name":{"S":"Amazon DynamoDB"},
  ":sub":{"S":"DynamoDB Thread 1"}
}
```

Example

Query the Reply table for a particular Id (partition key), but return only those items whose ReplyDateTime (sort key) begins with certain characters.

```
aws dynamodb query \
--table-name Reply \
--key-condition-expression "Id = :id and begins_with(ReplyDateTime, :dt)" \
--expression-attribute-values file://values.json
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{
```

```
:id":{"S":"Amazon DynamoDB#DynamoDB Thread 1"},  
":dt":{"S":"2015-09"}  
}
```

You can use any attribute name in a key condition expression, provided that the first character is a-z or A-Z and the rest of the characters (starting from the second character, if present) are a-z, A-Z, or 0-9. In addition, the attribute name must not be a DynamoDB reserved word. (For a complete list of these, see [Reserved words in DynamoDB](#).) If an attribute name does not meet these requirements, you must define an expression attribute name as a placeholder. For more information, see [Expression attribute names in DynamoDB](#).

For items with a given partition key value, DynamoDB stores these items close together, in sorted order by sort key value. In a Query operation, DynamoDB retrieves the items in sorted order, and then processes the items using KeyConditionExpression and any FilterExpression that might be present. Only then are the Query results sent back to the client.

A Query operation always returns a result set. If no matching items are found, the result set is empty.

Query results are always sorted by the sort key value. If the data type of the sort key is Number, the results are returned in numeric order. Otherwise, the results are returned in order of UTF-8 bytes. By default, the sort order is ascending. To reverse the order, set the ScanIndexForward parameter to false.

A single Query operation can retrieve a maximum of 1 MB of data. This limit applies before any FilterExpression or ProjectionExpression is applied to the results. If LastEvaluatedKey is present in the response and is non-null, you must paginate the result set (see [Paginating table query results](#)).

Filter expressions for the Query operation

If you need to further refine the Query results, you can optionally provide a filter expression. A *filter expression* determines which items within the Query results should be returned to you. All of the other results are discarded.

A filter expression is applied after a Query finishes, but before the results are returned. Therefore, a Query consumes the same amount of read capacity, regardless of whether a filter expression is present.

A Query operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated.

A filter expression cannot contain partition key or sort key attributes. You need to specify those attributes in the key condition expression, not the filter expression.

The syntax for a filter expression is similar to that of a key condition expression. Filter expressions can use the same comparators, functions, and logical operators as a key condition expression. In addition, filter expressions can use the not-equals operator (\neq), the OR operator, the CONTAINS operator, the IN operator, the BEGINS_WITH operator, the BETWEEN operator, the EXISTS operator, and the SIZE operator. For more information, see [Key condition expressions for the Query operation](#) and [Syntax for filter and condition expressions](#).

Example

The following AWS CLI example queries the Thread table for a particular ForumName (partition key) and Subject (sort key). Of the items that are found, only the most popular discussion threads are returned—in other words, only those threads with more than a certain number of Views.

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :fn and Subject = :sub" \
--filter-expression "#v >= :num" \
--expression-attribute-names '{"#v": "Views"}' \
--expression-attribute-values file://values.json
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{
  ":fn": {"S": "Amazon DynamoDB"},
  ":sub": {"S": "DynamoDB Thread 1"},
  ":num": {"N": "3"}
}
```

Note that Views is a reserved word in DynamoDB (see [Reserved words in DynamoDB](#)), so this example uses #v as a placeholder. For more information, see [Expression attribute names in DynamoDB](#).

Note

A filter expression removes items from the Query result set. If possible, avoid using Query where you expect to retrieve a large number of items but also need to discard most of those items.

Paginating table query results

DynamoDB *paginates* the results from Query operations. With pagination, the Query results are divided into "pages" of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on.

A single Query only returns a result set that fits within the 1 MB size limit. To determine whether there are more results, and to retrieve them one page at a time, applications should do the following:

1. Examine the low-level Query result:

- If the result contains a LastEvaluatedKey element and it's non-null, proceed to step 2.
- If there is *not* a LastEvaluatedKey in the result, there are no more items to be retrieved.

2. Construct a new Query request, with the same parameters as the previous one. However, this time, take the LastEvaluatedKey value from step 1 and use it as the ExclusiveStartKey parameter in the new Query request.

3. Run the new Query request.

4. Go to step 1.

In other words, the LastEvaluatedKey from a Query response should be used as the ExclusiveStartKey for the next Query request. If there is not a LastEvaluatedKey element in a Query response, then you have retrieved the final page of results. If LastEvaluatedKey is not empty, it does not necessarily mean that there is more data in the result set. The only way to know when you have reached the end of the result set is when LastEvaluatedKey is empty.

You can use the AWS CLI to view this behavior. The AWS CLI sends low-level Query requests to DynamoDB repeatedly, until LastEvaluatedKey is no longer present in the results. Consider the following AWS CLI example that retrieves movie titles from a particular year.

```
aws dynamodb query --table-name Movies \
    --projection-expression "title" \
    --key-condition-expression "#y = :yyyy" \
    --expression-attribute-names '{"#y":"year"}' \
    --expression-attribute-values '{":yyyy":{"N":"1993"}}' \
    --page-size 5 \
    --debug
```

Ordinarily, the AWS CLI handles pagination automatically. However, in this example, the AWS CLI `--page-size` parameter limits the number of items per page. The `--debug` parameter prints low-level information about requests and responses.

If you run the example, the first response from DynamoDB looks similar to the following.

```
2017-07-07 11:13:15,603 - MainThread - botocore.parsers - DEBUG - Response body:  
b'{"Count":5,"Items":[{"title":{"S":"A Bronx Tale"}},  
 {"title":{"S":"A Perfect World"}}, {"title":{"S":"Addams Family Values"}},  
 {"title":{"S":"Alive"}}, {"title":{"S":"Benny & Joon"}]},  
 "LastEvaluatedKey":{"year":{"N":"1993"}, "title":{"S":"Benny & Joon"}},  
 "ScannedCount":5}'
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI then issues another `Query` request to DynamoDB. This request and response pattern continues, until the final response.

```
2017-07-07 11:13:16,291 - MainThread - botocore.parsers - DEBUG - Response body:  
b'{"Count":1,"Items":[{"title":{"S":"What's Eating Gilbert  
Grape"}]}, "ScannedCount":1}'
```

The absence of `LastEvaluatedKey` indicates that there are no more items to retrieve.

Note

The AWS SDKs handle the low-level DynamoDB responses (including the presence or absence of `LastEvaluatedKey`) and provide various abstractions for paginating `Query` results. For example, the SDK for Java document interface provides `java.util.Iterator` support so that you can walk through the results one at a time.

For code examples in various programming languages, see the [Amazon DynamoDB Getting Started Guide](#) and the AWS SDK documentation for your language.

You can also reduce page size by limiting the number of items in the result set, with the `Limit` parameter of the `Query` operation.

For more information about querying with DynamoDB, see [Query operations in DynamoDB](#).

Other aspects of working with the Query operation

Limiting the number of items in the result set

With the Query operation, you can limit the number of items that it reads. To do this, set the Limit parameter to the maximum number of items that you want.

For example, suppose that you Query a table, with a Limit value of 6, and without a filter expression. The Query result contains the first six items from the table that match the key condition expression from the request.

Now suppose that you add a filter expression to the Query. In this case, DynamoDB reads up to six items, and then returns only those that match the filter expression. The final Query result contains six items or fewer, even if more items would have matched the filter expression if DynamoDB had kept reading more items.

Counting the items in the results

In addition to the items that match your criteria, the Query response contains the following elements:

- ScannedCount — The number of items that matched the key condition expression *before* a filter expression (if present) was applied.
- Count — The number of items that remain *after* a filter expression (if present) was applied.

Note

If you don't use a filter expression, ScannedCount and Count have the same value.

If the size of the Query result set is larger than 1 MB, ScannedCount and Count represent only a partial count of the total items. You need to perform multiple Query operations to retrieve all the results (see [Paginating table query results](#)).

Each Query response contains the ScannedCount and Count for the items that were processed by that particular Query request. To obtain grand totals for all of the Query requests, you could keep a running tally of both ScannedCount and Count.

Capacity units consumed by query

You can Query any table or secondary index, as long as you provide the name of the partition key attribute and a single value for that attribute. Query returns all items with that partition key value. Optionally, you can provide a sort key attribute and use a comparison operator to refine the search results. Query API operations consume read capacity units, as follows.

If you Query a...	DynamoDB consumes read capacity units from...
Table	The table's provisioned read capacity.
Global secondary index	The index's provisioned read capacity.
Local secondary index	The base table's provisioned read capacity.

By default, a Query operation does not return any data on how much read capacity it consumes. However, you can specify the `ReturnConsumedCapacity` parameter in a Query request to obtain this information. The following are the valid settings for `ReturnConsumedCapacity`:

- `NONE` — No consumed capacity data is returned. (This is the default.)
- `TOTAL` — The response includes the aggregate number of read capacity units consumed.
- `INDEXES` — The response shows the aggregate number of read capacity units consumed, together with the consumed capacity for each table and index that was accessed.

DynamoDB calculates the number of read capacity units consumed based on the number of items and the size of those items, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed is the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number is also the same whether or not you use a filter expression. Query consumes a minimum read capacity unit to perform one strongly consistent read per second, or two eventually consistent reads per second for an item up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB needs additional read request units. Empty tables and very large tables which have a sparse amount of partition keys might see some additional RCUs charged beyond the amount of data queried. This covers the cost of serving the Query request, even if no data exists.

Read consistency for query

A Query operation performs eventually consistent reads, by default. This means that the Query results might not reflect changes due to recently completed PutItem or UpdateItem operations. For more information, see [Read consistency](#).

If you require strongly consistent reads, set the ConsistentRead parameter to true in the Query request.

Querying tables and indexes: Java

The Query operation enables you to query a table or a secondary index in Amazon DynamoDB. You must provide a partition key value and an equality condition. If the table or index has a sort key, you can refine the results by providing a sort key value and a condition.

Note

The AWS SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

The following are the steps to retrieve an item using the AWS SDK for Java Document API.

1. Create an instance of the DynamoDB class.
2. Create an instance of the Table class to represent the table you want to work with.
3. Call the query method of the Table instance. You must specify the partition key value of the items that you want to retrieve, along with any optional query parameters.

The response includes an ItemCollection object that provides all items returned by the query.

The following Java code example demonstrates the preceding tasks. The example assumes that you have a Reply table that stores replies for forum threads. For more information, see [Creating tables and loading data for code examples in DynamoDB](#).

```
Reply ( Id, ReplyDateTime, ... )
```

Each forum thread has a unique ID and can have zero or more replies. Therefore, the Id attribute of the Reply table is composed of both the forum name and forum subject. Id (partition key) and ReplyDateTime (sort key) make up the composite primary key for the table.

The following query retrieves all replies for a specific thread subject. The query requires both the table name and the Subject value.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.US_WEST_2).build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("Reply");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Id = :v_id")
    .WithValueMap(new ValueMap()
        .withString(":v_id", "Amazon DynamoDB#DynamoDB Thread 1"));

ItemCollection<QueryOutcome> items = table.query(spec);

Iterator<Item> iterator = items.iterator();
Item item = null;
while (iterator.hasNext()) {
    item = iterator.next();
    System.out.println(item.toJSONPretty());
}
```

Specifying optional parameters

The `query` method supports several optional parameters. For example, you can optionally narrow the results from the preceding query to return replies in the past two weeks by specifying a condition. The condition is called a sort key condition, because DynamoDB evaluates the query condition that you specify against the sort key of the primary key. You can specify other optional parameters to retrieve only a specific list of attributes from items in the query result.

The following Java code example retrieves forum thread replies posted in the past 15 days. The example specifies optional parameters using the following:

- A `KeyConditionExpression` to retrieve the replies from a specific discussion forum (partition key) and, within that set of items, replies that were posted within the last 15 days (sort key).
- A `FilterExpression` to return only the replies from a specific user. The filter is applied after the query is processed, but before the results are returned to the user.
- A `ValueMap` to define the actual values for the `KeyConditionExpression` placeholders.

- A `ConsistentRead` setting of `true`, to request a strongly consistent read.

This example uses a `QuerySpec` object that gives access to all of the low-level `Query` input parameters.

Example

```
Table table = dynamoDB.getTable("Reply");

long twoWeeksAgoMilli = (new Date()).getTime() - (15L*24L*60L*60L*1000L);
Date twoWeeksAgo = new Date();
twoWeeksAgo.setTime(twoWeeksAgoMilli);
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
String twoWeeksAgoStr = df.format(twoWeeksAgo);

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Id = :v_id and ReplyDateTime > :v_reply_dt_tm")
    .withFilterExpression("PostedBy = :v_posted_by")
    .WithValueMap(new ValueMap()
        .withString(":v_id", "Amazon DynamoDB#DynamoDB Thread 1")
        .withString(":v_reply_dt_tm", twoWeeksAgoStr)
        .withString(":v_posted_by", "User B"))
    .withConsistentRead(true);

ItemCollection<QueryOutcome> items = table.query(spec);

Iterator<Item> iterator = items.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next().toJSONPretty());
}
```

You can also optionally limit the number of items per page by using the `withMaxPageSize` method. When you call the `query` method, you get an `ItemCollection` that contains the resulting items. You can then step through the results, processing one page at a time, until there are no more pages.

The following Java code example modifies the query specification shown previously. This time, the query spec uses the `withMaxPageSize` method. The `Page` class provides an iterator that allows the code to process the items on each page.

Example

```
spec.withMaxPageSize(10);

ItemCollection<QueryOutcome> items = table.query(spec);

// Process each page of results
int pageNum = 0;
for (Page<Item, QueryOutcome> page : items.pages()) {

    System.out.println("\nPage: " + ++pageNum);

    // Process each item on the current page
    Iterator<Item> item = page.iterator();
    while (item.hasNext()) {
        System.out.println(item.next().toJSONPretty());
    }
}
```

Example - query using Java

The following tables store information about a collection of forums. For more information, see [Creating tables and loading data for code examples in DynamoDB](#).

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

Example

```
Forum ( Name, ... )
Thread ( ForumName, Subject, Message, LastPostedBy, LastPostDateTime, ... )
Reply ( Id, ReplyDateTime, Message, PostedBy, ... )
```

In this Java code example, you run variations of finding replies for a thread "DynamoDB Thread 1" in forum "DynamoDB".

- Find replies for a thread.
- Find replies for a thread, specifying a limit on the number of items per page of results. If the number of items in the result set exceeds the page size, you get only the first page of results. This coding pattern ensures that your code processes all the pages in the query result.
- Find replies in the last 15 days.
- Find replies in a specific date range.

The preceding two queries show how you can specify sort key conditions to narrow the query results and use other optional query parameters.

 **Note**

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

```
package com.amazonaws.codesamples.document;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.Page;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class DocumentAPIQuery {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
```

```
static DynamoDB dynamoDB = new DynamoDB(client);

static String tableName = "Reply";

public static void main(String[] args) throws Exception {

    String forumName = "Amazon DynamoDB";
    String threadSubject = "DynamoDB Thread 1";

    findRepliesForAThread(forumName, threadSubject);
    findRepliesForAThreadSpecifyOptionalLimit(forumName, threadSubject);
    findRepliesInLast15DaysWithConfig(forumName, threadSubject);
    findRepliesPostedWithinTimePeriod(forumName, threadSubject);
    findRepliesUsingAFilterExpression(forumName, threadSubject);
}

private static void findRepliesForAThread(String forumName, String threadSubject) {

    Table table = dynamoDB.getTable(tableName);

    String replyId = forumName + "#" + threadSubject;

    QuerySpec spec = new QuerySpec().withKeyConditionExpression("Id = :v_id")
        .WithValueMap(new ValueMap().withString(":v_id", replyId));

    ItemCollection<QueryOutcome> items = table.query(spec);

    System.out.println("\nfindRepliesForAThread results:");

    Iterator<Item> iterator = items.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }

}

private static void findRepliesForAThreadSpecifyOptionalLimit(String forumName,
String threadSubject) {

    Table table = dynamoDB.getTable(tableName);

    String replyId = forumName + "#" + threadSubject;

    QuerySpec spec = new QuerySpec().withKeyConditionExpression("Id = :v_id")
```

```
.withValueMap(new ValueMap().withString(":v_id",
replyId)).withMaxPageSize(1);

ItemCollection<QueryOutcome> items = table.query(spec);

System.out.println("\nfindRepliesForAThreadSpecifyOptionalLimit results:");

// Process each page of results
int pageNum = 0;
for (Page<Item, QueryOutcome> page : items.pages()) {

    System.out.println("\nPage: " + ++pageNum);

    // Process each item on the current page
    Iterator<Item> item = page.iterator();
    while (item.hasNext()) {
        System.out.println(item.next().toJSONPretty());
    }
}

private static void findRepliesInLast15DaysWithConfig(String forumName, String
threadSubject) {

    Table table = dynamoDB.getTable(tableName);

    long twoWeeksAgoMilli = (new Date()).getTime() - (15L * 24L * 60L * 60L *
1000L);
    Date twoWeeksAgo = new Date();
    twoWeeksAgo.setTime(twoWeeksAgoMilli);
    SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
    String twoWeeksAgoStr = df.format(twoWeeksAgo);

    String replyId = forumName + "#" + threadSubject;

    QuerySpec spec = new QuerySpec().withProjectionExpression("Message,
ReplyDateTime, PostedBy")
        .withKeyConditionExpression("Id = :v_id and ReplyDateTime
<= :v_reply_dt_tm")
        .withValueMap(new ValueMap().withString(":v_id",
replyId).withString(":v_reply_dt_tm", twoWeeksAgoStr));

    ItemCollection<QueryOutcome> items = table.query(spec);
```

```
System.out.println("\nfindRepliesInLast15DaysWithConfig results:");
Iterator<Item> iterator = items.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next().toJSONPretty());
}

}

private static void findRepliesPostedWithinTimePeriod(String forumName, String
threadSubject) {

    Table table = dynamoDB.getTable(tableName);

    long startDateMilli = (new Date()).getTime() - (15L * 24L * 60L * 60L * 1000L);
    long endDateMilli = (new Date()).getTime() - (5L * 24L * 60L * 60L * 1000L);
    java.text.SimpleDateFormat df = new java.text.SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");
    String startDate = df.format(startDateMilli);
    String endDate = df.format(endDateMilli);

    String replyId = forumName + "#" + threadSubject;

    QuerySpec spec = new QuerySpec().withProjectionExpression("Message,
ReplyDateTime, PostedBy")
        .withKeyConditionExpression("Id = :v_id and ReplyDateTime
between :v_start_dt and :v_end_dt")
        .withValueMap(new ValueMap().withString(":v_id",
replyId).withString(":v_start_dt", startDate)
            .withString(":v_end_dt", endDate));

    ItemCollection<QueryOutcome> items = table.query(spec);

    System.out.println("\nfindRepliesPostedWithinTimePeriod results:");
    Iterator<Item> iterator = items.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }
}

private static void findRepliesUsingAFilterExpression(String forumName, String
threadSubject) {

    Table table = dynamoDB.getTable(tableName);
```

```
String replyId = forumName + "#" + threadSubject;

QuerySpec spec = new QuerySpec().withProjectionExpression("Message,
ReplyDateTime, PostedBy")
    .withKeyConditionExpression("Id
= :v_id").withFilterExpression("PostedBy = :v_postedby")
    .withValueMap(new ValueMap().withString(":v_id",
replyId).withString(":v_postedby", "User B"));

ItemCollection<QueryOutcome> items = table.query(spec);

System.out.println("\nfindRepliesUsingAFilterExpression results:");
Iterator<Item> iterator = items.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next().toJSONPretty());
}
}
```

Querying tables and indexes: .NET

The `Query` operation enables you to query a table or a secondary index in Amazon DynamoDB. You must provide a partition key value and an equality condition. If the table or index has a sort key, you can refine the results by providing a sort key value and a condition.

The following are the steps to query a table using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class and provide query operation parameters.
3. Run the `Query` method and provide the `QueryRequest` object that you created in the preceding step.

The response includes the `QueryResult` object that provides all items returned by the query.

The following C# code example demonstrates the preceding tasks. The code assumes that you have a `Reply` table that stores replies for forum threads. For more information, see [Creating tables and loading data for code examples in DynamoDB](#).

Example

```
Reply Id, ReplyDateTime, ... )
```

Each forum thread has a unique ID and can have zero or more replies. Therefore, the primary key is composed of both the Id (partition key) and ReplyDateTime (sort key).

The following query retrieves all replies for a specific thread subject. The query requires both the table name and the Subject value.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

var request = new QueryRequest
{
    TableName = "Reply",
    KeyConditionExpression = "Id = :v_Id",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {":v_Id", new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 1" }}}
};

var response = client.Query(request);

foreach (Dictionary<string, AttributeValue> item in response.Items)
{
    // Process the result.
    PrintItem(item);
}
```

Specifying optional parameters

The `Query` method supports several optional parameters. For example, you can optionally narrow the query result in the preceding query to return replies in the past two weeks by specifying a condition. The condition is called a *sort key condition*, because DynamoDB evaluates the query condition that you specify against the sort key of the primary key. You can specify other optional parameters to retrieve only a specific list of attributes from items in the query result. For more information, see [Query](#).

The following C# code example retrieves forum thread replies posted in the past 15 days. The example specifies the following optional parameters:

- A KeyConditionExpression to retrieve only the replies in the past 15 days.
- A ProjectionExpression parameter to specify a list of attributes to retrieve for items in the query result.
- A ConsistentRead parameter to perform a strongly consistent read.

Example

```
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
string twoWeeksAgoString = twoWeeksAgoDate.ToString(AWSSDKUtils.ISO8601DateFormat);

var request = new QueryRequest
{
    TableName = "Reply",
    KeyConditionExpression = "Id = :v_Id and ReplyDateTime > :v_twoWeeksAgo",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {":v_Id", new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 2" }},
        {":v_twoWeeksAgo", new AttributeValue { S = twoWeeksAgoString }}
    },
    ProjectionExpression = "Subject, ReplyDateTime, PostedBy",
    ConsistentRead = true
};

var response = client.Query(request);

foreach (Dictionary<string, AttributeValue> item in response.Items)
{
    // Process the result.
    PrintItem(item);
}
```

You can also optionally limit the page size, or the number of items per page, by adding the optional Limit parameter. Each time you run the Query method, you get one page of results that has the specified number of items. To fetch the next page, you run the Query method again by providing the primary key value of the last item in the previous page so that the method can return the next set of items. You provide this information in the request by setting the ExclusiveStartKey property. Initially, this property can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following C# example queries the Reply table. In the request, it specifies the `Limit` and `ExclusiveStartKey` optional parameters. The do/while loop continues to scan one page at time until the `LastEvaluatedKey` returns a null value.

Example

```
Dictionary<string,AttributeValue> lastKeyEvaluated = null;

do
{
    var request = new QueryRequest
    {
        TableName = "Reply",
        KeyConditionExpression = "Id = :v_Id",
        ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
            {"":v_Id", new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 2" }},
        },

        // Optional parameters.
        Limit = 1,
        ExclusiveStartKey = lastKeyEvaluated
    };

    var response = client.Query(request);

    // Process the query result.
    foreach (Dictionary<string, AttributeValue> item in response.Items)
    {
        PrintItem(item);
    }

    lastKeyEvaluated = response.LastEvaluatedKey;
} while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);
```

Example - querying using the AWS SDK for .NET

The following tables store information about a collection of forums. For more information, see [Creating tables and loading data for code examples in DynamoDB](#).

Example

```
Forum ( Name, ... )
Thread ( ForumName, Subject, Message, LastPostedBy, LastPostDateTime, ... )
Reply ( Id, ReplyDateTime, Message, PostedBy, ... )
```

In this example, you run variations of "Find replies for a thread "DynamoDB Thread 1" in forum "DynamoDB".

- Find replies for a thread.
- Find replies for a thread. Specify the `Limit` query parameter to set page size.

This function illustrates the use of pagination to process multipage result. DynamoDB has a page size limit and if your result exceeds the page size, you get only the first page of results. This coding pattern ensures your code processes all the pages in the query result.

- Find replies in the last 15 days.
- Find replies in a specific date range.

The preceding two queries show how you can specify sort key conditions to narrow query results and use other optional query parameters.

For step-by-step instructions for testing the following example, see [.NET code examples](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.Util;

namespace com.amazonaws.codesamples
{
    class LowLevelQuery
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
```

```
// Query a specific forum and thread.  
string forumName = "Amazon DynamoDB";  
string threadSubject = "DynamoDB Thread 1";  
  
FindRepliesForAThread(forumName, threadSubject);  
FindRepliesForAThreadSpecifyOptionalLimit(forumName, threadSubject);  
FindRepliesInLast15DaysWithConfig(forumName, threadSubject);  
FindRepliesPostedWithinTimePeriod(forumName, threadSubject);  
  
Console.WriteLine("Example complete. To continue, press Enter");  
Console.ReadLine();  
}  
catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message);  
Console.ReadLine(); }  
catch (AmazonServiceException e) { Console.WriteLine(e.Message);  
Console.ReadLine(); }  
catch (Exception e) { Console.WriteLine(e.Message); Console.ReadLine(); }  
}  
  
private static void FindRepliesPostedWithinTimePeriod(string forumName, string  
threadSubject)  
{  
    Console.WriteLine("**** Executing FindRepliesPostedWithinTimePeriod() ****");  
    string replyId = forumName + "#" + threadSubject;  
    // You must provide date value based on your test data.  
    DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(21);  
    string start = startDate.ToString(AWSSDKUtils.ISO8601DateFormat);  
  
    // You provide date value based on your test data.  
    DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(5);  
    string end = endDate.ToString(AWSSDKUtils.ISO8601DateFormat);  
  
    var request = new QueryRequest  
    {  
        TableName = "Reply",  
        ReturnConsumedCapacity = "TOTAL",  
        KeyConditionExpression = "Id = :v_replyId and ReplyDateTime  
between :v_start and :v_end",  
        ExpressionAttributeValues = new Dictionary<string, AttributeValue> {  
            {"":v_replyId", new AttributeValue {  
                S = replyId  
            }},  
            {"":v_start", new AttributeValue {  
                S = start  
            }}  
    };  
}
```

```
        }},  
        {"<:v_end", new AttributeValue {  
            S = end  
        }}  
    }  
};  
  
var response = client.Query(request);  
  
Console.WriteLine("\nNo. of reads used (by query in  
FindRepliesPostedWithinTimePeriod) {0}",  
    response.ConsumedCapacity.CapacityUnits);  
foreach (Dictionary<string, AttributeValue> item  
    in response.Items)  
{  
    PrintItem(item);  
}  
Console.WriteLine("To continue, press Enter");  
Console.ReadLine();  
}  
  
private static void FindRepliesInLast15DaysWithConfig(string forumName, string  
threadSubject)  
{  
    Console.WriteLine("**** Executing FindRepliesInLast15DaysWithConfig() ****");  
    string replyId = forumName + "#" + threadSubject;  
  
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);  
    string twoWeeksAgoString =  
        twoWeeksAgoDate.ToString(AWSSDKUtils.IS08601DateFormat);  
  
    var request = new QueryRequest  
{  
        TableName = "Reply",  
        ReturnConsumedCapacity = "TOTAL",  
        KeyConditionExpression = "Id = :v_replyId and ReplyDateTime  
> :v_interval",  
        ExpressionAttributeValues = new Dictionary<string, AttributeValue> {  
            {"<:v_replyId", new AttributeValue {  
                S = replyId  
            }},  
            {"<:v_interval", new AttributeValue {  
                S = twoWeeksAgoString  
            }}  
    };  
}
```

```
    },  
  
    // Optional parameter.  
    ProjectionExpression = "Id, ReplyDateTime, PostedBy",  
    // Optional parameter.  
    ConsistentRead = true  
};  
  
var response = client.Query(request);  
  
Console.WriteLine("No. of reads used (by query in  
FindRepliesInLast15DaysWithConfig) {0}",  
    response.ConsumedCapacity.CapacityUnits);  
foreach (Dictionary<string, AttributeValue> item  
    in response.Items)  
{  
    PrintItem(item);  
}  
Console.WriteLine("To continue, press Enter");  
Console.ReadLine();  
}  
  
private static void FindRepliesForAThreadSpecifyOptionalLimit(string forumName,  
string threadSubject)  
{  
    Console.WriteLine("**** Executing  
FindRepliesForAThreadSpecifyOptionalLimit() ****");  
    string replyId = forumName + "#" + threadSubject;  
  
    Dictionary<string, AttributeValue> lastKeyEvaluated = null;  
    do  
    {  
        var request = new QueryRequest  
        {  
            TableName = "Reply",  
            ReturnConsumedCapacity = "TOTAL",  
            KeyConditionExpression = "Id = :v_replyId",  
            ExpressionAttributeValues = new Dictionary<string, AttributeValue>  
        {  
            {":v_replyId", new AttributeValue {  
                S = replyId  
            }}  
        },  
    },
```

```
        Limit = 2, // The Reply table has only a few sample items. So the
page size is smaller.
        ExclusiveStartKey = lastKeyEvaluated
    };

    var response = client.Query(request);

    Console.WriteLine("No. of reads used (by query in
FindRepliesForAThreadSpecifyLimit) {0}\n",
                      response.ConsumedCapacity.CapacityUnits);
    foreach (Dictionary<string, AttributeValue> item
             in response.Items)
    {
        PrintItem(item);
    }
    lastKeyEvaluated = response.LastEvaluatedKey;
} while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);

Console.WriteLine("To continue, press Enter");

Console.ReadLine();
}

private static void FindRepliesForAThread(string forumName, string
threadSubject)
{
    Console.WriteLine("*** Executing FindRepliesForAThread() ***");
    string replyId = forumName + "#" + threadSubject;

    var request = new QueryRequest
    {
        TableName = "Reply",
        ReturnConsumedCapacity = "TOTAL",
        KeyConditionExpression = "Id = :v_replyId",
        ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
            {":v_replyId", new AttributeValue {
                S = replyId
            }}
        }
    };
}

var response = client.Query(request);
```

```
Console.WriteLine("No. of reads used (by query in FindRepliesForAThread)  
{0}\n",  
    response.ConsumedCapacity.CapacityUnits);  
foreach (Dictionary<string, AttributeValue> item in response.Items)  
{  
    PrintItem(item);  
}  
Console.WriteLine("To continue, press Enter");  
Console.ReadLine();  
}  
  
private static void PrintItem(  
    Dictionary<string, AttributeValue> attributeList)  
{  
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)  
    {  
        string attributeName = kvp.Key;  
        AttributeValue value = kvp.Value;  
  
        Console.WriteLine(  
            attributeName + " " +  
            (value.S == null ? "" : "S=[" + value.S + "]") +  
            (value.N == null ? "" : "N=[" + value.N + "]") +  
            (value.SS == null ? "" : "SS=[" + string.Join(",",  
value.SS.ToArray()) + "]") +  
            (value.NS == null ? "" : "NS=[" + string.Join(",",  
value.NS.ToArray()) + "]")  
        );  
    }  
    Console.WriteLine("*****");  
}  
}  
}
```

Working with scans in DynamoDB

A Scan operation in Amazon DynamoDB reads every item in a table or a secondary index. By default, a Scan operation returns all of the data attributes for every item in the table or index. You can use the `ProjectionExpression` parameter so that Scan only returns some of the attributes, rather than all of them.

Scan always returns a result set. If no matching items are found, the result set is empty.

A single Scan request can retrieve a maximum of 1 MB of data. Optionally, DynamoDB can apply a filter expression to this data, narrowing the results before they are returned to the user.

Topics

- [Filter expressions for scan](#)
- [Limiting the number of items in the result set](#)
- [Paginating the results](#)
- [Counting the items in the results](#)
- [Capacity units consumed by scan](#)
- [Read consistency for scan](#)
- [Parallel scan](#)
- [Scanning tables and indexes: Java](#)
- [Scanning tables and indexes: .NET](#)

Filter expressions for scan

If you need to further refine the Scan results, you can optionally provide a filter expression. A *filter expression* determines which items within the Scan results should be returned to you. All of the other results are discarded.

A filter expression is applied after a Scan finishes but before the results are returned. Therefore, a Scan consumes the same amount of read capacity, regardless of whether a filter expression is present.

A Scan operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated.

With Scan, you can specify any attributes in a filter expression—including partition key and sort key attributes.

The syntax for a filter expression is identical to that of a condition expression. Filter expressions can use the same comparators, functions, and logical operators as a condition expression. See [Comparison operator and function reference](#) for more information about logical operators.

Example

The following AWS Command Line Interface (AWS CLI) example scans the Thread table and returns only the items that were last posted to by a particular user.

```
aws dynamodb scan \
    --table-name Thread \
    --filter-expression "LastPostedBy = :name" \
    --expression-attribute-values '{":name":{"S":"User A"}}'
```

Limits the number of items in the result set

The Scan operation enables you to limit the number of items that it returns in the result. To do this, set the Limit parameter to the maximum number of items that you want the Scan operation to return, prior to filter expression evaluation.

For example, suppose that you Scan a table with a Limit value of 6 and without a filter expression. The Scan result contains the first six items from the table.

Now suppose that you add a filter expression to the Scan. In this case, DynamoDB applies the filter expression to the six items that were returned, discarding those that do not match. The final Scan result contains six items or fewer, depending on the number of items that were filtered.

Paginating the results

DynamoDB *paginates* the results from Scan operations. With pagination, the Scan results are divided into "pages" of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on.

A single Scan only returns a result set that fits within the 1 MB size limit.

To determine whether there are more results and to retrieve them one page at a time, applications should do the following:

1. Examine the low-level Scan result:
 - If the result contains a LastEvaluatedKey element, proceed to step 2.
 - If there is *not* a LastEvaluatedKey in the result, then there are no more items to be retrieved.
2. Construct a new Scan request, with the same parameters as the previous one. However, this time, take the LastEvaluatedKey value from step 1 and use it as the ExclusiveStartKey parameter in the new Scan request.
3. Run the new Scan request.
4. Go to step 1.

In other words, the `LastEvaluatedKey` from a `Scan` response should be used as the `ExclusiveStartKey` for the next `Scan` request. If there is not a `LastEvaluatedKey` element in a `Scan` response, you have retrieved the final page of results. (The absence of `LastEvaluatedKey` is the only way to know that you have reached the end of the result set.)

You can use the AWS CLI to view this behavior. The AWS CLI sends low-level `Scan` requests to DynamoDB, repeatedly, until `LastEvaluatedKey` is no longer present in the results. Consider the following AWS CLI example that scans the entire `Movies` table but returns only the movies from a particular genre.

```
aws dynamodb scan \
--table-name Movies \
--projection-expression "title" \
--filter-expression 'contains(info.genres,:gen)' \
--expression-attribute-values '{":gen":{"S":"Sci-Fi"}}' \
--page-size 100 \
--debug
```

Ordinarily, the AWS CLI handles pagination automatically. However, in this example, the AWS CLI `--page-size` parameter limits the number of items per page. The `--debug` parameter prints low-level information about requests and responses.

Note

Your pagination results will also differ based on the input parameters you pass.

- Using `aws dynamodb scan --table-name Prices --max-items 1` returns a `NextToken`
- Using `aws dynamodb scan --table-name Prices --limit 1` returns a `LastEvaluatedKey`.

Also be aware that using `--starting-token` in particular requires the `NextToken` value.

If you run the example, the first response from DynamoDB looks similar to the following.

```
2017-07-07 12:19:14,389 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":7,"Items":[{"title":{"S":"Monster on the Campus"}}, {"title":{"S":"+1"}},
```

```
{"title":{"S":"100 Degrees Below Zero"}}, {"title":{"S":"About Time"}}, {"title":{"S":"After Earth"}}, {"title":{"S":"Age of Dinosaurs"}}, {"title":{"S":"Cloudy with a Chance of Meatballs 2"}]}, {"LastEvaluatedKey": {"year": {"N": "2013"}, "title": {"S": "Curse of Chucky"}}, "ScannedCount": 100}
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI then issues another Scan request to DynamoDB. This request and response pattern continues, until the final response.

```
2017-07-07 12:19:17,830 - MainThread - botocore.parsers - DEBUG - Response body:  
b'{"Count":1,"Items":[{"title":{"S":"WarGames"}}], "ScannedCount":6}'
```

The absence of `LastEvaluatedKey` indicates that there are no more items to retrieve.

Note

The AWS SDKs handle the low-level DynamoDB responses (including the presence or absence of `LastEvaluatedKey`) and provide various abstractions for paginating Scan results. For example, the SDK for Java document interface provides `java.util.Iterator` support so that you can walk through the results one at a time.

For code examples in various programming languages, see the [Amazon DynamoDB Getting Started Guide](#) and the AWS SDK documentation for your language.

Counting the items in the results

In addition to the items that match your criteria, the Scan response contains the following elements:

- `ScannedCount` — The number of items evaluated, before any `ScanFilter` is applied. A high `ScannedCount` value with few, or no, `Count` results indicates an inefficient Scan operation. If you did not use a filter in the request, `ScannedCount` is the same as `Count`.
- `Count` — The number of items that remain, *after* a filter expression (if present) was applied.

Note

If you do not use a filter expression, ScannedCount and Count have the same value.

If the size of the Scan result set is larger than 1 MB, ScannedCount and Count represent only a partial count of the total items. You need to perform multiple Scan operations to retrieve all the results (see [Paginating the results](#)).

Each Scan response contains the ScannedCount and Count for the items that were processed by that particular Scan request. To get grand totals for all of the Scan requests, you could keep a running tally of both ScannedCount and Count.

Capacity units consumed by scan

You can Scan any table or secondary index. Scan operations consume read capacity units, as follows.

If you Scan a...	DynamoDB consumes read capacity units from...
Table	The table's provisioned read capacity.
Global secondary index	The index's provisioned read capacity.
Local secondary index	The base table's provisioned read capacity.

By default, a Scan operation does not return any data on how much read capacity it consumes. However, you can specify the `ReturnConsumedCapacity` parameter in a Scan request to obtain this information. The following are the valid settings for `ReturnConsumedCapacity`:

- `NONE` — No consumed capacity data is returned. (This is the default.)
- `TOTAL` — The response includes the aggregate number of read capacity units consumed.
- `INDEXES` — The response shows the aggregate number of read capacity units consumed, together with the consumed capacity for each table and index that was accessed.

DynamoDB calculates the number of read capacity units consumed based on the number of items and the size of those items, not on the amount of data that is returned to an application.

For this reason, the number of capacity units consumed is the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number is also the same whether or not you use a filter expression. Scan consumes a minimum read capacity unit to perform one strongly consistent read per second, or two eventually consistent reads per second for an item up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB needs additional read request units. Empty tables and very large tables which have a sparse amount of partition keys might see some additional RCUs charged beyond the amount of data scanned. This covers the cost of serving the Scan request, even if no data exists.

Read consistency for scan

A Scan operation performs eventually consistent reads, by default. This means that the Scan results might not reflect changes due to recently completed PutItem or UpdateItem operations. For more information, see [Read consistency](#).

If you require strongly consistent reads, as of the time that the Scan begins, set the ConsistentRead parameter to true in the Scan request. This ensures that all of the write operations that completed before the Scan began are included in the Scan response.

Setting ConsistentRead to true can be useful in table backup or replication scenarios, in conjunction with [DynamoDB Streams](#). You first use Scan with ConsistentRead set to true to obtain a consistent copy of the data in the table. During the Scan, DynamoDB Streams records any additional write activity that occurs on the table. After the Scan is complete, you can apply the write activity from the stream to the table.

 **Note**

A Scan operation with ConsistentRead set to true consumes twice as many read capacity units as compared to leaving ConsistentRead at its default value (false).

Parallel scan

By default, the Scan operation processes data sequentially. Amazon DynamoDB returns data to the application in 1 MB increments, and an application performs additional Scan operations to retrieve the next 1 MB of data.

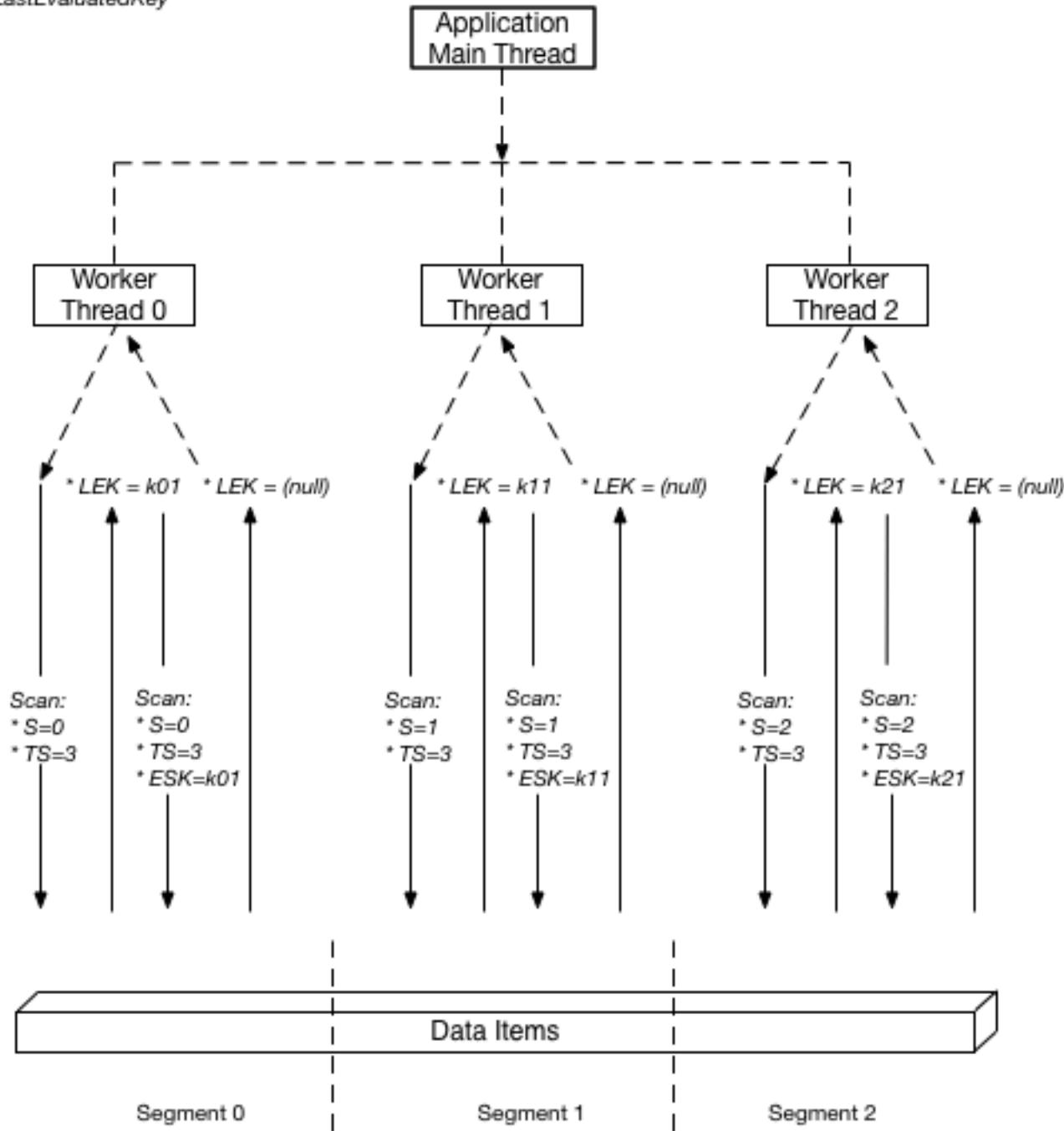
The larger the table or index being scanned, the more time the Scan takes to complete. In addition, a sequential Scan might not always be able to fully use the provisioned read throughput

capacity: Even though DynamoDB distributes a large table's data across multiple physical partitions, a Scan operation can only read one partition at a time. For this reason, the throughput of a Scan is constrained by the maximum throughput of a single partition.

To address these issues, the Scan operation can logically divide a table or secondary index into multiple *segments*, with multiple application workers scanning the segments in parallel. Each worker can be a thread (in programming languages that support multithreading) or an operating system process. To perform a parallel scan, each worker issues its own Scan request with the following parameters:

- Segment — A segment to be scanned by a particular worker. Each worker should use a different value for Segment.
- TotalSegments — The total number of segments for the parallel scan. This value must be the same as the number of workers that your application will use.

The following diagram shows how a multithreaded application performs a parallel Scan with three degrees of parallelism.

*S: Segment**TS: TotalSegments**ESK: ExclusiveStartKey**LEK: LastEvaluatedKey*

In this diagram, the application spawns three threads and assigns each thread a number. (Segments are zero-based, so the first number is always 0.) Each thread issues a Scan request, setting Segment to its designated number and setting TotalSegments to 3. Each thread scans its

designated segment, retrieving data 1 MB at a time, and returns the data to the application's main thread.

The values for Segment and TotalSegments apply to individual Scan requests, and you can use different values at any time. You might need to experiment with these values, and the number of workers you use, until your application achieves its best performance.

Note

A parallel scan with a large number of workers can easily consume all of the provisioned throughput for the table or index being scanned. It is best to avoid such scans if the table or index is also incurring heavy read or write activity from other applications.

To control the amount of data returned per request, use the Limit parameter. This can help prevent situations where one worker consumes all of the provisioned throughput, at the expense of all other workers.

Scanning tables and indexes: Java

The Scan operation reads all of the items in a table or index in Amazon DynamoDB.

The following are the steps to scan a table using the AWS SDK for Java Document API.

1. Create an instance of the AmazonDynamoDB class.
2. Create an instance of the ScanRequest class and provide scan parameter.

The only required parameter is the table name.

3. Run the scan method and provide the ScanRequest object that you created in the preceding step.

The following Reply table stores replies for forum threads.

Example

```
Reply ( Id, ReplyDateTime, Message, PostedBy )
```

The table maintains all the replies for various forum threads. Therefore, the primary key is composed of both the Id (partition key) and ReplyDateTime (sort key). The following Java code example scans the entire table. The ScanRequest instance specifies the name of the table to scan.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

ScanRequest scanRequest = new ScanRequest()
    .withTableName("Reply");

ScanResponse result = client.scan(scanRequest);
for (Map<String, AttributeValue> item : result.getItems()){
    printItem(item);
}
```

Specifying optional parameters

The scan method supports several optional parameters. For example, you can optionally use a filter expression to filter the scan result. In a filter expression, you can specify a condition and attribute names and values on which you want the condition evaluated. For more information, see [Scan](#).

The following Java example scans the ProductCatalog table to find items that are priced less than 0. The example specifies the following optional parameters:

- A filter expression to retrieve only the items priced less than 0 (error condition).
- A list of attributes to retrieve for items in the query results.

Example

```
Map<String, AttributeValue> expressionAttributeValues =
    new HashMap<String, AttributeValue>();
expressionAttributeValues.put(":val", new AttributeValue().withN("0"));

ScanRequest scanRequest = new ScanRequest()
    .withTableName("ProductCatalog")
    .withFilterExpression("Price < :val")
    .withProjectionExpression("Id")
    .withExpressionAttributeValues(expressionAttributeValues);
```

```
ScanResponse result = client.scan(scanRequest);
for (Map<String, AttributeValue> item : result.getItems()) {
    printItem(item);
}
```

You can also optionally limit the page size, or the number of items per page, by using the `withLimit` method of the scan request. Each time you run the `scan` method, you get one page of results that has the specified number of items. To fetch the next page, you run the `scan` method again by providing the primary key value of the last item in the previous page so that the `scan` method can return the next set of items. You provide this information in the request by using the `withExclusiveStartKey` method. Initially, the parameter of this method can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following Java code example scans the `ProductCatalog` table. In the request, the `withLimit` and `withExclusiveStartKey` methods are used. The `do/while` loop continues to scan one page at time until the `getLastEvaluatedKey` method of the result returns a value of null.

Example

```
Map<String, AttributeValue> lastKeyEvaluated = null;
do {
    ScanRequest scanRequest = new ScanRequest()
        .withTableName("ProductCatalog")
        .withLimit(10)
        .withExclusiveStartKey(lastKeyEvaluated);

    ScanResponse result = client.scan(scanRequest);
    for (Map<String, AttributeValue> item : result.getItems()){
        printItem(item);
    }
    lastKeyEvaluated = result.getLastEvaluatedKey();
} while (lastKeyEvaluated != null);
```

Example - scan using Java

The following Java code example provides a working sample that scans the `ProductCatalog` table to find items that are priced less than 100.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

```
package com.amazonaws.codesamples.document;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class DocumentAPIScan {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);
    static String tableName = "ProductCatalog";

    public static void main(String[] args) throws Exception {
        findProductsForPriceLessThanOneHundred();
    }
}
```

```
private static void findProductsForPriceLessThanOneHundred() {  
  
    Table table = dynamoDB.getTable(tableName);  
  
    Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();  
    expressionAttributeValues.put(":pr", 100);  
  
    ItemCollection<ScanOutcome> items = table.scan("Price < :pr", //  
FilterExpression  
        "Id, Title, ProductCategory, Price", // ProjectionExpression  
        null, // ExpressionAttributeNames - not used in this example  
        expressionAttributeValues);  
  
    System.out.println("Scan of " + tableName + " for items with a price less than  
100.");  
    Iterator<Item> iterator = items.iterator();  
    while (iterator.hasNext()) {  
        System.out.println(iterator.next().toJSONPretty());  
    }  
}  
  
}
```

Example - parallel scan using Java

The following Java code example demonstrates a parallel scan. The program deletes and re-creates a table named ParallelScanTest, and then loads the table with data. When the data load is finished, the program spawns multiple threads and issues parallel Scan requests. The program prints runtime statistics for each parallel request.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java 1.x: DynamoDBMapper](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating tables and loading data for code examples in DynamoDB](#) section.

For step-by-step instructions to run the following example, see [Java code examples](#).

```
package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.ScanSpec;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class DocumentAPIParallelScan {

    // total number of sample items
    static int scanItemCount = 300;

    // number of items each scan request should return
    static int scanItemLimit = 10;
```

```
// number of logical segments for parallel scan
static int parallelScanThreads = 16;

// table that will be used for scanning
static String parallelScanTestTableName = "ParallelScanTest";

static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
static DynamoDB dynamoDB = new DynamoDB(client);

public static void main(String[] args) throws Exception {
    try {

        // Clean up the table
        deleteTable(parallelScanTestTableName);
        createTable(parallelScanTestTableName, 10L, 5L, "Id", "N");

        // Upload sample data for scan
        uploadSampleProducts(parallelScanTestTableName, scanItemCount);

        // Scan the table using multiple threads
        parallelScan(parallelScanTestTableName, scanItemLimit,
parallelScanThreads);
    } catch (AmazonServiceException ase) {
        System.err.println(ase.getMessage());
    }
}

private static void parallelScan(String tableName, int itemLimit, int
numberOfThreads) {
    System.out.println(
        "Scanning " + tableName + " using " + numberOfThreads + " threads " +
itemLimit + " items at a time");
    ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

    // Divide DynamoDB table into logical segments
    // Create one task for scanning each segment
    // Each thread will be scanning one segment
    int totalSegments = numberOfThreads;
    for (int segment = 0; segment < totalSegments; segment++) {
        // Runnable task that will only scan one segment
        ScanSegmentTask task = new ScanSegmentTask(tableName, itemLimit,
totalSegments, segment);

        // Execute the task
    }
}
```

```
        executor.execute(task);
    }

    shutDownExecutorService(executor);
}

// Runnable task for scanning a single segment of a DynamoDB table
private static class ScanSegmentTask implements Runnable {

    // DynamoDB table to scan
    private String tableName;

    // number of items each scan request should return
    private int itemLimit;

    // Total number of segments
    // Equals to total number of threads scanning the table in parallel
    private int totalSegments;

    // Segment that will be scanned with by this task
    private int segment;

    public ScanSegmentTask(String tableName, int itemLimit, int totalSegments, int segment) {
        this.tableName = tableName;
        this.itemLimit = itemLimit;
        this.totalSegments = totalSegments;
        this.segment = segment;
    }

    @Override
    public void run() {
        System.out.println("Scanning " + tableName + " segment " + segment + " out
of " + totalSegments
                + " segments " + itemLimit + " items at a time...");
        int totalScannedItemCount = 0;

        Table table = dynamoDB.getTable(tableName);

        try {
            ScanSpec spec = new
ScanSpec().withMaxResultSize(itemLimit).withTotalSegments(totalSegments)
                    .withSegment(segment);
        }
    }
}
```

```
ItemCollection<ScanOutcome> items = table.scan(spec);
Iterator<Item> iterator = items.iterator();

Item currentItem = null;
while (iterator.hasNext()) {
    totalScannedItemCount++;
    currentItem = iterator.next();
    System.out.println(currentItem.toString());
}

} catch (Exception e) {
    System.err.println(e.getMessage());
} finally {
    System.out.println("Scanned " + totalScannedItemCount + " items from
segment " + segment + " out of "
                    + totalSegments + " of " + tableName);
}
}

private static void uploadSampleProducts(String tableName, int itemCount) {
    System.out.println("Adding " + itemCount + " sample items to " + tableName);
    for (int productIndex = 0; productIndex < itemCount; productIndex++) {
        uploadProduct(tableName, productIndex);
    }
}

private static void uploadProduct(String tableName, int productIndex) {

    Table table = dynamoDB.getTable(tableName);

    try {
        System.out.println("Processing record #" + productIndex);

        Item item = new Item().withPrimaryKey("Id", productIndex)
            .withString("Title", "Book " + productIndex + " "
Title").withString("ISBN", "111-111111111")
            .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1"))).withNumber("Price", 2)
            .withString("Dimensions", "8.5 x 11.0 x
0.5").withNumber("PageCount", 500)
            .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
        table.putItem(item);
    }
}
```

```
        } catch (Exception e) {
            System.err.println("Failed to create item " + productIndex + " in " +
tableName);
            System.err.println(e.getMessage());
        }
    }

private static void deleteTable(String tableName) {
    try {

        Table table = dynamoDB.getTable(tableName);
        table.delete();
        System.out.println("Waiting for " + tableName + " to be deleted...this may
take a while...");
        table.waitForDelete();

    } catch (Exception e) {
        System.err.println("Failed to delete table " + tableName);
        e.printStackTrace(System.err);
    }
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
String partitionKeyName, String partitionKeyType) {

    createTable(tableName, readCapacityUnits, writeCapacityUnits, partitionKeyName,
partitionKeyType, null, null);
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
String partitionKeyName, String partitionKeyType, String sortKeyName,
String sortKeyType) {

    try {
        System.out.println("Creating table " + tableName);

        List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
        keySchema.add(new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH)); // //
Partition
    }
}
```

```
// key

List<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
attributeDefinitions
    .add(new AttributeDefinition().withAttributeName(partitionKeyName)
        .withAttributeType(partitionKeyType));

if (sortKeyName != null) {
    keySchema.add(new
KeySchemaElement().withAttributeName(sortKeyName).withKeyType(KeyType.RANGE)); // Sort

    // key
    attributeDefinitions
        .add(new
AttributeDefinition().withAttributeName(sortKeyName).withAttributeType(sortKeyType));
}

Table table = dynamoDB.createTable(tableName, keySchema,
attributeDefinitions, new ProvisionedThroughput()

.withReadCapacityUnits(readCapacityUnits).withWriteCapacityUnits(writeCapacityUnits));
System.out.println("Waiting for " + tableName + " to be created...this may
take a while...");
table.waitForActive();

} catch (Exception e) {
    System.err.println("Failed to create table " + tableName);
    e.printStackTrace(System.err);
}
}

private static void shutDownExecutorService(ExecutorService executor) {
    executor.shutdown();
    try {
        if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
            executor.shutdownNow();
        }
    } catch (InterruptedException e) {
        executor.shutdownNow();

        // Preserve interrupt status
        Thread.currentThread().interrupt();
    }
}
```

```
    }  
}  
}
```

Scanning tables and indexes: .NET

The Scan operation reads all of the items in a table or index in Amazon DynamoDB.

The following are the steps to scan a table using the AWS SDK for .NET low-level API:

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `ScanRequest` class and provide scan operation parameters.
The only required parameter is the table name.
3. Run the `Scan` method and provide the `ScanRequest` object that you created in the preceding step.

The following Reply table stores replies for forum threads.

Example

```
>Reply ( <emphasis role="underline">Id</emphasis>, <emphasis  
role="underline">ReplyDateTime</emphasis>, Message, PostedBy )
```

The table maintains all the replies for various forum threads. Therefore, the primary key is composed of both the `Id` (partition key) and `ReplyDateTime` (sort key). The following C# code example scans the entire table. The `ScanRequest` instance specifies the name of the table to scan.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
  
var request = new ScanRequest  
{  
    TableName = "Reply",  
};  
  
var response = client.Scan(request);  
var result = response.ScanResult;
```

```
foreach (Dictionary<string, AttributeValue> item in response.ScanResult.Items)
{
    // Process the result.
    PrintItem(item);
}
```

Specifying optional parameters

The Scan method supports several optional parameters. For example, you can optionally use a scan filter to filter the scan result. In a scan filter, you can specify a condition and an attribute name on which you want the condition evaluated. For more information, see [Scan](#).

The following C# code scans the ProductCatalog table to find items that are priced less than 0. The sample specifies the following optional parameters:

- A FilterExpression parameter to retrieve only the items priced less than 0 (error condition).
- A ProjectionExpression parameter to specify the attributes to retrieve for items in the query results.

The following C# example scans the ProductCatalog table to find all items priced less than 0.

Example

```
var forumScanRequest = new ScanRequest
{
    TableName = "ProductCatalog",
    // Optional parameters.
    ExpressionAttributeValues = new Dictionary<string,AttributeValue> {
        {":val", new AttributeValue { N = "0" }}
    },
    FilterExpression = "Price < :val",
    ProjectionExpression = "Id"
};
```

You can also optionally limit the page size or the number of items per page, by adding the optional Limit parameter. Each time you run the Scan method, you get one page of results that has the specified number of items. To fetch the next page, you run the Scan method again by providing the primary key value of the last item in the previous page so that the Scan method can return the next set of items. You provide this information in the request by setting the ExclusiveStartKey

property. Initially, this property can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following C# code example scans the ProductCatalog table. In the request, it specifies the Limit and ExclusiveStartKey optional parameters. The do/while loop continues to scan one page at time until the LastEvaluatedKey returns a null value.

Example

```
Dictionary<string, AttributeValue> lastKeyEvaluated = null;
do
{
    var request = new ScanRequest
    {
        TableName = "ProductCatalog",
        Limit = 10,
        ExclusiveStartKey = lastKeyEvaluated
    };

    var response = client.Scan(request);

    foreach (Dictionary<string, AttributeValue> item
        in response.Items)
    {
        PrintItem(item);
    }
    lastKeyEvaluated = response.LastEvaluatedKey;

} while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);
```

Example - scan using .NET

The following C# code provides a working example that scans the ProductCatalog table to find items priced less than 0.

For step-by-step instructions for testing the following sample, see [.NET code examples](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
```

```
namespace com.amazonaws.codesamples
{
    class LowLevelScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                FindProductsForPriceLessThanZero();

                Console.WriteLine("Example complete. To continue, press Enter");
                Console.ReadLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
        }

        private static void FindProductsForPriceLessThanZero()
        {
            Dictionary<string, AttributeValue> lastKeyEvaluated = null;
            do
            {
                var request = new ScanRequest
                {
                    TableName = "ProductCatalog",
                    Limit = 2,
                    ExclusiveStartKey = lastKeyEvaluated,
                    ExpressionAttributeValues = new Dictionary<string, AttributeValue>
                {
                    {":val", new AttributeValue {
                        N = "0"
                    }}
                },
                FilterExpression = "Price < :val",
                ProjectionExpression = "Id, Title, Price"
            };
        }
    }
}
```

```
        var response = client.Scan(request);

        foreach (Dictionary<string, AttributeValue> item
            in response.Items)
        {
            Console.WriteLine("\nScanThreadTableUsePaging - printing.....");
            PrintItem(item);
        }
        lastKeyEvaluated = response.LastEvaluatedKey;
    } while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);

    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}

private static void PrintItem(
    Dictionary<string, AttributeValue> attributeList)
{
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
    {
        string attributeName = kvp.Key;
        AttributeValue value = kvp.Value;

        Console.WriteLine(
            attributeName + " " +
            (value.S == null ? "" : "S=[[" + value.S + "]]") +
            (value.N == null ? "" : "N=[[" + value.N + "]]") +
            (value.SS == null ? "" : "SS=[[" + string.Join(",", 
value.SS.ToArray()) + "]]") +
            (value.NS == null ? "" : "NS=[[" + string.Join(",", 
value.NS.ToArray()) + "]]")
        );
    }
    Console.WriteLine("*****");
}
}
```

Example - parallel scan using .NET

The following C# code example demonstrates a parallel scan. The program deletes and then recreates the ProductCatalog table and then loads the table with data. When the data load is

finished, the program spawns multiple threads and issues parallel Scan requests. Finally, the program prints a summary of runtime statistics.

For step-by-step instructions for testing the following sample, see [.NET code examples](#).

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelParallelScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "ProductCatalog";
        private static int exampleItemCount = 100;
        private static int scanItemLimit = 10;
        private static int totalSegments = 5;

        static void Main(string[] args)
        {
            try
            {
                DeleteExampleTable();
                CreateExampleTable();
                UploadExampleData();
                ParallelScanExampleTable();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void ParallelScanExampleTable()
        {
```

```
        Console.WriteLine("\n*** Creating {0} Parallel Scan Tasks to scan {1}",  
totalSegments, tableName);  
        Task[] tasks = new Task[totalSegments];  
        for (int segment = 0; segment < totalSegments; segment++)  
{  
            int tmpSegment = segment;  
            Task task = Task.Factory.StartNew(() =>  
            {  
                ScanSegment(totalSegments, tmpSegment);  
            });  
  
            tasks[segment] = task;  
        }  
  
        Console.WriteLine("All scan tasks are created, waiting for them to  
complete.");  
        Task.WaitAll(tasks);  
  
        Console.WriteLine("All scan tasks are completed.");  
    }  
  
    private static void ScanSegment(int totalSegments, int segment)  
{  
        Console.WriteLine("**** Starting to Scan Segment {0} of {1} out of {2} total  
segments ****", segment, tableName, totalSegments);  
        Dictionary<string, AttributeValue> lastEvaluatedKey = null;  
        int totalScannedItemCount = 0;  
        int totalScanRequestCount = 0;  
        do  
{  
            var request = new ScanRequest  
{  
                TableName = tableName,  
                Limit = scanItemLimit,  
                ExclusiveStartKey = lastEvaluatedKey,  
                Segment = segment,  
                TotalSegments = totalSegments  
            };  
  
            var response = client.Scan(request);  
            lastEvaluatedKey = response.LastEvaluatedKey;  
            totalScanRequestCount++;  
            totalScannedItemCount += response.ScannedCount;  
            foreach (var item in response.Items)
```

```
        {
            Console.WriteLine("Segment: {0}, Scanned Item with Title: {1}",
segment, item["Title"].S);
        }
    } while (lastEvaluatedKey.Count != 0);

    Console.WriteLine("**** Completed Scan Segment {0} of {1}.
TotalScanRequestCount: {2}, TotalScannedItemCount: {3} ***", segment, tableName,
totalScanRequestCount, totalScannedItemCount);
}

private static void UploadExampleData()
{
    Console.WriteLine("\n*** Uploading {0} Example Items to {1} Table***",
exampleItemCount, tableName);
    Console.Write("Uploading Items: ");
    for (int itemIndex = 0; itemIndex < exampleItemCount; itemIndex++)
    {
        Console.Write("{0}, ", itemIndex);
        CreateItem(itemIndex.ToString());
    }
    Console.WriteLine();
}

private static void CreateItem(string itemIndex)
{
    var request = new PutItemRequest
    {
        TableName = tableName,
        Item = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            N = itemIndex
        }},
        { "Title", new AttributeValue {
            S = "Book " + itemIndex + " Title"
        }},
        { "ISBN", new AttributeValue {
            S = "11-11-11-11"
        }},
        { "Authors", new AttributeValue {
            SS = new List<string>{"Author1", "Author2" }
        }},
        { "Price", new AttributeValue {

```

```
        N = "20.00"
    },
    { "Dimensions", new AttributeValue {
        S = "8.5x11.0x.75"
    },
    { "InPublication", new AttributeValue {
        BOOL = false
    } }
}
};

client.PutItem(request);
}

private static void CreateExampleTable()
{
    Console.WriteLine("\n*** Creating {0} Table ***", tableName);
    var request = new CreateTableRequest
    {
        AttributeDefinitions = new List<AttributeDefinition>()
    {
        new AttributeDefinition
        {
            AttributeName = "Id",
            AttributeType = "N"
        }
    },
    KeySchema = new List<KeySchemaElement>
    {
        new KeySchemaElement
        {
            AttributeName = "Id",
            KeyType = "HASH" //Partition key
        }
    },
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = 5,
        WriteCapacityUnits = 6
    },
    TableName = tableName
};

    var response = client.CreateTable(request);
}
```

```
        var result = response;
        var tableDescription = result.TableDescription;
        Console.WriteLine("{1}: {0} \t ReadsPerSec: {2} \t WritesPerSec: {3}",
            tableDescription.TableStatus,
            tableDescription.TableName,
            tableDescription.ProvisionedThroughput.ReadCapacityUnits,
            tableDescription.ProvisionedThroughput.WriteCapacityUnits);

        string status = tableDescription.TableStatus;
        Console.WriteLine(tableName + " - " + status);

        WaitUntilTableReady(tableName);
    }

private static void DeleteExampleTable()
{
    try
    {
        Console.WriteLine("\n*** Deleting {0} Table ***", tableName);
        var request = new DeleteTableRequest
        {
            TableName = tableName
        };

        var response = client.DeleteTable(request);
        var result = response;
        Console.WriteLine("{0} is being deleted...", tableName);
        WaitUntilTableDeleted(tableName);
    }
    catch (ResourceNotFoundException)
    {
        Console.WriteLine("{0} Table delete failed: Table does not exist",
            tableName);
    }
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
```

```
{  
    var res = client.DescribeTable(new DescribeTableRequest  
    {  
        TableName = tableName  
    });  
  
    Console.WriteLine("Table name: {0}, status: {1}",  
        res.Table.TableName,  
        res.Table.TableStatus);  
    status = res.Table.TableStatus;  
}  
catch (ResourceNotFoundException)  
{  
    // DescribeTable is eventually consistent. So you might  
    // get resource not found. So we handle the potential exception.  
}  
}  
} while (status != "ACTIVE");  
}  
  
private static void WaitUntilTableDeleted(string tableName)  
{  
    string status = null;  
    // Let us wait until table is deleted. Call DescribeTable.  
    do  
    {  
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.  
        try  
        {  
            var res = client.DescribeTable(new DescribeTableRequest  
            {  
                TableName = tableName  
            });  
  
            Console.WriteLine("Table name: {0}, status: {1}",  
                res.Table.TableName,  
                res.Table.TableStatus);  
            status = res.Table.TableStatus;  
        }  
        catch (ResourceNotFoundException)  
        {  
            Console.WriteLine("Table name: {0} is not found. It is deleted",  
                tableName);  
            return;  
        }  
    }
```

```
        } while (status == "DELETING");
    }
}
}
```

PartiQL - a SQL-compatible query language for Amazon DynamoDB

Amazon DynamoDB supports [PartiQL](#), a SQL-compatible query language, to select, insert, update, and delete data in Amazon DynamoDB. Using PartiQL, you can easily interact with DynamoDB tables and run ad hoc queries using the AWS Management Console, NoSQL Workbench, AWS Command Line Interface, and DynamoDB APIs for PartiQL.

PartiQL operations provide the same availability, latency, and performance as the other DynamoDB data plane operations.

The following sections describe the DynamoDB implementation of PartiQL.

Topics

- [What is PartiQL?](#)
- [PartiQL in Amazon DynamoDB](#)
- [Getting started with PartiQL for DynamoDB](#)
- [PartiQL data types for DynamoDB](#)
- [PartiQL statements for DynamoDB](#)
- [Use PartiQL functions with amazon DynamoDB](#)
- [PartiQL arithmetic, comparison, and logical operators for DynamoDB](#)
- [Performing transactions with PartiQL for DynamoDB](#)
- [Running batch operations with PartiQL for DynamoDB](#)
- [IAM security policies with PartiQL for DynamoDB](#)

What is PartiQL?

PartiQL provides SQL-compatible query access across multiple data stores containing structured data, semistructured data, and nested data. It is widely used within Amazon and is now available as part of many AWS services, including DynamoDB.

For the PartiQL specification and a tutorial on the core query language, see the [PartiQL documentation](#).

Note

- Amazon DynamoDB supports a *subset* of the [PartiQL](#) query language.
- Amazon DynamoDB does not support the [Amazon ion](#) data format or Amazon Ion literals.

PartiQL in Amazon DynamoDB

To run PartiQL queries in DynamoDB, you can use:

- The DynamoDB console
- The NoSQL Workbench
- The AWS Command Line Interface (AWS CLI)
- The DynamoDB APIs

For information about using these methods to access DynamoDB, see [Accessing DynamoDB](#).

Getting started with PartiQL for DynamoDB

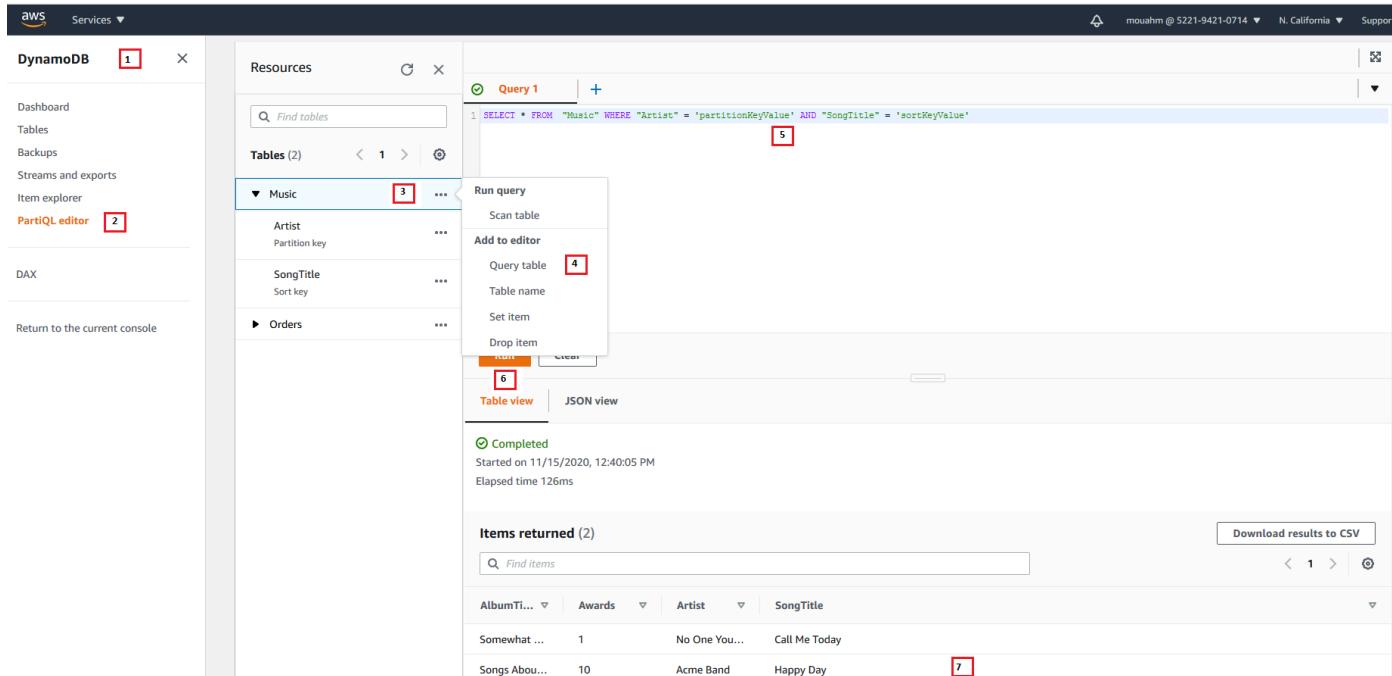
This section describes how to use PartiQL for DynamoDB from the Amazon DynamoDB console, the AWS Command Line Interface (AWS CLI), and DynamoDB APIs.

In the following examples, the DynamoDB table that is defined in the [Getting started with DynamoDB](#) tutorial is a pre-requisite.

For information about using the DynamoDB console, AWS Command Line Interface, or DynamoDB APIs to access DynamoDB, see [Accessing DynamoDB](#).

To [download](#) and use the [NoSQL workbench](#) to build [PartiQL for DynamoDB](#) statements choose **PartiQL operations** at the top right corner of the NoSQL Workbench for DynamoDB [Operation builder](#).

Console



Note

PartiQL for DynamoDB is only available in the new DynamoDB console. To use the new DynamoDB console, choose **Try the Preview of the new console** in the navigation pane on the left side of the console.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **PartiQL editor**.
3. Choose the **Music** table.
4. Choose **Query table**. This action generates a query that will not result in a full table scan.
5. Replace `partitionKeyValue` with the string value `Acme Band`. Replace `sortKeyValue` with the string value `Happy Day`.
6. Choose the **Run** button.
7. You can view the results of the query by choosing the **Table view** or the **JSON view** buttons.

NoSQL workbench

The screenshot shows the NoSQL workbench interface. At the top, there are three radio button options: "PartiQL statement" (selected), "PartiQL transaction", and "PartiQL batch". Below this, a "Statement" input field contains the following PartiQL code:

```
1 | SELECT *
2 | FROM Music
3 | WHERE Artist=? and SongTitle=?
```

Number 1 is highlighted around the first line of the code. Number 2 is highlighted around the third line of the code.

Below the statement, there is a section for "Optional request parameters" with a dropdown menu showing "3.a". Number 3.a is highlighted around this dropdown.

Further down, there is a toggle switch for "Enable strongly consistent reads" and a "Parameters" section. The "Parameters" section contains two entries:

Attribute type	Attribute value
String	Acme Band
String	PartiQL Rocks

Number 3.c is highlighted around the "Attribute value" field for the first entry. Number 3.b is highlighted around the "+ Add new parameter" button.

At the bottom of the interface are four buttons: "Clear form", "Run", "Generate code", and "Save operation". Number 5 is highlighted around the "Run" button, number 4 is highlighted around the "Generate code" button, and number 6 is highlighted around the "Save operation" button.

1. Choose **PartiQL statement**.
2. Enter the following PartiQL [SELECT statement](#)

```
SELECT *
FROM Music
WHERE Artist=? and SongTitle=?
```

3. To specify a value for the Artist and SongTitle parameters:
 - a. Choose **Optional request parameters**.
 - b. Choose **Add new parameters**.
 - c. Choose the attribute type **string** and value **Acme Band**.

- d. Repeat steps b and c, and choose type **string** and value PartiQL Rocks.
4. If you want to generate code, choose **Generate code**.

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

5. If you want the operation to be run immediately, choose **Run**.

AWS CLI

1. Create an item in the Music table using the INSERT PartiQL statement.

```
aws dynamodb execute-statement --statement "INSERT INTO Music \
    VALUE \
    {'Artist':'Acme Band','SongTitle':'PartiQL Rocks'}"
```

2. Retrieve an item from the Music table using the SELECT PartiQL statement.

```
aws dynamodb execute-statement --statement "SELECT * FROM Music \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

3. Update an item in the Music table using the UPDATE PartiQL statement.

```
aws dynamodb execute-statement --statement "UPDATE Music \
    SET AwardsWon=1 \
    SET AwardDetail={'Grammys':[2020,
    2018]} \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

Add a list value for an item in the Music table.

```
aws dynamodb execute-statement --statement "UPDATE Music \
    SET AwardDetail.Grammys
    =list_append(AwardDetail.Grammys,[2016]) \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

Remove a list value for an item in the Music table.

```
aws dynamodb execute-statement --statement "UPDATE Music \
    REMOVE AwardDetail.Grammys[2] \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

Add a new map member for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music \
    SET AwardDetail.BillBoard=[2020] \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

Add a new string set attribute for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music \
    SET BandMembers =<<'member1',
    'member2'>> \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

Update a string set attribute for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music \
    SET BandMembers
    =set_add(BandMembers, <<'newmember'>>) \
    WHERE Artist='Acme Band' AND
    SongTitle='PartiQL Rocks'"
```

4. Delete an item from the `Music` table using the `DELETE` PartiQL statement.

```
aws dynamodb execute-statement --statement "DELETE FROM Music \
    WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'"
```

Java

```
import java.util.ArrayList;
import java.util.List;

import com.amazonaws.AmazonClientException;
```

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.ConditionalCheckFailedException;
import com.amazonaws.services.dynamodbv2.model.ExecuteStatementRequest;
import com.amazonaws.services.dynamodbv2.model.ExecuteStatementResult;
import com.amazonaws.services.dynamodbv2.model.InternalServerErrorException;
import
com.amazonaws.services.dynamodbv2.model.ItemCollectionSizeLimitExceededException;
import
com.amazonaws.services.dynamodbv2.model.ProvisionedThroughputExceededException;
import com.amazonaws.services.dynamodbv2.model.RequestLimitExceededException;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import com.amazonaws.services.dynamodbv2.model.TransactionConflictException;

public class DynamoDBPartiQGettingStarted {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-1");

        try {
            // Create ExecuteStatementRequest
            ExecuteStatementRequest executeStatementRequest = new
ExecuteStatementRequest();
            List<AttributeValue> parameters= getPartiQLParameters();

            //Create an item in the Music table using the INSERT PartiQL statement
            processResults(executeStatementRequest(dynamoDB, "INSERT INTO Music
value {'Artist':?:,'SongTitle':?}'", parameters));

            //Retrieve an item from the Music table using the SELECT PartiQL
statement.
            processResults(executeStatementRequest(dynamoDB, "SELECT * FROM Music
where Artist=? and SongTitle=?", parameters));

            //Update an item in the Music table using the UPDATE PartiQL statement.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music
SET AwardsWon=1 SET AwardDetail={'Grammys':[2020, 2018]} where Artist=? and
SongTitle=?", parameters));

            //Add a list value for an item in the Music table.
        }
    }
}
```

```
        processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
AwardDetail.Grammys =list_append(AwardDetail.Grammys,[2016]) where Artist=? and
SongTitle=?", parameters));

        //Remove a list value for an item in the Music table.
        processResults(executeStatementRequest(dynamoDB, "UPDATE Music REMOVE
AwardDetail.Grammys[2] where Artist=? and SongTitle=?", parameters));

        //Add a new map member for an item in the Music table.
        processResults(executeStatementRequest(dynamoDB, "UPDATE Music set
AwardDetail.BillBoard=[2020] where Artist=? and SongTitle=?", parameters));

        //Add a new string set attribute for an item in the Music table.
        processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
BandMembers =<<'member1', 'member2'>> where Artist=? and SongTitle=?", parameters));

        //update a string set attribute for an item in the Music table.
        processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
BandMembers =set_add(BandMembers, <<'newmember'>>) where Artist=? and SongTitle=?", parameters));

        //Retrieve an item from the Music table using the SELECT PartiQL
statement.
        processResults(executeStatementRequest(dynamoDB, "SELECT * FROM Music
where Artist=? and SongTitle=?", parameters));

        //delete an item from the Music Table
        processResults(executeStatementRequest(dynamoDB, "DELETE FROM Music
where Artist=? and SongTitle=?", parameters));
    } catch (Exception e) {
        handleExecuteStatementErrors(e);
    }
}

private static AmazonDynamoDB createDynamoDbClient(String region) {
    return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
}

private static List<AttributeValue> getPartiQLParameters() {
    List<AttributeValue> parameters = new ArrayList<AttributeValue>();
    parameters.add(new AttributeValue("Acme Band"));
    parameters.add(new AttributeValue("PartiQL Rocks"));
    return parameters;
}
```

```
}

private static ExecuteStatementResult executeStatementRequest(AmazonDynamoDB
client, String statement, List<AttributeValue> parameters ) {
    ExecuteStatementRequest request = new ExecuteStatementRequest();
    request.setStatement(statement);
    request.setParameters(parameters);
    return client.executeStatement(request);
}

private static void processResults(ExecuteStatementResult
executeStatementResult) {
    System.out.println("ExecuteStatement successful: "+
executeStatementResult.toString());

}

// Handles errors during ExecuteStatement execution. Use recommendations in
error messages below to add error handling specific to
// your application use-case.
private static void handleExecuteStatementErrors(Exception exception) {
    try {
        throw exception;
    } catch (ConditionalCheckFailedException ccfe) {
        System.out.println("Condition check specified in the operation failed,
review and update the condition " +
                            "check before retrying. Error: " +
ccfe.getErrorMessage());
    } catch (TransactionConflictException tce) {
        System.out.println("Operation was rejected because there is an ongoing
transaction for the item, generally " +
                            "safe to retry with exponential back-off.
Error: " + tce.getErrorMessage());
    } catch (ItemCollectionSizeLimitExceededException icslee) {
        System.out.println("An item collection is too large, you're using Local
Secondary Index and exceeded " +
                            "size limit of items per
partition key. Consider using Global Secondary Index instead. Error: " +
icslee.getErrorMessage());
    } catch (Exception e) {
        handleCommonErrors(e);
    }
}
```

```
private static void handleCommonErrors(Exception exception) {  
    try {  
        throw exception;  
    } catch (InternalServerErrorException isee) {  
        System.out.println("Internal Server Error, generally safe to retry with  
exponential back-off. Error: " + isee.getErrorMessage());  
    } catch (RequestLimitExceededException rlee) {  
        System.out.println("Throughput exceeds the current throughput limit for  
your account, increase account level throughput before " +  
                           "retrying. Error: " +  
rlee.getErrorMessage());  
    } catch (ProvisionedThroughputExceededException ptee) {  
        System.out.println("Request rate is too high. If you're using a custom  
retry strategy make sure to retry with exponential back-off. " +  
                           "Otherwise consider reducing frequency of  
requests or increasing provisioned capacity for your table or secondary index.  
Error: " +  
                           ptee.getErrorMessage());  
    } catch (ResourceNotFoundException rafe) {  
        System.out.println("One of the tables was not found, verify table exists  
before retrying. Error: " + rafe.getErrorMessage());  
    } catch (AmazonServiceException ase) {  
        System.out.println("An AmazonServiceException occurred, indicates that  
the request was correctly transmitted to the DynamoDB " +  
                           "service, but for some reason, the service  
was not able to process it, and returned an error response instead. Investigate and  
" +  
                           "configure retry strategy. Error type: " +  
ase.getErrorType() + ". Error message: " + ase.getErrorMessage());  
    } catch (AmazonClientException ace) {  
        System.out.println("An AmazonClientException occurred, indicates that  
the client was unable to get a response from DynamoDB " +  
                           "service, or the client was unable to parse  
the response from the service. Investigate and configure retry strategy. "+  
                           "Error: " + ace.getMessage());  
    } catch (Exception e) {  
        System.out.println("An exception occurred, investigate and configure  
retry strategy. Error: " + e.getMessage());  
    }  
}
```

}

PartiQL data types for DynamoDB

The following table lists the data types you can use with PartiQL for DynamoDB.

DynamoDB data type	PartiQL representation	Notes
Boolean	TRUE FALSE	Not case sensitive.
Binary	N/A	Only supported via code.
List	[value1, value2,...]	There are no restrictions on the data types that can be stored in a List type, and the elements in a List do not have to be of the same type.
Map	{ 'name' : value }	There are no restrictions on the data types that can be stored in a Map type, and the elements in a Map do not have to be of the same type.
Null	NULL	Not case sensitive.
Number	1, 1.0, 1e0	Numbers can be positive, negative, or zero. Numbers can have up to 38 digits of precision.
Number Set	<>number1, number2>>	The elements in a number set must be of type Number.
String Set	<>'string1', 'string2'>>	The elements in a string set must be of type String.
String	'string value'	Single quotes must be used to specify String values.

Examples

The following statement demonstrates how to insert the following data types: String, Number, Map, List, Number Set and String Set.

```
INSERT INTO TypesTable value {'primarykey':'1',
'NumberType':1,
'MapType' : {'entryname1': 'value', 'entryname2': 4},
'ListType': [1,'stringval'],
'NumberSetType':<<1,34,32,4.5>>,
'StringSetType':<<'stringval','stringval2'>>
}
```

The following statement demonstrates how to insert new elements into the Map, List, Number Set and String Set types and change the value of a Number type.

```
UPDATE TypesTable
SET NumberType=NumberType + 100
SET MapType.NewMapEntry=[2020, 'stringvalue', 2.4]
SET ListType = LIST_APPEND(ListType, [4, <<'string1', 'string2'>>])
SET NumberSetType= SET_ADD(NumberSetType, <<345, 48.4>>)
SET StringSetType = SET_ADD(StringSetType, <<'stringsetvalue1', 'stringsetvalue2'>>)
WHERE primarykey='1'
```

The following statement demonstrates how to remove elements from the Map, List, Number Set and String Set types and change the value of a Number type.

```
UPDATE TypesTable
SET NumberType=NumberType - 1
REMOVE ListType[1]
REMOVE MapType.NewMapEntry
SET NumberSetType = SET_DELETE( NumberSetType, <<345>>)
SET StringSetType = SET_DELETE( StringSetType, <<'stringsetvalue1'>>)
WHERE primarykey='1'
```

For more information, see [DynamoDB data types](#).

PartiQL statements for DynamoDB

Amazon DynamoDB supports the following PartiQL statements.

Note

DynamoDB does not support all PartiQL statements.

This reference provides basic syntax and usage examples of PartiQL statements that you manually run using the AWS CLI or APIs.

Data manipulation language (DML) is the set of PartiQL statements that you use to manage data in DynamoDB tables. You use DML statements to add, modify, or delete data in a table.

The following DML and query language statements are supported:

- [PartiQL select statements for DynamoDB](#)
- [PartiQL update statements for DynamoDB](#)
- [PartiQL insert statements for DynamoDB](#)
- [PartiQL delete statements for DynamoDB](#)

[Performing transactions with PartiQL for DynamoDB](#) and [Running batch operations with PartiQL for DynamoDB](#) are also supported by PartiQL for DynamoDB.

PartiQL select statements for DynamoDB

Use the SELECT statement to retrieve data from a table in Amazon DynamoDB.

Using the SELECT statement can result in a full table scan if an equality or IN condition with a partition key is not provided in the WHERE clause. A scan operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation.

If you want to avoid full table scan in PartiQL, you can:

- Author your SELECT statements to not result in full table scans by making sure your [WHERE clause condition](#) is configured accordingly.
- Disable full table scans using the IAM policy specified at [Example: Allow select statements and deny full table scan statements in PartiQL for DynamoDB](#), in the DynamoDB developer guide.

For more information see [Best practices for querying and scanning data](#), in the DynamoDB developer guide.

Topics

- [Syntax](#)
- [Parameters](#)
- [Examples](#)

Syntax

```
SELECT expression [, ...]
FROM table[.index]
[ WHERE condition ] [ [ORDER BY key [DESC|ASC] , ...]
```

Parameters

expression

(Required) A projection formed from the * wildcard or a projection list of one or more attribute names or document paths from the result set. An expression can consist of calls to [Use PartiQL functions with amazon DynamoDB](#) or fields that are modified by [PartiQL arithmetic, comparison, and logical operators for DynamoDB](#).

table

(Required) The table name to query.

index

(Optional) The name of the index to query.

Note

You must add double quotation marks to the table name and index name when querying an index.

```
SELECT *
FROM "TableName"."IndexName"
```

condition

(Optional) The selection criteria for the query.

⚠ Important

To ensure that a SELECT statement does not result in a full table scan, the WHERE clause condition must specify a partition key. Use the equality or IN operator.

For example, if you have an Orders table with an OrderID partition key and other non-key attributes, including an Address, the following statements would not result in a full table scan:

```
SELECT *
FROM "Orders"
WHERE OrderID = 100
```

```
SELECT *
FROM "Orders"
WHERE OrderID = 100 and Address='some address'
```

```
SELECT *
FROM "Orders"
WHERE OrderID = 100 or pk = 200
```

```
SELECT *
FROM "Orders"
WHERE OrderID IN [100, 300, 234]
```

The following SELECT statements, however, will result in a full table scan:

```
SELECT *
FROM "Orders"
WHERE OrderID > 1
```

```
SELECT *
FROM "Orders"
WHERE Address='some address'
```

```
SELECT *
FROM "Orders"
WHERE OrderID = 100 OR Address='some address'
```

key

(Optional) A hash key or a sort key to use to order returned results. The default order is ascending (ASC) specify DESC if you want the results retuned in descending order.

Note

If you omit the WHERE clause, then all of the items in the table are retrieved.

Examples

The following query returns one item, if one exists, from the Orders table by specifying the partition key, OrderID, and using the equality operator.

```
SELECT OrderID, Total  
FROM "Orders"  
WHERE OrderID = 1
```

The following query returns all items in the Orders table that have a specific partition key, OrderID, values using the OR operator.

```
SELECT OrderID, Total  
FROM "Orders"  
WHERE OrderID = 1 OR OrderID = 2
```

The following query returns all items in the Orders table that have a specific partition key, OrderID, values using the IN operator. The returned results are in descending order, based on the OrderID key attribute value.

```
SELECT OrderID, Total  
FROM "Orders"  
WHERE OrderID IN [1, 2, 3] ORDER BY OrderID DESC
```

The following query shows a full table scan that returns all items from the Orders table that have a Total greater than 500, where Total is a non-key attribute.

```
SELECT OrderID, Total  
FROM "Orders"
```

```
WHERE Total > 500
```

The following query shows a full table scan that returns all items from the `Orders` table within a specific `Total` order range, using the `IN` operator and a non-key attribute `Total`.

```
SELECT OrderID, Total  
FROM "Orders"  
WHERE Total IN [500, 600]
```

The following query shows a full table scan that returns all items from the `Orders` table within a specific `Total` order range, using the `BETWEEN` operator and a non-key attribute `Total`.

```
SELECT OrderID, Total  
FROM "Orders"  
WHERE Total BETWEEN 500 AND 600
```

The following query returns the first date a firestick device was used to watch by specifying the partition key `CustomerID` and sort key `MovieID` in the `WHERE` clause condition and using document paths in the `SELECT` clause.

```
SELECT Devices.FireStick.DateWatched[0]  
FROM WatchList  
WHERE CustomerID= 'C1' AND MovieID= 'M1'
```

The following query shows a full table scan that returns the list of items where a firestick device was first used after 12/24/19 using document paths in the `WHERE` clause condition.

```
SELECT Devices  
FROM WatchList  
WHERE Devices.FireStick.DateWatched[0] >= '12/24/19'
```

PartiQL update statements for DynamoDB

Use the `UPDATE` statement to modify the value of one or more attributes within an item in an Amazon DynamoDB table.

Note

You can only update one item at a time; you cannot issue a single DynamoDB PartiQL statement that updates multiple items. For information on updating multiple items, see

[Performing transactions with PartiQL for DynamoDB](#) or [Running batch operations with PartiQL for DynamoDB](#).

Topics

- [Syntax](#)
- [Parameters](#)
- [Return value](#)
- [Examples](#)

Syntax

```
UPDATE table
[SET | REMOVE] path [= data] [...]
WHERE condition [RETURNING returnvalues]
<returnvalues> ::= [ALL OLD | MODIFIED OLD | ALL NEW | MODIFIED NEW] *
```

Parameters

table

(Required) The table containing the data to be modified.

path

(Required) An attribute name or document path to be created or modified.

data

(Required) An attribute value or the result of an operation.

The supported operations to use with SET:

- LIST_APPEND: adds a value to a list type.
- SET_ADD: adds a value to a number or string set.
- SET_DELETE: removes a value from a number or string set.

condition

(Required) The selection criteria for the item to be modified. This condition must resolve to a single primary key value.

returnvalues

(Optional) Use `returnvalues` if you want to get the item attributes as they appear before or after they are updated. The valid values are:

- `ALL OLD` * - Returns all of the attributes of the item, as they appeared before the update operation.
- `MODIFIED OLD` * - Returns only the updated attributes, as they appeared before the update operation.
- `ALL NEW` * - Returns all of the attributes of the item, as they appear after the update operation.
- `MODIFIED NEW` * - Returns only the updated attributes, as they appear after the `UpdateItem` operation.

Return value

This statement does not return a value unless `returnvalues` parameter is specified.

Note

If the WHERE clause of the UPDATE statement does not evaluate to true for any item in the DynamoDB table, `ConditionalCheckFailedException` is returned.

Examples

Update an attribute value in an existing item. If the attribute does not exist, it is created.

The following query updates an item in the "Music" table by adding an attribute of type number (`AwardsWon`) and an attribute of type map (`AwardDetail`).

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

You can add `RETURNING ALL OLD *` to return the attributes as they appeared before the Update operation.

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
RETURNING ALL OLD *
```

This returns the following:

```
{
  "Items": [
    {
      "Artist": {
        "S": "Acme Band"
      },
      "SongTitle": {
        "S": "PartiQL Rocks"
      }
    }
  ]
}
```

You can add RETURNING ALL NEW * to return the attributes as they appeared after the Update operation.

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
RETURNING ALL NEW *
```

This returns the following:

```
{
  "Items": [
    {
      "AwardDetail": {
        "M": {
          "Grammys": {
            "L": [
              {
                "N": "2020"
              }
            ]
          }
        }
      }
    }
  ]
}
```

```
        },
        {
            "N": "2018"
        }
    ]
}
},
"AwardsWon": {
    "N": "1"
}
]
}
```

The following query updates an item in the "Music" table by appending to a list AwardDetail.Grammys.

```
UPDATE "Music"
SET AwardDetail.Grammys =list_append(AwardDetail.Grammys,[2016])
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by removing from a list AwardDetail.Grammys.

```
UPDATE "Music"
REMOVE AwardDetail.Grammys[2]
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by adding BillBoard to the map AwardDetail.

```
UPDATE "Music"
SET AwardDetail.BillBoard=[2020]
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by adding the string set attribute BandMembers.

```
UPDATE "Music"
SET BandMembers =<<'member1', 'member2'>>
```

```
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by adding newbandmember to the string set BandMembers.

```
UPDATE "Music"
SET BandMembers =set_add(BandMembers, <<'newbandmember'>>)
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

PartiQL delete statements for DynamoDB

Use the DELETE statement to delete an existing item from your Amazon DynamoDB table.

Note

You can only delete one item at a time. You cannot issue a single DynamoDB PartiQL statement that deletes multiple items. For information on deleting multiple items, see [Performing transactions with PartiQL for DynamoDB](#) or [Running batch operations with PartiQL for DynamoDB](#).

Topics

- [Syntax](#)
- [Parameters](#)
- [Return value](#)
- [Examples](#)

Syntax

```
DELETE FROM table
WHERE condition [RETURNING returnvalues]
<returnvalues> ::= ALL OLD *
```

Parameters

table

(Required) The DynamoDB table containing the item to be deleted.

condition

(Required) The selection criteria for the item to be deleted; this condition must resolve to a single primary key value.

returnvalues

(Optional) Use `returnvalues` if you want to get the item attributes as they appeared before they were deleted. The valid values are:

- `ALL OLD *` - The content of the old item is returned.

Return value

This statement does not return a value unless `returnvalues` parameter is specified.

Note

If the DynamoDB table does not have any item with the same primary key as that of the item for which the `DELETE` is issued, `SUCCESS` is returned with 0 items deleted. If the table has an item with same primary key, but the condition in the `WHERE` clause of the `DELETE` statement evaluates to false, `ConditionalCheckFailedException` is returned.

Examples

The following query deletes an item in the "Music" table.

```
DELETE FROM "Music" WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks'
```

You can add the parameter `RETURNING ALL OLD *` to return the data that was deleted.

```
DELETE FROM "Music" WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks'  
RETURNING ALL OLD *
```

The Delete statement now returns the following:

```
{  
  "Items": [  
    {
```

```
        "Artist": {  
            "S": "Acme Band"  
        },  
        "SongTitle": {  
            "S": "PartiQL Rocks"  
        }  
}
```

PartiQL insert statements for DynamoDB

Use the `INSERT` statement to add an item to a table in Amazon DynamoDB.

Note

You can only insert one item at a time; you cannot issue a single DynamoDB PartiQL statement that inserts multiple items. For information on inserting multiple items, see [Performing transactions with PartiQL for DynamoDB](#) or [Running batch operations with PartiQL for DynamoDB](#).

Topics

- [Syntax](#)
- [Parameters](#)
- [Return value](#)
- [Examples](#)

Syntax

Insert a single item.

```
INSERT INTO table VALUE item
```

Parameters

table

(Required) The table where you want to insert the data. The table must already exist.

item

(Required) A valid DynamoDB item represented as a [PartiQL tuple](#). You must specify only *one* item and each attribute name in the item is case-sensitive and can be denoted with *single* quotation marks (' . . . ') in PartiQL.

String values are also denoted with *single* quotation marks (' . . . ') in PartiQL.

Return value

This statement does not return any values.

Note

If the DynamoDB table already has an item with the same primary key as the primary key of the item being inserted, `DuplicateItemException` is returned.

Examples

```
INSERT INTO "Music" value {'Artist' : 'Acme Band', 'SongTitle' : 'PartiQL Rocks'}
```

Use PartiQL functions with amazon DynamoDB

PartiQL in Amazon DynamoDB supports the following built-in variants of SQL standard functions.

Note

Any SQL functions that are not included in this list are not currently supported in DynamoDB.

Aggregate functions

- [Using the SIZE function with PartiQL for amazon DynamoDB](#)

Conditional functions

- [Using the EXISTS function with PartiQL for DynamoDB](#)

- [Using the ATTRIBUTE_TYPE function with PartiQL for DynamoDB](#)
- [Using the BEGINS_WITH function with PartiQL for DynamoDB](#)
- [Using the CONTAINS function with PartiQL for DynamoDB](#)
- [Using the MISSING function with PartiQL for DynamoDB](#)

Using the EXISTS function with PartiQL for DynamoDB

You can use EXISTS to perform the same function as ConditionCheck does in the [TransactWriteItems](#) API. The EXISTS function can only be used in transactions.

Given a value, returns TRUE if the value is a non-empty collection. Otherwise, returns FALSE.

Note

This function can only be used in transactional operations.

Syntax

```
EXISTS ( statement )
```

Arguments

statement

(Required) The SELECT statement that the function evaluates.

Note

The SELECT statement must specify a full primary key and one other condition.

Return type

bool

Examples

```
EXISTS(  
    SELECT * FROM "Music"
```

```
WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks')
```

Using the `BEGINS_WITH` function with PartiQL for DynamoDB

Returns TRUE if the attribute specified begins with a particular substring.

Syntax

```
begins_with(path, value )
```

Arguments

path

(Required) The attribute name or document path to use.

value

(Required) The string to search for.

Return type

bool

Examples

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND begins_with("Address", '7834 24th')
```

Using the `MISSING` function with PartiQL for DynamoDB

Returns TRUE if the item does not contain the attribute specified. Only equality and inequality operators can be used with this function.

Syntax

```
attributename IS | IS NOT MISSING
```

Arguments

attributename

(Required) The attribute name to look for.

Return type

bool

Examples

```
SELECT * FROM Music WHERE "Awards" is MISSING
```

Using the ATTRIBUTE_TYPE function with PartiQL for DynamoDB

Returns TRUE if the attribute at the specified path is of a particular data type.

Syntax

```
attribute_type( attributename, type )
```

Arguments

attributename

(Required) The attribute name to use.

type

(Required) The attribute type to check for. For a list of valid values, see DynamoDB [attribute_type](#).

Return type

bool

Examples

```
SELECT * FROM "Music" WHERE attribute_type("Artist", 'S')
```

Using the CONTAINS function with PartiQL for DynamoDB

Returns TRUE if the attribute specified by the path is one of the following:

- A String that contains a particular substring.

- A Set that contains a particular element within the set.

For more information, see the DynamoDB [contains](#) function.

Syntax

```
contains( path, substring )
```

Arguments

path

(Required) The attribute name or document path to use.

substring

(Required) The attribute substring or set member to check for. For more information, see the DynamoDB [contains](#) function.

Return type

bool

Examples

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND contains("Address", 'Kirkland')
```

Using the SIZE function with PartiQL for amazon DynamoDB

Returns a number representing an attribute's size in bytes. The following are valid data types for use with size. For more information, see the DynamoDB [size](#) function.

Syntax

```
size( path )
```

Arguments

path

(Required) The attribute name or document path.

For supported types, see DynamoDB [size](#) function.

Return type

int

Examples

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND size("Image") >300
```

PartiQL arithmetic, comparison, and logical operators for DynamoDB

PartiQL in Amazon DynamoDB supports the following [SQL standard operators](#).

Note

Any SQL operators that are not included in this list are not currently supported in DynamoDB.

Arithmetic operators

Operator	Description
+	Add
-	Subtract

Comparison operators

Operator	Description
=	Equal to
<>	Not Equal to
!=	Not Equal to
>	Greater than

Operator	Description
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators

Operator	Description
AND	TRUE if all the conditions separated by AND are TRUE
BETWEEN	TRUE if the operand is within the range of comparisons
IN	TRUE if the operand is equal to one of a list of expressions (at max 50 hash attribute values or at max 100 non-key attribute values)
IS	TRUE if the operand is a given, PartiQL data type, including NULL or MISSING
NOT	Reverses the value of a given Boolean expression
OR	TRUE if any of the conditions separated by OR are TRUE

Performing transactions with PartiQL for DynamoDB

This section describes how to use transactions with PartiQL for DynamoDB. PartiQL transactions are limited to 100 total statements (actions).

For more information on DynamoDB transactions, see [Managing complex workflows with DynamoDB transactions](#).

Note

The entire transaction must consist of either read statements or write statements. You can't mix both in one transaction. The EXISTS function is an exception. You can use it to check the condition of specific attributes of the item in a similar manner to ConditionCheck in the [TransactWriteItems](#) API operation.

Topics

- [Syntax](#)
- [Parameters](#)
- [Return values](#)
- [Examples](#)

Syntax

```
[  
  {  
    "Statement": " statement ",  
    "Parameters": [  
      {  
        " parametertype " : " parametervalue "  
      }, ...]  
    } , ...  
  ]
```

Parameters***statement***

(Required) A PartiQL for DynamoDB supported statement.

Note

The entire transaction must consist of either read statements or write statements. You can't mix both in one transaction.

parameter type

(Optional) A DynamoDB type, if parameters were used when specifying the PartiQL statement.

parameter value

(Optional) A parameter value if parameters were used when specifying the PartiQL statement.

Return values

This statement doesn't return any values for Write operations (INSERT, UPDATE, or DELETE).

However, it returns different values for Read operations (SELECT) based on the conditions specified in the WHERE clause.

Note

If any of the singleton INSERT, UPDATE, or DELETE operations return an error, the transactions are canceled with the TransactionCanceledException exception, and the cancellation reason code includes the errors from the individual singleton operations.

Examples

The following example runs multiple statements as a transaction.

AWS CLI

1. Save the following JSON code to a file called partiq.json.

```
[  
  {  
    "Statement": "EXISTS(SELECT * FROM \"Music\" where Artist='No One You  
Know' and SongTitle='Call Me Today' and Awards is MISSING)"  
  },  
  {  
    "Statement": "INSERT INTO Music value {'Artist':?:,'SongTitle':?}',  
    "Parameters": [{\"S\": \"Acme Band\"}, {"S": \"Best Song\"}]  
  },  
  {  
    "Statement": "UPDATE \"Music\" SET AwardsWon=1 SET  
    AwardDetail={'Grammys':[2020, 2018]} where Artist='Acme Band' and  
    SongTitle='PartiQL Rocks'"  
  }]
```

```
    }  
]
```

- Run the following command in a command prompt.

```
aws dynamodb execute-transaction --transact-statements file://partiql.json
```

Java

```
public class DynamoDBPartiqlTransaction {  
  
    public static void main(String[] args) {  
        // Create the DynamoDB Client with the region you want  
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-2");  
  
        try {  
            // Create ExecuteTransactionRequest  
            ExecuteTransactionRequest executeTransactionRequest =  
createExecuteTransactionRequest();  
            ExecuteTransactionResult executeTransactionResult =  
dynamoDB.executeTransaction(executeTransactionRequest);  
            System.out.println("ExecuteTransaction successful.");  
            // Handle executeTransactionResult  
  
        } catch (Exception e) {  
            handleExecuteTransactionErrors(e);  
        }  
    }  
  
    private static AmazonDynamoDB createDynamoDbClient(String region) {  
        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();  
    }  
  
    private static ExecuteTransactionRequest createExecuteTransactionRequest() {  
        ExecuteTransactionRequest request = new ExecuteTransactionRequest();  
  
        // Create statements  
        List<ParameterizedStatement> statements = getPartiQLTransactionStatements();  
  
        request.setTransactStatements(statements);  
        return request;  
    }  
}
```

```
private static List<ParameterizedStatement> getPartiQLTransactionStatements() {  
    List<ParameterizedStatement> statements = new  
    ArrayList<ParameterizedStatement>();  
  
    statements.add(new ParameterizedStatement()  
        .withStatement("EXISTS(SELECT * FROM \"Music\" where  
Artist='No One You Know' and SongTitle='Call Me Today' and Awards is MISSING)"));  
  
    statements.add(new ParameterizedStatement()  
        .withStatement("INSERT INTO \"Music\" value  
{'Artist':'?', 'SongTitle':'?'})")  
        .withParameters(new AttributeValue("Acme Band"), new  
AttributeValue("Best Song")));  
  
    statements.add(new ParameterizedStatement()  
        .withStatement("UPDATE \"Music\" SET AwardsWon=1  
SET AwardDetail={'Grammys':[2020, 2018]} where Artist='Acme Band' and  
SongTitle='PartiQL Rocks'"));  
  
    return statements;  
}  
  
// Handles errors during ExecuteTransaction execution. Use recommendations in  
error messages below to add error handling specific to  
// your application use-case.  
private static void handleExecuteTransactionErrors(Exception exception) {  
    try {  
        throw exception;  
    } catch (TransactionCanceledException tce) {  
        System.out.println("Transaction Cancelled, implies a client issue, fix  
before retrying. Error: " + tce.getErrorMessage());  
    } catch (TransactionInProgressException tipe) {  
        System.out.println("The transaction with the given request token is  
already in progress, consider changing " +  
            "retry strategy for this type of error. Error: " +  
tipe.getErrorMessage());  
    } catch (IdempotentParameterMismatchException ipme) {  
        System.out.println("Request rejected because it was retried with a  
different payload but with a request token that was already used, " +  
            "change request token for this payload to be accepted. Error: " +  
ipme.getErrorMessage());  
    } catch (Exception e) {  
        handleCommonErrors(e);  
    }  
}
```

```
        }

    }

    private static void handleCommonErrors(Exception exception) {
        try {
            throw exception;
        } catch (InternalServerErrorException isee) {
            System.out.println("Internal Server Error, generally safe to retry with exponential back-off. Error: " + isee.getErrorMessage());
        } catch (RequestLimitExceededException rlee) {
            System.out.println("Throughput exceeds the current throughput limit for your account, increase account level throughput before " +
                "retrying. Error: " + rlee.getErrorMessage());
        } catch (ProvisionedThroughputExceededException ptee) {
            System.out.println("Request rate is too high. If you're using a custom retry strategy make sure to retry with exponential back-off. " +
                "Otherwise consider reducing frequency of requests or increasing provisioned capacity for your table or secondary index. Error: " +
                ptee.getErrorMessage());
        } catch (ResourceNotFoundException rafe) {
            System.out.println("One of the tables was not found, verify table exists before retrying. Error: " + rafe.getErrorMessage());
        } catch (AmazonServiceException ase) {
            System.out.println("An AmazonServiceException occurred, indicates that the request was correctly transmitted to the DynamoDB " +
                "service, but for some reason, the service was not able to process it, and returned an error response instead. Investigate and " +
                "configure retry strategy. Error type: " + ase.getErrorType() + ". Error message: " + ase.getErrorMessage());
        } catch (AmazonClientException ace) {
            System.out.println("An AmazonClientException occurred, indicates that the client was unable to get a response from DynamoDB " +
                "service, or the client was unable to parse the response from the service. Investigate and configure retry strategy. " +
                "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure retry strategy. Error: " + e.getMessage());
        }
    }
}
```

The following example shows the different return values when DynamoDB reads items with different conditions specified in the WHERE clause.

AWS CLI

1. Save the following JSON code to a file called partiqql.json.

```
[  
    // Item exists and projected attribute exists  
    {  
        "Statement": "SELECT * FROM \"Music\" WHERE Artist='No One You Know' and SongTitle='Call Me Today'"  
    },  
    // Item exists but projected attributes do not exist  
    {  
        "Statement": "SELECT non_existent_projected_attribute FROM \"Music\" WHERE Artist='No One You Know' and SongTitle='Call Me Today'"  
    },  
    // Item does not exist  
    {  
        "Statement": "SELECT * FROM \"Music\" WHERE Artist='No One I Know' and SongTitle='Call You Today'"  
    }  
]
```

2. following command in a command prompt.

```
aws dynamodb execute-transaction --transact-statements file://partiqql.json
```

3. The following response is returned:

```
{  
    "Responses": [  
        // Item exists and projected attribute exists  
        {  
            "Item": {  
                "Artist":{  
                    "S": "No One You Know"  
                },  
                "SongTitle":{  
                    "S": "Call Me Today"  
                }  
            }  
        }  
    ]
```

```
        },
        // Item exists but projected attributes do not exist
        {
            "Item": {}
        },
        // Item does not exist
        {}
    ]
}
```

Running batch operations with PartiQL for DynamoDB

This section describes how to use batch statements with PartiQL for DynamoDB.

Note

- The entire batch must consist of either read statements or write statements; you cannot mix both in one batch.
- BatchExecuteStatement and BatchWriteItem can perform no more than 25 statements per batch.

Topics

- [Syntax](#)
- [Parameters](#)
- [Examples](#)

Syntax

```
[
{
    "Statement": "statement",
    "Parameters": [
        {
            "parametertype": "parametervalue"
        }, ...
    ], ...
}]
```

Parameters

statement

(Required) A PartiQL for DynamoDB supported statement.

Note

- The entire batch must consist of either read statements or write statements; you cannot mix both in one batch.
- BatchExecuteStatement and BatchWriteItem can perform no more than 25 statements per batch.

parametertype

(Optional) A DynamoDB type, if parameters were used when specifying the PartiQL statement.

parametervalue

(Optional) A parameter value if parameters were used when specifying the PartiQL statement.

Examples

AWS CLI

1. Save the following json to a file called partiql.json

```
[  
  {  
    "Statement": "INSERT INTO Music value {'Artist':'?', 'SongTitle':'?'}",  
    "Parameters": [{"S": "Acme Band"}, {"S": "Best Song"}]  
  },  
  {  
    "Statement": "UPDATE Music SET AwardsWon=1 SET AwardDetail={'Grammys':[2020,  
    2018]} where Artist='Acme Band' and SongTitle='PartiQL Rocks'"  
  }  
]
```

2. Run the following command in a command prompt.

```
aws dynamodb batch-execute-statement --statements file://partiql.json
```

Java

```
public class DynamoDBPartiqlBatch {  
  
    public static void main(String[] args) {  
        // Create the DynamoDB Client with the region you want  
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-2");  
  
        try {  
            // Create BatchExecuteStatementRequest  
            BatchExecuteStatementRequest batchExecuteStatementRequest =  
createBatchExecuteStatementRequest();  
            BatchExecuteStatementResult batchExecuteStatementResult =  
dynamoDB.batchExecuteStatement(batchExecuteStatementRequest);  
            System.out.println("BatchExecuteStatement successful.");  
            // Handle batchExecuteStatementResult  
  
        } catch (Exception e) {  
            handleBatchExecuteStatementErrors(e);  
        }  
    }  
  
    private static AmazonDynamoDB createDynamoDbClient(String region) {  
  
        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();  
    }  
  
    private static BatchExecuteStatementRequest createBatchExecuteStatementRequest()  
{  
        BatchExecuteStatementRequest request = new BatchExecuteStatementRequest();  
  
        // Create statements  
        List<BatchStatementRequest> statements = getPartiQLBatchStatements();  
  
        request.setStatements(statements);  
        return request;  
    }  
  
    private static List<BatchStatementRequest> getPartiQLBatchStatements() {
```

```
        List<BatchStatementRequest> statements = new
ArrayList<BatchStatementRequest>();

        statements.add(new BatchStatementRequest()
                        .withStatement("INSERT INTO Music value
{'Artist':'Acme Band', 'SongTitle':'PartiQL Rocks'}"));

        statements.add(new BatchStatementRequest()
                        .withStatement("UPDATE Music set
AwardDetail.BillBoard=[2020] where Artist='Acme Band' and SongTitle='PartiQL
Rocks'"));

        return statements;
    }

// Handles errors during BatchExecuteStatement execution. Use recommendations in
error messages below to add error handling specific to
// your application use-case.
private static void handleBatchExecuteStatementErrors(Exception exception) {
    try {
        throw exception;
    } catch (Exception e) {
        // There are no API specific errors to handle for BatchExecuteStatement,
common DynamoDB API errors are handled below
        handleCommonErrors(e);
    }
}

private static void handleCommonErrors(Exception exception) {
    try {
        throw exception;
    } catch (InternalServerErrorException isee) {
        System.out.println("Internal Server Error, generally safe to retry with
exponential back-off. Error: " + isee.getErrorMessage());
    } catch (RequestLimitExceededException rlee) {
        System.out.println("Throughput exceeds the current throughput limit for
your account, increase account level throughput before " +
                "retrying. Error: " + rlee.getErrorMessage());
    } catch (ProvisionedThroughputExceededException ptee) {
        System.out.println("Request rate is too high. If you're using a custom
retry strategy make sure to retry with exponential back-off. " +
                "Otherwise consider reducing frequency of requests or increasing
provisioned capacity for your table or secondary index. Error: " +
                ptee.getErrorMessage());
    }
}
```

```
        } catch (ResourceNotFoundException rufe) {
            System.out.println("One of the tables was not found, verify table exists before retrying. Error: " + rufe.getErrorMessage());
        } catch (AmazonServiceException ase) {
            System.out.println("An AmazonServiceException occurred, indicates that the request was correctly transmitted to the DynamoDB " +
                "service, but for some reason, the service was not able to process it, and returned an error response instead. Investigate and " +
                "configure retry strategy. Error type: " + ase.getErrorType() + ". Error message: " + ase.getErrorMessage());
        } catch (AmazonClientException ace) {
            System.out.println("An AmazonClientException occurred, indicates that the client was unable to get a response from DynamoDB " +
                "service, or the client was unable to parse the response from the service. Investigate and configure retry strategy. " +
                "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure retry strategy. Error: " + e.getMessage());
        }
    }

}
```

IAM security policies with PartiQL for DynamoDB

The following permissions are required:

- To read items using PartiQL for DynamoDB, you must have dynamodb:PartiQLSelect permission on the table or index.
- To insert items using PartiQL for DynamoDB, you must have dynamodb:PartiQLInsert permission on the table or index.
- To update items using PartiQL for DynamoDB, you must have dynamodb:PartiQLUpdate permission on the table or index.
- To delete items using PartiQL for DynamoDB, you must have dynamodb:PartiQLDelete permission on the table or index.

Example: Allow all PartiQL for DynamoDB statements (Select/Insert/Update/Delete) on a table

The following IAM policy grants permissions to run all PartiQL for DynamoDB statements on a table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PartiQLInsert",  
                "dynamodb:PartiQLUpdate",  
                "dynamodb:PartiQLDelete",  
                "dynamodb:PartiQLSelect"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Music"  
            ]  
        }  
    ]  
}
```

Example: Allow PartiQL for DynamoDB select statements on a table

The following IAM policy grants permissions to run the select statement on a specific table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PartiQLSelect"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Music"  
            ]  
        }  
    ]  
}
```

Example: Allow PartiQL for DynamoDB insert statements on an index

The following IAM policy grants permissions to run the `insert` statement on a specific index.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PartiQLInsert"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Music/index/index1"  
            ]  
        }  
    ]  
}
```

Example: Allow PartiQL for DynamoDB transactional statements only on a table

The following IAM policy grants permissions to run only transactional statements on a specific table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PartiQLInsert",  
                "dynamodb:PartiQLUpdate",  
                "dynamodb:PartiQLDelete",  
                "dynamodb:PartiQLSelect"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Music"  
            ],  
            "Condition": {  
                "StringEquals": {  
                    "dynamodb:EnclosingOperation": [  
                        "ExecuteTransaction"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        }
    }
}
]
```

Example: Allow PartiQL for DynamoDB non-transactional reads and writes and block PartiQL transactional reads and writes on a table.

The following IAM policy grants permissions to run PartiQL for DynamoDB non-transactional reads and writes while blocking PartiQL for DynamoDB transactional reads and writes.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:PartiQLInsert",
                "dynamodb:PartiQLUpdate",
                "dynamodb:PartiQLDelete",
                "dynamodb:PartiQLSelect"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
            ],
            "Condition": {
                "StringEquals": {
                    "dynamodb:EnclosingOperation": [
                        "ExecuteTransaction"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PartiQLInsert",
                "dynamodb:PartiQLUpdate",
                "dynamodb:PartiQLDelete",
                "dynamodb:PartiQLSelect"
            ],
            "Resource": [

```

```
        "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
    ]
}
]
```

Example: Allow select statements and deny full table scan statements in PartiQL for DynamoDB

The following IAM policy grants permissions to run the `select` statement on a specific table while blocking select statements that result in a full table scan.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "dynamodb:PartiQLSelect"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/WatchList"
      ],
      "Condition": {
        "Bool": {
          "dynamodb:FullTableScan": [
            "true"
          ]
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PartiQLSelect"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/WatchList"
      ]
    }
  ]
}
```

Improving data access with secondary indexes

Topics

- [Using Global Secondary Indexes in DynamoDB](#)
- [Local Secondary Indexes](#)

Amazon DynamoDB provides fast access to items in a table by specifying primary key values. However, many applications might benefit from having one or more secondary (or alternate) keys available, to allow efficient access to data with attributes other than the primary key. To address this, you can create one or more secondary indexes on a table and issue Query or Scan requests against these indexes.

A *secondary index* is a data structure that contains a subset of attributes from a table, along with an alternate key to support Query operations. You can retrieve data from the index using a Query, in much the same way as you use Query with a table. A table can have multiple secondary indexes, which give your applications access to many different query patterns.

 **Note**

You can also Scan an index, in much the same way as you would Scan a table.

Every secondary index is associated with exactly one table, from which it obtains its data. This is called the *base table* for the index. When you create an index, you define an alternate key for the index (partition key and sort key). You also define the attributes that you want to be *projected*, or copied, from the base table into the index. DynamoDB copies these attributes into the index, along with the primary key attributes from the base table. You can then query or scan the index just as you would query or scan a table.

Every secondary index is automatically maintained by DynamoDB. When you add, modify, or delete items in the base table, any indexes on that table are also updated to reflect these changes.

DynamoDB supports two types of secondary indexes:

- [**Global secondary index**](#) — An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary

index is stored in its own partition space away from the base table and scales separately from the base table.

- **Local secondary index** — An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value.

For a comparison of global secondary indexes and local secondary indexes, see this video.

[Making the right choice between GSI and LSI](#)

You should consider your application's requirements when you determine which type of index to use. The following table shows the main differences between a global secondary index and a local secondary index.

Characteristic	Global secondary index	Local secondary index
Key Schema	The primary key of a global secondary index can be either simple (partition key) or composite (partition key and sort key).	The primary key of a local secondary index must be composite (partition key and sort key).
Key Attributes	The index partition key and sort key (if present) can be any base table attributes of type string, number, or binary.	The partition key of the index is the same attribute as the partition key of the base table. The sort key can be any base table attribute of type string, number, or binary.
Size Restrictions Per Partition Key Value	There are no size restrictions for global secondary indexes.	For each partition key value, the total size of all indexed items must be 10 GB or less.
Online Index Operations	Global secondary indexes can be created at the same time that you create a table. You can also add a new global secondary index to an existing	Local secondary indexes are created at the same time that you create a table. You cannot add a local secondary index to an existing table, nor can you

Characteristic	Global secondary index	Local secondary index
	<p>table, or delete an existing global secondary index.</p> <p>For more information, see Managing Global Secondary Indexes.</p>	delete any local secondary indexes that currently exist.
Queries and Partitions	A global secondary index lets you query over the entire table, across all partitions.	A local secondary index lets you query over a single partition, as specified by the partition key value in the query.
Read Consistency	Queries on global secondary indexes support eventual consistency only.	When you query a local secondary index, you can choose either eventual consistency or strong consistency.
Provisioned Throughput Consumption	Every global secondary index has its own provisioned throughput settings for read and write activity. Queries or scans on a global secondary index consume capacity units from the index, not from the base table. The same holds true for global secondary index updates due to table writes. A global secondary index associated with global tables consumes write capacity units.	Queries or scans on a local secondary index consume read capacity units from the base table. When you write to a table its local secondary indexes are also updated, and these updates consume write capacity units from the base table. A local secondary index associated with global tables consumes replicated write capacity units.

Characteristic	Global secondary index	Local secondary index
Projected Attributes	With global secondary index queries or scans, you can only request the attributes that are projected into the index. DynamoDB does not fetch any attributes from the table.	If you query or scan a local secondary index, you can request attributes that are not projected in to the index. DynamoDB automatically fetches those attributes from the table.

If you want to create more than one table with secondary indexes, you must do so sequentially. For example, you would create the first table and wait for it to become ACTIVE, create the next table and wait for it to become ACTIVE, and so on. If you try to concurrently create more than one table with a secondary index, DynamoDB returns a `LimitExceededException`.

Each secondary index uses the same [table class](#) and [capacity mode](#) as the base table it is associated with. For each secondary index, you must specify the following:

- The type of index to be created – either a global secondary index or a local secondary index.
- A name for the index. The naming rules for indexes are the same as those for tables, as listed in [Service, account, and table quotas in Amazon DynamoDB](#). The name must be unique for the base table it is associated with, but you can use the same name for indexes that are associated with different base tables.
- The key schema for the index. Every attribute in the index key schema must be a top-level attribute of type String, Number, or Binary. Other data types, including documents and sets, are not allowed. Other requirements for the key schema depend on the type of index:
 - For a global secondary index, the partition key can be any scalar attribute of the base table. A sort key is optional, and it too can be any scalar attribute of the base table.
 - For a local secondary index, the partition key must be the same as the base table's partition key, and the sort key must be a non-key base table attribute.
- Additional attributes, if any, to project from the base table into the index. These attributes are in addition to the table's key attributes, which are automatically projected into every index. You can project attributes of any data type, including scalars, documents, and sets.
- The provisioned throughput settings for the index, if necessary:

- For a global secondary index, you must specify read and write capacity unit settings. These provisioned throughput settings are independent of the base table's settings.
- For a local secondary index, you do not need to specify read and write capacity unit settings. Any read and write operations on a local secondary index draw from the provisioned throughput settings of its base table.

For maximum query flexibility, you can create up to 20 global secondary indexes (default quota) and up to 5 local secondary indexes per table.

The quota of global secondary indexes per table is five for the following AWS Regions:

- AWS GovCloud (US-East)
- AWS GovCloud (US-West)
- Europe (Stockholm)

To get a detailed listing of secondary indexes on a table, use the `DescribeTable` operation. `DescribeTable` returns the name, storage size, and item counts for every secondary index on the table. These values are not updated in real time, but they are refreshed approximately every six hours.

You can access the data in a secondary index using either the `Query` or `Scan` operation. You must specify the name of the base table and the name of the index that you want to use, the attributes to be returned in the results, and any condition expressions or filters that you want to apply. DynamoDB can return the results in ascending or descending order.

When you delete a table, all of the indexes associated with that table are also deleted.

For best practices, see [Best practices for using secondary indexes in DynamoDB](#).

Using Global Secondary Indexes in DynamoDB

Some applications might need to perform many kinds of queries, using a variety of different attributes as query criteria. To support these requirements, you can create one or more *global secondary indexes* and issue `Query` requests against these indexes in Amazon DynamoDB.

Topics

- [Scenario: Using a Global Secondary Index](#)
- [Attribute projections](#)

- [Reading data from a Global Secondary Index](#)
- [Data synchronization between tables and Global Secondary Indexes](#)
- [Table classes with Global Secondary Index](#)
- [Provisioned throughput considerations for Global Secondary Indexes](#)
- [Storage considerations for Global Secondary Indexes](#)
- [Managing Global Secondary Indexes](#)
- [Working with Global Secondary Indexes: Java](#)
- [Working with Global Secondary Indexes: .NET](#)
- [Working with Global Secondary Indexes: AWS CLI](#)

Scenario: Using a Global Secondary Index

To illustrate, consider a table named GameScores that tracks users and scores for a mobile gaming application. Each item in GameScores is identified by a partition key (UserId) and a sort key (GameTitle). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)

GameScores

UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
"101"	"Galaxy Invaders"	5842	"2015-09-15:17:24:31"	21	72	...
"101"	"Meteor Blasters"	1000	"2015-10-22:23:18:01"	12	3	...
"101"	"Starship X"	24	"2015-08-31:13:14:21"	4	9	...
"102"	"Alien Adventure"	192	"2015-07-12:11:07:56"	32	192	...
"102"	"Galaxy Invaders"	0	"2015-09-18:07:33:42"	0	5	...
"103"	"Attack Ships"	3	"2015-10-19:01:13:24"	1	8	...
"103"	"Galaxy Invaders"	2317	"2015-09-11:06:53:00"	40	3	...
"103"	"Meteor Blasters"	723	"2015-10-19:01:13:24"	22	12	...
"103"	"Starship X"	42	"2015-07-11:06:53:00"	4	19	...
...

Now suppose that you wanted to write a leaderboard application to display top scores for each game. A query that specified the key attributes (`UserId` and `GameTitle`) would be very efficient. However, if the application needed to retrieve data from `GameScores` based on `GameTitle` only, it would need to use a `Scan` operation. As more items are added to the table, scans of all the data would become slow and inefficient. This makes it difficult to answer questions such as the following:

- What is the top score ever recorded for the game Meteor Blasters?
- Which user had the highest score for Galaxy Invaders?
- What was the highest ratio of wins vs. losses?

To speed up queries on non-key attributes, you can create a global secondary index. A global secondary index contains a selection of attributes from the base table, but they are organized by a primary key that is different from that of the table. The index key does not need to have any of the key attributes from the table. It doesn't even need to have the same key schema as a table.

For example, you could create a global secondary index named `GameTitleIndex`, with a partition key of `GameTitle` and a sort key of `TopScore`. The base table's primary key attributes are always projected into an index, so the `UserId` attribute is also present. The following diagram shows what `GameTitleIndex` index would look like.

GameTitleIndex

GameTitle	TopScore	UserId
"Alien Adventure"	192	"102"
"Attack Ships"	3	"103"
"Galaxy Invaders"	0	"102"
"Galaxy Invaders"	2317	"103"
"Galaxy Invaders"	5842	"101"
"Meteor Blasters"	723	"103"
"Meteor Blasters"	1000	"101"
"Starship X"	24	"101"
"Starship X"	42	"103"
...

Now you can query GameTitleIndex and easily obtain the scores for Meteor Blasters. The results are ordered by the sort key values, TopScore. If you set the ScanIndexForward parameter to false, the results are returned in descending order, so the highest score is returned first.

Every global secondary index must have a partition key, and can have an optional sort key. The index key schema can be different from the base table schema. You could have a table with a simple primary key (partition key), and create a global secondary index with a composite primary key (partition key and sort key)—or vice versa. The index key attributes can consist of any top-level String, Number, or Binary attributes from the base table. Other scalar types, document types, and set types are not allowed.

You can project other base table attributes into the index if you want. When you query the index, DynamoDB can retrieve these projected attributes efficiently. However, global secondary index queries cannot fetch attributes from the base table. For example, if you query GameTitleIndex as shown in the previous diagram, the query could not access any non-key attributes other than TopScore (although the key attributes GameTitle and UserId would automatically be projected).

In a DynamoDB table, each key value must be unique. However, the key values in a global secondary index do not need to be unique. To illustrate, suppose that a game named Comet Quest is especially difficult, with many new users trying but failing to get a score above zero. The following is some data that could represent this.

UserId	GameTitle	TopScore
123	Comet Quest	0
201	Comet Quest	0
301	Comet Quest	0

When this data is added to the GameScores table, DynamoDB propagates it to GameTitleIndex. If we then query the index using Comet Quest for GameTitle and 0 for TopScore, the following data is returned.

GameTitle	TopScore	Userid
"Comet Quest"	0	"123"
"Comet Quest"	0	"201"
"Comet Quest"	0	"301"

Only the items with the specified key values appear in the response. Within that set of data, the items are in no particular order.

A global secondary index only tracks data items where its key attributes actually exist. For example, suppose that you added another new item to the GameScores table, but only provided the required primary key attributes.

UserId	GameTitle
400	Comet Quest

Because you didn't specify the `TopScore` attribute, DynamoDB would not propagate this item to `GameTitleIndex`. Thus, if you queried `GameScores` for all the `Comet Quest` items, you would get the following four items.

UserId	GameTitle	TopScore
"123"	"Comet Quest"	0
"201"	"Comet Quest"	0
"301"	"Comet Quest"	0
"400"	"Comet Quest"	

A similar query on `GameTitleIndex` would still return three items, rather than four. This is because the item with the nonexistent `TopScore` is not propagated to the index.

GameTitle	TopScore	Userid
"Comet Quest"	0	"123"
"Comet Quest"	0	"201"
"Comet Quest"	0	"301"

Attribute projections

A *projection* is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, you need to specify the attributes that will be projected into the index. DynamoDB provides three different options for this:

- *KEYS_ONLY* – Each item in the index consists only of the table partition key and sort key values, plus the index key values. The *KEYS_ONLY* option results in the smallest possible secondary index.
- *INCLUDE* – In addition to the attributes described in *KEYS_ONLY*, the secondary index will include other non-key attributes that you specify.

- **ALL** – The secondary index includes all of the attributes from the source table. Because all of the table data is duplicated in the index, an ALL projection results in the largest possible secondary index.

In the previous diagram, GameTitleIndex has only one projected attribute: UserId. So while an application can efficiently determine the UserId of the top scorers for each game using GameTitle and TopScore in queries, it can't efficiently determine the highest ratio of wins vs. losses for the top scorers. To do so, it would have to perform an additional query on the base table to fetch the wins and losses for each of the top scorers. A more efficient way to support queries on this data would be to project these attributes from the base table into the global secondary index, as shown in this diagram.

GameTitleIndex

<i>GameTitle</i>	<i>TopScore</i>	<i>UserId</i>	<i>Wins</i>	<i>Losses</i>
“Alien Adventure”	192	“102”	32	192
“Attack Ships”	3	“103”	1	8
“Galaxy Invaders”	0	“102”	0	5
“Galaxy Invaders”	2317	“103”	40	3
“Galaxy Invaders”	5842	“101”	21	72
“Meteor Blasters”	723	“103”	22	12
“Meteor Blasters”	1000	“101”	12	3
“Starship X”	24	“101”	4	9
“Starship X”	42	“103”	4	19
***	***	***	***	***

Because the non-key attributes Wins and Losses are projected into the index, an application can determine the wins vs. losses ratio for any game, or for any combination of game and user ID.

When you choose the attributes to project into a global secondary index, you must consider the tradeoff between provisioned throughput costs and storage costs:

- If you need to access just a few attributes with the lowest possible latency, consider projecting only those attributes into a global secondary index. The smaller the index, the less that it costs to store it, and the less your write costs are.
- If your application frequently accesses some non-key attributes, you should consider projecting those attributes into a global secondary index. The additional storage costs for the global secondary index offset the cost of performing frequent table scans.
- If you need to access most of the non-key attributes on a frequent basis, you can project these attributes—or even the entire base table—into a global secondary index. This gives you maximum flexibility. However, your storage cost would increase, or even double.
- If your application needs to query a table infrequently, but must perform many writes or updates against the data in the table, consider projecting KEYS_ONLY. The global secondary index would be of minimal size, but would still be available when needed for query activity.

Reading data from a Global Secondary Index

You can retrieve items from a global secondary index using the Query and Scan operations. The GetItem and BatchGetItem operations can't be used on a global secondary index.

Querying a Global Secondary Index

You can use the Query operation to access one or more items in a global secondary index. The query must specify the name of the base table and the name of the index that you want to use, the attributes to be returned in the query results, and any query conditions that you want to apply. DynamoDB can return the results in ascending or descending order.

Consider the following data returned from a Query that requests gaming data for a leaderboard application.

```
{  
    "TableName": "GameScores",  
    "IndexName": "GameTitleIndex",  
    "KeyConditionExpression": "GameTitle = :v_title",  
    "ExpressionAttributeValues": {  
        ":v_title": {"S": "Meteor Blasters"}  
    "ProjectionExpression": "UserId, TopScore",  
    "ScanIndexForward": false  
}
```

In this query:

- DynamoDB accesses *GameTitleIndex*, using the *GameTitle* partition key to locate the index items for Meteor Blasters. All of the index items with this key are stored adjacent to each other for rapid retrieval.
- Within this game, DynamoDB uses the index to access all of the user IDs and top scores for this game.
- The results are returned, sorted in descending order because the *ScanIndexForward* parameter is set to false.

Scanning a Global Secondary Index

You can use the Scan operation to retrieve all of the data from a global secondary index. You must provide the base table name and the index name in the request. With a Scan, DynamoDB reads all of the data in the index and returns it to the application. You can also request that only some of the data be returned, and that the remaining data should be discarded. To do this, use the *FilterExpression* parameter of the Scan operation. For more information, see [Filter expressions for scan](#).

Data synchronization between tables and Global Secondary Indexes

DynamoDB automatically synchronizes each global secondary index with its base table. When an application writes or deletes items in a table, any global secondary indexes on that table are updated asynchronously, using an eventually consistent model. Applications never write directly to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations when changing read/write Capacity Mode](#).

When you create a global secondary index, you specify one or more index key attributes and their data types. This means that whenever you write an item to the base table, the data types for those attributes must match the index key schema's data types. In the case of *GameTitleIndex*, the *GameTitle* partition key in the index is defined as a *String* data type. The *TopScore* sort key in the index is of type *Number*. If you try to add an item to the *GameScores* table and specify a different data type for either *GameTitle* or *TopScore*, DynamoDB returns a *ValidationException* because of the data type mismatch.

When you put or delete items in a table, the global secondary indexes on that table are updated in an eventually consistent fashion. Changes to the table data are propagated to the global secondary indexes within a fraction of a second, under normal conditions. However, in some unlikely failure scenarios, longer propagation delays might occur. Because of this, your applications need to anticipate and handle situations where a query on a global secondary index returns results that are not up to date.

If you write an item to a table, you don't have to specify the attributes for any global secondary index sort key. Using GameTitleIndex as an example, you would not need to specify a value for the TopScore attribute to write a new item to the GameScores table. In this case, DynamoDB does not write any data to the index for this particular item.

A table with many global secondary indexes incurs higher costs for write activity than tables with fewer indexes. For more information, see [Provisioned throughput considerations for Global Secondary Indexes](#).

Table classes with Global Secondary Index

A global secondary index will always use the same table class as its base table. Any time a new global secondary index is added for a table, the new index will use the same table class as its base table. When a table's table class is updated, all associated global secondary indexes are updated as well.

Provisioned throughput considerations for Global Secondary Indexes

When you create a global secondary index on a provisioned mode table, you must specify read and write capacity units for the expected workload on that index. The provisioned throughput settings of a global secondary index are separate from those of its base table. A Query operation on a global secondary index consumes read capacity units from the index, not the base table. When you put, update or delete items in a table, the global secondary indexes on that table are also updated. These index updates consume write capacity units from the index, not from the base table.

For example, if you Query a global secondary index and exceed its provisioned read capacity, your request will be throttled. If you perform heavy write activity on the table, but a global secondary index on that table has insufficient write capacity, the write activity on the table will be throttled.

⚠ Important

To avoid potential throttling, the provisioned write capacity for a global secondary index should be equal or greater than the write capacity of the base table because new updates write to both the base table and global secondary index.

To view the provisioned throughput settings for a global secondary index, use the [DescribeTable](#) operation. Detailed information about all of the table's global secondary indexes is returned.

Read capacity units

Global secondary indexes support eventually consistent reads, each of which consume one half of a read capacity unit. This means that a single global secondary index query can retrieve up to 2×4 KB = 8 KB per read capacity unit.

For global secondary index queries, DynamoDB calculates the provisioned read activity in the same way as it does for queries against tables. The only difference is that the calculation is based on the sizes of the index entries, rather than the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned. The result is then rounded up to the next 4 KB boundary. For more information about how DynamoDB calculates provisioned throughput usage, see [Managing settings on DynamoDB provisioned capacity tables](#).

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all the attribute names and values across all of the items returned.

For example, consider a global secondary index where each item contains 2,000 bytes of data. Now suppose that you `Query` this index and that the query's `KeyConditionExpression` matches eight items. The total size of the matching items is $2,000 \text{ bytes} \times 8 \text{ items} = 16,000 \text{ bytes}$. This result is then rounded up to the nearest 4 KB boundary. Because global secondary index queries are eventually consistent, the total cost is $0.5 \times (16 \text{ KB} / 4 \text{ KB})$, or 2 read capacity units.

Write capacity units

When an item in a table is added, updated, or deleted, and a global secondary index is affected by this, the global secondary index consumes provisioned write capacity units for the operation. The total provisioned throughput cost for a write consists of the sum of write capacity units consumed by writing to the base table and those consumed by updating the global secondary indexes. If a

write to a table does not require a global secondary index update, no write capacity is consumed from the index.

For a table write to succeed, the provisioned throughput settings for the table and all of its global secondary indexes must have enough write capacity to accommodate the write. Otherwise, the write to the table is throttled.

The cost of writing an item to a global secondary index depends on several factors:

- If you write a new item to the table that defines an indexed attribute, or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required, one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.
- If an update to the table only changes the value of projected attributes in the index key schema, but does not change the value of any indexed key attribute, one write is required to update the values of the projected attributes into the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1 KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries will need to return and projecting only those attributes into the index.

Storage considerations for Global Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any global secondary indexes in which those attributes should appear. Your AWS account is charged for storage of the item in the base table and also for storage of attributes in any global secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- The size in bytes of the base table primary key (partition key and sort key)

- The size in bytes of the index key attribute
- The size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a global secondary index, you can estimate the average size of an item in the index and then multiply by the number of items in the base table that have the global secondary index key attributes.

If a table contains an item where a particular attribute is not defined, but that attribute is defined as an index partition key or sort key, DynamoDB doesn't write any data for that item to the index.

Managing Global Secondary Indexes

This section describes how to create, modify, and delete global secondary indexes in Amazon DynamoDB.

Topics

- [Creating a table with Global Secondary Indexes](#)
- [Describing the Global Secondary Indexes on a table](#)
- [Adding a Global Secondary Index to an existing table](#)
- [Deleting a Global Secondary Index](#)
- [Modifying a Global Secondary Index during creation](#)
- [Detecting and correcting index key violations](#)

Creating a table with Global Secondary Indexes

To create a table with one or more global secondary indexes, use the `CreateTable` operation with the `GlobalSecondaryIndexes` parameter. For maximum query flexibility, you can create up to 20 global secondary indexes (default quota) per table.

You must specify one attribute to act as the index partition key. You can optionally specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the same as a key attribute in the table. For example, in the *GameScores* table (see [Using Global Secondary Indexes in DynamoDB](#)), neither `TopScore` nor `TopScoreDateTime` are key attributes. You could create a global secondary index with a partition key of `TopScore` and a sort key of

TopScoreDateTime. You might use such an index to determine whether there is a correlation between high scores and the time of day a game is played.

Each index key attribute must be a scalar of type String, Number, or Binary. (It cannot be a document or a set.) You can project attributes of any data type into a global secondary index. This includes scalars, documents, and sets. For a complete list of data types, see [Data types](#).

If using provisioned mode, you must provide ProvisionedThroughput settings for the index, consisting of ReadCapacityUnits and WriteCapacityUnits. These provisioned throughput settings are separate from those of the table, but behave in similar ways. For more information, see [Provisioned throughput considerations for Global Secondary Indexes](#).

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations when changing read/write Capacity Mode](#).

Note

Backfill operations and ongoing write operations share write throughput within the global secondary index. When creating a new GSI, it can be important to check if your choice of partition key is producing uneven or narrowed distribution of data or traffic across the new index's partition key values. If this occurs, you could be seeing backfill and write operations occurring at the same time and throttling writes to the base table. The service takes measures to minimize the potential for this scenario, but has no insight into the shape of customer data with respect to the index partition key, the chosen projection, or the sparseness of the index primary key.

If you suspect that your new global secondary index might have narrow or skewed data or traffic distribution across partition key values, consider the following before adding new indexes to operationally important tables.

- It might be safest to add the index at a time when your application is driving the least amount of traffic.
- Consider enabling CloudWatch Contributor Insights on your base table and indexes. This will give you valuable insight into your traffic distribution.
- For provisioned capacity mode base tables and indexes, set the provisioned write capacity of your new index to at least double that of your base table. Watch WriteThrottleEvents, ThrottledRequests, OnlineIndexPercentageProgress, OnlineIndexConsumedWriteCapacity and OnlineIndexThrottleEvents CloudWatch metrics throughout the process. Adjust the provisioned write capacity as

- required to complete the backfill in a reasonable time without any significant throttling effects on your ongoing operations.
- Be prepared to cancel the index creation if you experience operational impact due to write throttling, and increasing the provisioned write capacity in your new GSI does not resolve it.

Describing the Global Secondary Indexes on a table

To view the status of all the global secondary indexes on a table, use the `DescribeTable` operation. The `GlobalSecondaryIndexes` portion of the response shows all of the indexes on the table, along with the current status of each (`IndexStatus`).

The `IndexStatus` for a global secondary index will be one of the following:

- `CREATING` — The index is currently being created, and is not yet available for use.
- `ACTIVE` — The index is ready for use, and applications can perform `Query` operations on the index.
- `UPDATING` — The provisioned throughput settings of the index are being changed.
- `DELETING` — The index is currently being deleted, and can no longer be used.

When DynamoDB has finished building a global secondary index, the index status changes from `CREATING` to `ACTIVE`.

Adding a Global Secondary Index to an existing table

To add a global secondary index to an existing table, use the `UpdateTable` operation with the `GlobalSecondaryIndexUpdates` parameter. You must provide the following:

- An index name. The name must be unique among all the indexes on the table.
- The key schema of the index. You must specify one attribute for the index partition key; you can optionally specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the same as a key attribute in the table. The data types for each schema attribute must be scalar: `String`, `Number`, or `Binary`.
- The attributes to be projected from the table into the index:
 - `KEYS_ONLY` — Each item in the index consists only of the table partition key and sort key values, plus the index key values.

- **INCLUDE** — In addition to the attributes described in KEYS_ONLY, the secondary index includes other non-key attributes that you specify.
- **ALL** — The index includes all of the attributes from the source table.
- The provisioned throughput settings for the index, consisting of ReadCapacityUnits and WriteCapacityUnits. These provisioned throughput settings are separate from those of the table.

You can only create one global secondary index per `UpdateTable` operation.

Phases of index creation

When you add a new global secondary index to an existing table, the table continues to be available while the index is being built. However, the new index is not available for Query operations until its status changes from CREATING to ACTIVE.

Note

Global secondary index creation does not use Application Auto Scaling. Increasing the MIN Application Auto Scaling capacity will not decrease the creation time of the global secondary index.

Behind the scenes, DynamoDB builds the index in two phases:

Resource Allocation

DynamoDB allocates the compute and storage resources that are needed for building the index.

During the resource allocation phase, the `IndexStatus` attribute is CREATING and the `Backfilling` attribute is false. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is in the resource allocation phase, you can't delete the index or delete its parent table. You also can't modify the provisioned throughput of the index or the table. You cannot add or delete other indexes on the table. However, you can modify the provisioned throughput of these other indexes.

Backfilling

For each item in the table, DynamoDB determines which set of attributes to write to the index based on its projection (KEYS_ONLY, INCLUDE, or ALL). It then writes these attributes to the index. During the backfill phase, DynamoDB tracks the items that are being added, deleted, or updated in the table. The attributes from these items are also added, deleted, or updated in the index as appropriate.

During the backfilling phase, the `IndexStatus` attribute is set to CREATING, and the `Backfilling` attribute is true. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is backfilling, you cannot delete its parent table. However, you can still delete the index or modify the provisioned throughput of the table and any of its global secondary indexes.

Note

During the backfilling phase, some writes of violating items might succeed while others are rejected. After backfilling, all writes to items that violate the new index's key schema are rejected. We recommend that you run the Violation Detector tool after the backfill phase finishes to detect and resolve any key violations that might have occurred. For more information, see [Detecting and correcting index key violations](#).

While the resource allocation and backfilling phases are in progress, the index is in the CREATING state. During this time, DynamoDB performs read operations on the table. You are not charged for read operations from the base table to populate the global secondary index. However, you are charged for write operations to populate the newly created global secondary index.

When the index build is complete, its status changes to ACTIVE. You can't Query or Scan the index until it is ACTIVE.

Note

In some cases, DynamoDB can't write data from the table to the index because of index key violations. This can occur if:

- The data type of an attribute value does not match the data type of an index key schema data type.

- The size of an attribute exceeds the maximum length for an index key attribute.
- An index key attribute has an empty String or empty Binary attribute value.

Index key violations do not interfere with global secondary index creation. However, when the index becomes ACTIVE, the violating keys are not present in the index.

DynamoDB provides a standalone tool for finding and resolving these issues. For more information, see [Detecting and correcting index key violations](#).

Adding a Global Secondary Index to a large table

The time required for building a global secondary index depends on several factors, such as the following:

- The size of the table
- The number of items in the table that qualify for inclusion in the index
- The number of attributes projected into the index
- The provisioned write capacity of the index
- Write activity on the main table during index builds

If you are adding a global secondary index to a very large table, it might take a long time for the creation process to complete. To monitor progress and determine whether the index has sufficient write capacity, consult the following Amazon CloudWatch metrics:

- `OnlineIndexPercentageProgress`
- `OnlineIndexConsumedWriteCapacity`
- `OnlineIndexThrottleEvents`

Note

For more information about CloudWatch metrics related to DynamoDB, see [DynamoDB metrics](#).

If the provisioned write throughput setting on the index is too low, the index build will take longer to complete. To shorten the time it takes to build a new global secondary index, you can increase its provisioned write capacity temporarily.

Note

As a general rule, we recommend setting the provisioned write capacity of the index to 1.5 times the write capacity of the table. This is a good setting for many use cases. However, your actual requirements might be higher or lower.

While an index is being backfilled, DynamoDB uses internal system capacity to read from the table. This is to minimize the impact of the index creation and to assure that your table does not run out of read capacity.

However, it is possible that the volume of incoming write activity might exceed the provisioned write capacity of the index. This is a bottleneck scenario, in which the index creation takes more time because the write activity to the index is throttled. During the index build, we recommend that you monitor the Amazon CloudWatch metrics for the index to determine whether its consumed write capacity is exceeding its provisioned capacity. In a bottleneck scenario, you should increase the provisioned write capacity on the index to avoid write throttling during the backfill phase.

After the index has been created, you should set its provisioned write capacity to reflect the normal usage of your application.

Deleting a Global Secondary Index

If you no longer need a global secondary index, you can delete it using the `UpdateTable` operation.

You can delete only one global secondary index per `UpdateTable` operation.

While the global secondary index is being deleted, there is no effect on any read or write activity in the parent table. While the deletion is in progress, you can still modify the provisioned throughput on other indexes.

Note

- When you delete a table using the DeleteTable action, all of the global secondary indexes on that table are also deleted.
- Your account will not be charged for the delete operation of the global secondary index.

Modifying a Global Secondary Index during creation

While an index is being built, you can use the `DescribeTable` operation to determine what phase it is in. The description for the index includes a Boolean attribute, `Backfilling`, to indicate whether DynamoDB is currently loading the index with items from the table. If `Backfilling` is true, the resource allocation phase is complete and the index is now backfilling.

While the backfill is proceeding, you can update the provisioned throughput parameters for the index. You might decide to do this in order to speed up the index build: You can increase the write capacity of the index while it is being built, and then decrease it afterward. To modify the provisioned throughput settings of the index, use the `UpdateTable` operation. The index status changes to `UPDATING`, and `Backfilling` is true until the index is ready for use.

During the backfilling phase, you can delete the index that is being created. During this phase, you can't add or delete other indexes on the table.

Note

For indexes that were created as part of a `CreateTable` operation, the `Backfilling` attribute does not appear in the `DescribeTable` output. For more information, see [Phases of index creation](#).

Detecting and correcting index key violations

During the backfill phase of global secondary index creation, Amazon DynamoDB examines each item in the table to determine whether it is eligible for inclusion in the index. Some items might not be eligible because they would cause index key violations. In these cases, the items remain in the table, but the index doesn't have a corresponding entry for that item.

An *index key violation* occurs in the following situations:

- There is a data type mismatch between an attribute value and the index key schema data type. For example, suppose that one of the items in the GameScores table had a TopScore value of type String. If you added a global secondary index with a partition key of TopScore, of type Number, the item from the table would violate the index key.
- An attribute value from the table exceeds the maximum length for an index key attribute. The maximum length of a partition key is 2048 bytes, and the maximum length of a sort key is 1024 bytes. If any of the corresponding attribute values in the table exceed these limits, the item from the table would violate the index key.

 **Note**

If a String or Binary attribute value is set for an attribute that is used as an index key, then the attribute value must have a length greater than zero; otherwise, the item from the table would violate the index key.

This tool does not flag this index key violation, at this time.

If an index key violation occurs, the backfill phase continues without interruption. However, any violating items are not included in the index. After the backfill phase completes, all writes to items that violate the new index's key schema will be rejected.

To identify and fix attribute values in a table that violate an index key, use the Violation Detector tool. To run Violation Detector, you create a configuration file that specifies the name of a table to be scanned, the names and data types of the global secondary index partition key and sort key, and what actions to take if any index key violations are found. Violation Detector can run in one of two different modes:

- **Detection mode** — Detect index key violations. Use detection mode to report the items in the table that would cause key violations in a global secondary index. (You can optionally request that these violating table items be deleted immediately when they are found.) The output from detection mode is written to a file, which you can use for further analysis.
- **Correction mode** — Correct index key violations. In correction mode, Violation Detector reads an input file with the same format as the output file from detection mode. Correction mode reads the records from the input file and, for each record, it either deletes or updates the corresponding items in the table. (Note that if you choose to update the items, you must edit the input file and set appropriate values for these updates.)

Downloading and running Violation Detector

Violation Detector is available as an executable Java Archive (.jar file), and runs on Windows, macOS, or Linux computers. Violation Detector requires Java 1.7 (or later) and Apache Maven.

- [Download violation detector from GitHub](#)

Follow the instructions in the README.md file to download and install Violation Detector using Maven.

To start Violation Detector, go to the directory where you have built `ViolationDetector.java` and enter the following command.

```
java -jar ViolationDetector.jar [options]
```

The Violation Detector command line accepts the following options:

- `-h | --help` — Prints a usage summary and options for Violation Detector.
- `-p | --configFilePath value` — The fully qualified name of a Violation Detector configuration file. For more information, see [The Violation Detector configuration file](#).
- `-t | --detect value` — Detect index key violations in the table, and write them to the Violation Detector output file. If the value of this parameter is set to `keep`, items with key violations are not modified. If the value is set to `delete`, items with key violations are deleted from the table.
- `-c | --correct value` — Read index key violations from an input file, and take corrective actions on the items in the table. If the value of this parameter is set to `update`, items with key violations are updated with new, non-violating values. If the value is set to `delete`, items with key violations are deleted from the table.

The Violation Detector configuration file

At runtime, the Violation Detector tool requires a configuration file. The parameters in this file determine which DynamoDB resources that Violation Detector can access, and how much provisioned throughput it can consume. The following table describes these parameters.

Parameter name	Description	Required?
awsCredentialsFile	The fully qualified name of a file containing your AWS credentials. The credentials file must be in the following format: <pre>accessKey = access_key y_id_goes_here secretKey = secret_key y_goes_here</pre>	Yes
dynamoDBRegion	The AWS Region in which the table resides. For example: us-west-2 .	Yes
tableName	The name of the DynamoDB table to be scanned.	Yes
gsiHashKeyName	The name of the index partition key.	Yes
gsiHashKeyType	The data type of the index partition key—String, Number, or Binary: S N B	Yes
gsiRangeKeyName	The name of the index sort key. Do not specify this parameter if the index only has a simple primary key (partition key).	No
gsiRangeKeyType	The data type of the index sort key—String, Number, or Binary:	No

Parameter name	Description	Required?
	S N B Do not specify this parameter if the index only has a simple primary key (partition key).	
recordDetails	Whether to write the full details of index key violations to the output file. If set to true (the default), full information about the violating items is reported. If set to false, only the number of violations is reported.	No
recordGsiValueInViolationRecord	Whether to write the values of the violating index keys to the output file. If set to true (default), the key values are reported. If set to false, the key values are not reported.	No

Parameter name	Description	Required?
detectionOutputPath	<p>The full path of the Violation Detector output file. This parameter supports writing to a local directory or to Amazon Simple Storage Service (Amazon S3). The following are examples:</p> <pre>detectionOutputPath = //local/path/<i>filename.csv</i> detection OutputPath = s3://<i>bucket/filename.csv</i></pre> <p>Information in the output file appears in comma-separated values (CSV) format. If you don't set <code>detection OutputPath</code>, the output file is named <code>violation_detection.csv</code> and is written to your current working directory.</p>	No

Parameter name	Description	Required?
numOfSegments	<p>The number of parallel scan segments to be used when Violation Detector scans the table. The default value is 1, meaning that the table is scanned in a sequential manner. If the value is 2 or higher, then Violation Detector divides the table into that many logical segments and an equal number of scan threads.</p> <p>The maximum setting for <code>numOfSegments</code> is 4096.</p> <p>For larger tables, a parallel scan is generally faster than a sequential scan. In addition, if the table is large enough to span multiple partitions, a parallel scan distributes its read activity evenly across multiple partitions.</p> <p>For more information about parallel scans in DynamoDB, see Parallel scan.</p>	No

Parameter name	Description	Required?
numOfViolations	<p>The upper limit of index key violations to write to the output file. If set to -1 (the default), the entire table is scanned. If set to a positive integer, then Violation Detector stops after it encounters that number of violations.</p>	No
numOfRecords	<p>The number of items in the table to be scanned. If set to -1 (the default), the entire table is scanned. If set to a positive integer, Violation Detector stops after it scans that many items in the table.</p>	No
readWriteIOPSPercent	<p>Regulates the percentage of provisioned read capacity units that are consumed during the table scan. Valid values range from 1 to 100. The default value (25) means that Violation Detector will consume no more than 25% of the table's provisioned read throughput.</p>	No

Parameter name	Description	Required?
correctionInputPath	<p>The full path of the Violation Detector correction input file. If you run Violation Detector in correction mode, the contents of this file are used to modify or delete data items in the table that violate the global secondary index.</p> <p>The format of the <code>correctionInputPath</code> file is the same as that of the <code>detectionOutputPath</code> file. This lets you process the output from detection mode as input in correction mode.</p>	No

Parameter name	Description	Required?
	<p>correctionOutputPath</p> <p>The full path of the Violation Detector correction output file. This file is created only if there are update errors.</p> <p>This parameter supports writing to a local directory or to Amazon S3. The following are examples:</p> <pre>correctionOutputPath = //<i>local/path/</i> <i>filename.csv</i></pre> <pre>correctionOutputPath = s3://<i>bucket/filename.csv</i></pre> <p>Information in the output file appears in CSV format. If you don't set <code>correctionOutputPath</code>, the output file is named <code>violation_update_errors.csv</code> and is written to your current working directory.</p>	No

Detection

To detect index key violations, use Violation Detector with the `--detect` command line option. To show how this option works, consider the `ProductCatalog` table shown in [Creating tables and loading data for code examples in DynamoDB](#). The following is a list of items in the table. Only the primary key (`Id`) and the `Price` attribute are shown.

Id (primary key)	Price
101	5
102	20
103	200
201	100
202	200
203	300
204	400
205	500

All of the values for Price are of type Number. However, because DynamoDB is schemaless, it is possible to add an item with a non-numeric Price. For example, suppose that you add another item to the ProductCatalog table.

Id (primary key)	Price
999	"Hello"

The table now has a total of nine items.

Now you add a new global secondary index to the table: PriceIndex. The primary key for this index is a partition key, Price, which is of type Number. After the index has been built, it will contain eight items—but the ProductCatalog table has nine items. The reason for this discrepancy is that the value "Hello" is of type String, but PriceIndex has a primary key of type Number. The String value violates the global secondary index key, so it is not present in the index.

To use Violation Detector in this scenario, you first create a configuration file such as the following.

```
# Properties file for violation detection tool configuration.  
# Parameters that are not specified will use default values.  
  
awsCredentialsFile = /home/alice/credentials.txt  
dynamoDBRegion = us-west-2  
tableName = ProductCatalog  
gsiHashKeyName = Price  
gsiHashKeyType = N  
recordDetails = true  
recordGsiValueInViolationRecord = true  
detectionOutputPath = ./gsiViolationCheck.csv  
correctionInputPath = ./gsiViolationCheck.csv  
numOfSegments = 1  
readWriteIOPSPercent = 40
```

Next, you run Violation Detector as in the following example.

```
$ java -jar ViolationDetector.jar --configFilePath config.txt --detect keep  
  
Violation detection started: sequential scan, Table name: ProductCatalog, GSI name:  
PriceIndex  
Progress: Items scanned in total: 9, Items scanned by this thread: 9, Violations  
found by this thread: 1, Violations deleted by this thread: 0  
Violation detection finished: Records scanned: 9, Violations found: 1, Violations  
deleted: 0, see results at: ./gsiViolationCheck.csv
```

If the `recordDetails` config parameter is set to `true`, Violation Detector writes details of each violation to the output file, as in the following example.

```
Table Hash Key,GSI Hash Key Value,GSI Hash Key Violation Type,GSI Hash Key Violation  
Description,GSI Hash Key Update Value(FOR USER),Delete Blank Attributes When Updating?  
(Y/N)  
  
999,"{""S"":""Hello""}",Type Violation,Expected: N Found: S,,
```

The output file is in CSV format. The first line in the file is a header, followed by one record per item that violates the index key. The fields of these violation records are as follows:

- **Table hash key** — The partition key value of the item in the table.
- **Table range key** — The sort key value of the item in the table.
- **GSI hash key value** — The partition key value of the global secondary index.

- **GSI hash key violation type** — Either Type Violation or Size Violation.
- **GSI hash key violation description** — The cause of the violation.
- **GSI hash key update Value(FOR USER)** — In correction mode, a new user-supplied value for the attribute.
- **GSI range key value** — The sort key value of the global secondary index.
- **GSI range key violation type** — Either Type Violation or Size Violation.
- **GSI range key violation description** — The cause of the violation.
- **GSI range key update Value(FOR USER)** — In correction mode, a new user-supplied value for the attribute.
- **Delete blank attribute when Updating(Y/N)** — In correction mode, determines whether to delete (Y) or keep (N) the violating item in the table—but only if either of the following fields are blank:
 - GSI Hash Key Update Value(FOR USER)
 - GSI Range Key Update Value(FOR USER)

If either of these fields are non-blank, then Delete Blank Attribute When Updating(Y/N) has no effect.

 **Note**

The output format might vary, depending on the configuration file and command line options. For example, if the table has a simple primary key (without a sort key), no sort key fields will be present in the output.

The violation records in the file might not be in sorted order.

Correction

To correct index key violations, use Violation Detector with the `--correct` command line option. In correction mode, Violation Detector reads the input file specified by the `correctionInputPath` parameter. This file has the same format as the `detectionOutputPath` file, so that you can use the output from detection as input for correction.

Violation Detector provides two different ways to correct index key violations:

- **Delete violations** — Delete the table items that have violating attribute values.

- **Update violations** — Update the table items, replacing the violating attributes with non-violating values.

In either case, you can use the output file from detection mode as input for correction mode.

Continuing with the ProductCatalog example, suppose that you want to delete the violating item from the table. To do this, you use the following command line.

```
$ java -jar ViolationDetector.jar --configFilePath config.txt --correct delete
```

At this point, you are asked to confirm whether you want to delete the violating items.

```
Are you sure to delete all violations on the table?y/n  
y  
Confirmed, will delete violations on the table...  
Violation correction from file started: Reading records from file: ./  
gsiViolationCheck.csv, will delete these records from table.  
Violation correction from file finished: Violations delete: 1, Violations Update: 0
```

Now both ProductCatalog and PriceIndex have the same number of items.

Working with Global Secondary Indexes: Java

You can use the AWS SDK for Java Document API to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes.

The following are the common steps for table operations.

1. Create an instance of the DynamoDB class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.
3. Call the appropriate method provided by the client that you created in the preceding step.

Topics

- [Create a table with a Global Secondary Index](#)
- [Describe a table with a Global Secondary Index](#)
- [Query a Global Secondary Index](#)

- [Example: Global Secondary Indexes using the AWS SDK for Java document API](#)

Create a table with a Global Secondary Index

You can create global secondary indexes at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more global secondary indexes. The following Java code example creates a table to hold information about weather data. The partition key is `Location` and the sort key is `Date`. A global secondary index named `PrecipIndex` allows fast access to precipitation data for various locations.

The following are the steps to create a table with a global secondary index, using the DynamoDB document API.

1. Create an instance of the DynamoDB class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the global secondary index, you must provide the index name, its provisioned throughput settings, the attribute definitions for the index sort key, the key schema for the index, and the attribute projection.

3. Call the `createTable` method by providing the request object as a parameter.

The following Java code example demonstrates the preceding steps. The code creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index partition key is `Date` and its sort key is `Precipitation`. All of the table attributes are projected into the index. Users can query this index to obtain weather data for a particular date, optionally sorting the data by precipitation amount.

Because `Precipitation` is not a key attribute for the table, it is not required. However, `WeatherData` items without `Precipitation` do not appear in `PrecipIndex`.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

// Attribute definitions
ArrayList<AttributeDefinition> attributeDefinitions = new
    ArrayList<AttributeDefinition>();

attributeDefinitions.add(new AttributeDefinition()
```

```
.withAttributeName("Location")
.withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
.withAttributeName("Date")
.withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
.withAttributeName("Precipitation")
.withAttributeType("N"));

// Table key schema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new KeySchemaElement()
.withAttributeName("Location")
.withKeyType(KeyType.HASH)); //Partition key
tableKeySchema.add(new KeySchemaElement()
.withAttributeName("Date")
.withKeyType(KeyType.RANGE)); //Sort key

// PrecipIndex
GlobalSecondaryIndex precipIndex = new GlobalSecondaryIndex()
.withIndexName("PrecipIndex")
.withProvisionedThroughput(new ProvisionedThroughput()
.withReadCapacityUnits((long) 10)
.withWriteCapacityUnits((long) 1))
.withProjection(new Projection().withProjectionType(ProjectionType.ALL));

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();

indexKeySchema.add(new KeySchemaElement()
.withAttributeName("Date")
.withKeyType(KeyType.HASH)); //Partition key
indexKeySchema.add(new KeySchemaElement()
.withAttributeName("Precipitation")
.withKeyType(KeyType.RANGE)); //Sort key

precipIndex.setKeySchema(indexKeySchema);

CreateTableRequest createTableRequest = new CreateTableRequest()
.withTableName("WeatherData")
.withProvisionedThroughput(new ProvisionedThroughput()
.withReadCapacityUnits((long) 5)
.withWriteCapacityUnits((long) 1))
.withAttributeDefinitions(attributeDefinitions)
.withKeySchema(tableKeySchema)
```

```
.withGlobalSecondaryIndexes(precipIndex);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a table with a Global Secondary Index

To get information about global secondary indexes on a table, use `DescribeTable`. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access global secondary index information a table.

1. Create an instance of the DynamoDB class.
2. Create an instance of the Table class to represent the index you want to work with.
3. Call the `describe` method on the Table object.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
TableDescription tableDesc = table.describe();

Iterator<GlobalSecondaryIndexDescription> gsiIter =
    tableDesc.getGlobalSecondaryIndexes().iterator();
while (gsiIter.hasNext()) {
    GlobalSecondaryIndexDescription gsiDesc = gsiIter.next();
    System.out.println("Info for index "
        + gsiDesc.getIndexName() + ":");

    Iterator<KeySchemaElement> kseIter = gsiDesc.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
}
```

```
Projection projection = gsiDesc.getProjection();
System.out.println("\tThe projection type is: "
    + projection.getProjectionType());
if (projection.getProjectionType().toString().equals("INCLUDE")) {
    System.out.println("\t\tThe non-key projected attributes are: "
        + projection.getNonKeyAttributes());
}
}
```

Query a Global Secondary Index

You can use `Query` on a global secondary index, in much the same way you `Query` a table. You need to specify the index name, the query criteria for the index partition key and sort key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a partition key of `Date` and a sort key of `Precipitation`. The index query returns all of the weather data for a particular date, where the precipitation is greater than zero.

The following are the steps to query a global secondary index using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the index you want to work with.
3. Create an instance of the `Index` class for the index you want to query.
4. Call the `query` method on the `Index` object.

The attribute name `Date` is a `DynamoDB` reserved word. Therefore, you must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
Index index = table.getIndex("PrecipIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("#d = :v_date and Precipitation = :v_precip")
    .withNameMap(new NameMap()
```

```
.with("#d", "Date")
.withValueMap(new ValueMap()
    .withString(":v_date", "2013-08-10")
    .withNumber(":v_precip", 0));

ItemCollection<QueryOutcome> items = index.query(spec);
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next().toJSONPretty());
}
```

Example: Global Secondary Indexes using the AWS SDK for Java document API

The following Java code example shows how to work with global secondary indexes. The example creates a table named `Issues`, which might be used in a simple bug tracking system for software development. The partition key is `IssueId` and the sort key is `Title`. There are three global secondary indexes on this table:

- `CreateDateIndex` — The partition key is `CreateDate` and the sort key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- `TitleIndex` — The partition key is `Title` and the sort key is `IssueId`. No attributes other than the table keys are projected into the index.
- `DueDateIndex` — The partition key is `DueDate`, and there is no sort key. All of the table attributes are projected into the index.

After the `Issues` table is created, the program loads the table with data representing software bug reports. It then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions for testing the following example, see [Java code examples](#).

Example

```
package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.GlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class DocumentAPIGlobalSecondaryIndexExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    public static String tableName = "Issues";

    public static void main(String[] args) throws Exception {

        createTable();
        loadData();

        queryIndex("CreateDateIndex");
        queryIndex("TitleIndex");
        queryIndex("DueDateIndex");

        deleteTable(tableName);
    }

    public static void createTable() {

        // Attribute definitions
        ArrayList<AttributeDefinition> attributeDefinitions = new
        ArrayList<AttributeDefinition>();
    }
}
```

```
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("IssueId").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Title").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("CreateDate").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("DueDate").withAttributeType("S"));

        // Key schema for table
        ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.HASH)); // Partition

        // key
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.RANGE)); // Sort

        // key

        // Initial provisioned throughput settings for the indexes
        ProvisionedThroughput ptIndex = new
ProvisionedThroughput().withReadCapacityUnits(1L)
                .withWriteCapacityUnits(1L);

        // CreateDateIndex
        GlobalSecondaryIndex createDateIndex = new
GlobalSecondaryIndex().withIndexName("CreateDateIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("CreateDate").withKeyType(KeyType.HASH), // Partition

                // key
                new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

        // key
        .withProjection(
                new
Projection().withProjectionType("INCLUDE").withNonKeyAttributes("Description",
"Status"));
```

```
// TitleIndex
GlobalSecondaryIndex titleIndex = new
GlobalSecondaryIndex().withIndexName("TitleIndex")
    .withProvisionedThroughput(ptIndex)
    .withKeySchema(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.HASH), // Partition

        // key
        new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

        // key
        .withProjection(new Projection().withProjectionType("KEYS_ONLY"));

// DueDateIndex
GlobalSecondaryIndex dueDateIndex = new
GlobalSecondaryIndex().withIndexName("DueDateIndex")
    .withProvisionedThroughput(ptIndex)
    .withKeySchema(new
KeySchemaElement().withAttributeName("DueDate").withKeyType(KeyType.HASH)) // Partition

        // key
        .withProjection(new Projection().withProjectionType("ALL"));

CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
    .withProvisionedThroughput(
        new ProvisionedThroughput().withReadCapacityUnits((long)
1).withWriteCapacityUnits((long) 1))

    .withAttributeDefinitions(attributeDefinitions).withKeySchema(tableKeySchema)
        .withGlobalSecondaryIndexes(createDateIndex, titleIndex, dueDateIndex);

System.out.println("Creating table " + tableName + "...");
dynamoDB.createTable(createTableRequest);

// Wait for table to become active
System.out.println("Waiting for " + tableName + " to become ACTIVE...");
try {
    Table table = dynamoDB.getTable(tableName);
    table.waitForActive();
} catch (InterruptedException e) {
    e.printStackTrace();
```

```
    }

}

public static void queryIndex(String indexName) {

    Table table = dynamoDB.getTable(tableName);

System.out.println("\n*****\n");
    System.out.print("Querying index " + indexName + "...");

    Index index = table.getIndex(indexName);

    ItemCollection<QueryOutcome> items = null;

    QuerySpec querySpec = new QuerySpec();

    if (indexName == "CreateDateIndex") {
        System.out.println("Issues filed on 2013-11-01");
        querySpec.withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
            .withValueMap(new ValueMap().withString(":v_date",
"2013-11-01").withString(":v_issue", "A-"));
        items = index.query(querySpec);
    } else if (indexName == "TitleIndex") {
        System.out.println("Compilation errors");
        querySpec.withKeyConditionExpression("Title = :v_title and
begins_with(IssueId, :v_issue)")
            .withValueMap(
                new ValueMap().withString(":v_title", "Compilation
error").withString(":v_issue", "A-"));
        items = index.query(querySpec);
    } else if (indexName == "DueDateIndex") {
        System.out.println("Items that are due on 2013-11-30");
        querySpec.withKeyConditionExpression("DueDate = :v_date")
            .withValueMap(new ValueMap().withString(":v_date", "2013-11-30"));
        items = index.query(querySpec);
    } else {
        System.out.println("\nNo valid index name provided");
        return;
    }

    Iterator<Item> iterator = items.iterator();
```

```
System.out.println("Query: printing results...");

while (iterator.hasNext()) {
    System.out.println(iterator.next().toJSONPretty());
}

}

public static void deleteTable(String tableName) {

    System.out.println("Deleting table " + tableName + "...");

    Table table = dynamoDB.getTable(tableName);
    table.delete();

    // Wait for table to be deleted
    System.out.println("Waiting for " + tableName + " to be deleted...");
    try {
        table.waitForDelete();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void loadData() {

    System.out.println("Loading data into table " + tableName + "...");

    // IssueId, Title,
    // Description,
    // CreateDate, LastUpdateDate, DueDate,
    // Priority, Status

    putItem("A-101", "Compilation error", "Can't compile Project X - bad version
number. What does this mean?",
           "2013-11-01", "2013-11-02", "2013-11-10", 1, "Assigned");

    putItem("A-102", "Can't read data file", "The main data file is missing, or the
permissions are incorrect",
           "2013-11-01", "2013-11-04", "2013-11-30", 2, "In progress");

    putItem("A-103", "Test failure", "Functional test of Project X produces
errors", "2013-11-01", "2013-11-02",
           "2013-11-10", 1, "In progress");
}
```

```
        putItem("A-104", "Compilation error", "Variable 'messageCount' was not
initialized.", "2013-11-15",
        "2013-11-16", "2013-11-30", 3, "Assigned");

        putItem("A-105", "Network issue", "Can't ping IP address 127.0.0.1. Please fix
this.", "2013-11-15",
        "2013-11-16", "2013-11-19", 5, "Assigned");

    }

public static void putItem(
    String issueId, String title, String description, String createDate, String
lastUpdateDate, String dueDate,
    Integer priority, String status) {

    Table table = dynamoDB.getTable(tableName);

    Item item = new Item().withPrimaryKey("IssueId", issueId).withString("Title",
title)
        .withString("Description", description).withString("CreateDate",
createDate)
        .withString("LastUpdateDate", lastUpdateDate).withString("DueDate",
dueDate)
        .withNumber("Priority", priority).withString("Status", status);

    table.putItem(item);
}

}
```

Working with Global Secondary Indexes: .NET

You can use the AWS SDK for .NET low-level API to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes. These operations map to the corresponding DynamoDB operations. For more information, see the [Amazon DynamoDB API Reference](#).

The following are the common steps for table operations using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

For example, create a `CreateTableRequest` object to create a table and `QueryRequest` object to query a table or an index.

3. Run the appropriate method provided by the client that you created in the preceding step.

Topics

- [Create a table with a Global Secondary Index](#)
- [Describe a table with a Global Secondary Index](#)
- [Query a Global Secondary Index](#)
- [Example: Global Secondary Indexes using the AWS SDK for .NET low-level API](#)

Create a table with a Global Secondary Index

You can create global secondary indexes at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more global secondary indexes. The following C# code example creates a table to hold information about weather data. The partition key is `Location` and the sort key is `Date`. A global secondary index named `PrecipIndex` allows fast access to precipitation data for various locations.

The following are the steps to create a table with a global secondary index, using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values.

For the global secondary index, you must provide the index name, its provisioned throughput settings, the attribute definitions for the index sort key, the key schema for the index, and the attribute projection.

3. Run the `CreateTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps. The code creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index partition key is `Date` and its sort key is `Precipitation`. All of the table attributes are projected into the index. Users

can query this index to obtain weather data for a particular date, optionally sorting the data by precipitation amount.

Because Precipitation is not a key attribute for the table, it is not required. However, WeatherData items without Precipitation do not appear in PrecipIndex.

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

// Attribute definitions
var attributeDefinitions = new List<AttributeDefinition>()
{
    {new AttributeDefinition{
        AttributeName = "Location",
        AttributeType = "S"}},
    {new AttributeDefinition{
        AttributeName = "Date",
        AttributeType = "S"}},
    {new AttributeDefinition(){
        AttributeName = "Precipitation",
        AttributeType = "N"}
    }
};

// Table key schema
var tableKeySchema = new List<KeySchemaElement>()
{
    {new KeySchemaElement {
        AttributeName = "Location",
        KeyType = "HASH"}}, //Partition key
    {new KeySchemaElement {
        AttributeName = "Date",
        KeyType = "RANGE"} //Sort key
};
};

// PrecipIndex
var precipIndex = new GlobalSecondaryIndex
{
    IndexName = "PrecipIndex",
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)10,
```

```
        WriteCapacityUnits = (long)1
    },
    Projection = new Projection { ProjectionType = "ALL" }
};

var indexKeySchema = new List<KeySchemaElement> {
    {new KeySchemaElement { AttributeName = "Date", KeyType = "HASH"}}, //Partition key
    {new KeySchemaElement{AttributeName = "Precipitation",KeyType = "RANGE"}} //Sort key
};

precipIndex.KeySchema = indexKeySchema;

CreateTableRequest createTableRequest = new CreateTableRequest
{
    TableName = tableName,
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)5,
        WriteCapacityUnits = (long)1
    },
    AttributeDefinitions = attributeDefinitions,
    KeySchema = tableKeySchema,
    GlobalSecondaryIndexes = { precipIndex }
};

CreateTableResponse response = client.CreateTable(createTableRequest);
Console.WriteLine(response.CreateTableResult.TableDescription.TableName);
Console.WriteLine(response.CreateTableResult.TableDescription.TableStatus);
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a table with a Global Secondary Index

To get information about global secondary indexes on a table, use `DescribeTable`. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access global secondary index information for a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Run the `describeTable` method by providing the request object as a parameter.

Create an instance of the `DescribeTableRequest` class to provide the request information. You must provide the table name.

3.

The following C# code example demonstrates the preceding steps.

Example

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest
{ TableName = tableName });

List<GlobalSecondaryIndexDescription> globalSecondaryIndexes =
response.DescribeTableResult.Table.GlobalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

foreach (GlobalSecondaryIndexDescription gsiDescription in globalSecondaryIndexes) {
    Console.WriteLine("Info for index " + gsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in gsiDescription.KeySchema) {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
    }

    Projection projection = gsiDescription.Projection;
    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

    if (projection.ProjectionType.ToString().Equals("INCLUDE")) {
        Console.WriteLine("\t\tThe non-key projected attributes are: "
+ projection.NonKeyAttributes);
    }
}
```

Query a Global Secondary Index

You can use `Query` on a global secondary index, in much the same way you `Query` a table. You need to specify the index name, the query criteria for the index partition key and sort key (if

present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a partition key of `Date` and a sort key of `Precipitation`. The index query returns all of the weather data for a particular date, where the precipitation is greater than zero.

The following are the steps to query a global secondary index using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class to provide the request information.
3. Run the `query` method by providing the request object as a parameter.

The attribute name `Date` is a DynamoDB reserved word. Therefore, you must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following C# code example demonstrates the preceding steps.

Example

```
client = new AmazonDynamoDBClient();

QueryRequest queryRequest = new QueryRequest
{
    TableName = "WeatherData",
    IndexName = "PrecipIndex",
    KeyConditionExpression = "#dt = :v_date and Precipitation > :v_precip",
    ExpressionAttributeNames = new Dictionary<String, String> {
        {"#dt", "Date"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {":v_date", new AttributeValue { S = "2013-08-01" }},
        {":v_precip", new AttributeValue { N = "0" }}
    },
    ScanIndexForward = true
};

var result = client.Query(queryRequest);

var items = result.Items;
foreach (var currentItem in items)
{
    foreach (string attr in currentItem.Keys)
    {
        Console.Write(attr + "---> ");
    }
}
```

```
        if (attr == "Precipitation")
    {
        Console.WriteLine(currentItem[attr].N);
    }
    else
    {
        Console.WriteLine(currentItem[attr].S);
    }

}
Console.WriteLine();
}
```

Example: Global Secondary Indexes using the AWS SDK for .NET low-level API

The following C# code example shows how to work with global secondary indexes. The example creates a table named `Issues`, which might be used in a simple bug tracking system for software development. The partition key is `IssueId` and the sort key is `Title`. There are three global secondary indexes on this table:

- `CreateDateIndex` — The partition key is `CreateDate` and the sort key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- `TitleIndex` — The partition key is `Title` and the sort key is `IssueId`. No attributes other than the table keys are projected into the index.
- `DueDateIndex` — The partition key is `DueDate`, and there is no sort key. All of the table attributes are projected into the index.

After the `Issues` table is created, the program loads the table with data representing software bug reports. It then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions for testing the following sample, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
```

```
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelGlobalSecondaryIndexExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        public static String tableName = "Issues";

        public static void Main(string[] args)
        {
            CreateTable();
            LoadData();

            QueryIndex("CreateDateIndex");
            QueryIndex("TitleIndex");
            QueryIndex("DueDateIndex");

            DeleteTable(tableName);

            Console.WriteLine("To continue, press enter");
            Console.Read();
        }

        private static void CreateTable()
        {
            // Attribute definitions
            var attributeDefinitions = new List<AttributeDefinition>()
            {
                {new AttributeDefinition {
                    AttributeName = "IssueId", AttributeType = "S"
                }},
                {new AttributeDefinition {
                    AttributeName = "Title", AttributeType = "S"
                }},
                {new AttributeDefinition {
                    AttributeName = "CreateDate", AttributeType = "S"
                }},
                {new AttributeDefinition {
                    AttributeName = "DueDate", AttributeType = "S"
                }}
            };
        }
    }
}
```

```
};

// Key schema for table
var tableKeySchema = new List<KeySchemaElement>() {
{
    new KeySchemaElement {
        AttributeName= "IssueId",
        KeyType = "HASH" //Partition key
    }
},
{
    new KeySchemaElement {
        AttributeName = "Title",
        KeyType = "RANGE" //Sort key
    }
}
};

// Initial provisioned throughput settings for the indexes
var ptIndex = new ProvisionedThroughput
{
    ReadCapacityUnits = 1L,
    WriteCapacityUnits = 1L
};

// CreateDateIndex
var createDateIndex = new GlobalSecondaryIndex()
{
    IndexName = "CreateDateIndex",
    ProvisionedThroughput = ptIndex,
    KeySchema = {
        new KeySchemaElement {
            AttributeName = "CreateDate", KeyType = "HASH" //Partition key
        },
        new KeySchemaElement {
            AttributeName = "IssueId", KeyType = "RANGE" //Sort key
        }
    },
    Projection = new Projection
    {
        ProjectionType = "INCLUDE",
        NonKeyAttributes = {
            "Description", "Status"
        }
    }
};
```

```
        }

    };

    // TitleIndex
    var titleIndex = new GlobalSecondaryIndex()
    {
        IndexName = "TitleIndex",
        ProvisionedThroughput = ptIndex,
        KeySchema = {
            new KeySchemaElement {
                AttributeName = "Title", KeyType = "HASH" //Partition key
            },
            new KeySchemaElement {
                AttributeName = "IssueId", KeyType = "RANGE" //Sort key
            }
        },
        Projection = new Projection
        {
            ProjectionType = "KEYS_ONLY"
        }
    };

    // DueDateIndex
    var dueDateIndex = new GlobalSecondaryIndex()
    {
        IndexName = "DueDateIndex",
        ProvisionedThroughput = ptIndex,
        KeySchema = {
            new KeySchemaElement {
                AttributeName = "DueDate",
                KeyType = "HASH" //Partition key
            }
        },
        Projection = new Projection
        {
            ProjectionType = "ALL"
        }
    };

    var createTableRequest = new CreateTableRequest
    {
        TableName = tableName,
```

```
ProvisionedThroughput = new ProvisionedThroughput
{
    ReadCapacityUnits = (long)1,
    WriteCapacityUnits = (long)1
},
AttributeDefinitions = attributeDefinitions,
KeySchema = tableKeySchema,
GlobalSecondaryIndexes = {
    createDateIndex, titleIndex, dueDateIndex
}
};

Console.WriteLine("Creating table " + tableName + "...");
client.CreateTable(createTableRequest);

WaitUntilTableReady(tableName);
}

private static void LoadData()
{
    Console.WriteLine("Loading data into table " + tableName + "...");

    // IssueId, Title,
    // Description,
    // CreateDate, LastUpdateDate, DueDate,
    // Priority, Status

    putItem("A-101", "Compilation error",
        "Can't compile Project X - bad version number. What does this mean?",
        "2013-11-01", "2013-11-02", "2013-11-10",
        1, "Assigned");

    putItem("A-102", "Can't read data file",
        "The main data file is missing, or the permissions are incorrect",
        "2013-11-01", "2013-11-04", "2013-11-30",
        2, "In progress");

    putItem("A-103", "Test failure",
        "Functional test of Project X produces errors",
        "2013-11-01", "2013-11-02", "2013-11-10",
        1, "In progress");

    putItem("A-104", "Compilation error",
        "Variable 'messageCount' was not initialized.");
}
```

```
        "2013-11-15", "2013-11-16", "2013-11-30",
        3, "Assigned");

    putItem("A-105", "Network issue",
            "Can't ping IP address 127.0.0.1. Please fix this.",
            "2013-11-15", "2013-11-16", "2013-11-19",
            5, "Assigned");
}

private static void putItem(
    String issueId, String title,
    String description,
    String createDate, String lastUpdateDate, String dueDate,
    Int32 priority, String status)
{
    Dictionary<String, AttributeValue> item = new Dictionary<string,
AttributeValue>();

    item.Add("IssueId", new AttributeValue
    {
        S = issueId
    });
    item.Add("Title", new AttributeValue
    {
        S = title
    });
    item.Add("Description", new AttributeValue
    {
        S = description
    });
    item.Add("CreateDate", new AttributeValue
    {
        S = createDate
    });
    item.Add("LastUpdateDate", new AttributeValue
    {
        S = lastUpdateDate
    });
    item.Add("DueDate", new AttributeValue
    {
        S = dueDate
    });
    item.Add("Priority", new AttributeValue
    {
```

```
N = priority.ToString()
});
item.Add("Status", new AttributeValue
{
    S = status
});

try
{
    client.PutItem(new PutItemRequest
    {
        TableName = tableName,
        Item = item
    });
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

private static void QueryIndex(string indexName)
{
    Console.WriteLine
        ("*****\n");
    Console.WriteLine("Querying index " + indexName + "...");

    QueryRequest queryRequest = new QueryRequest
    {
        TableName = tableName,
        IndexName = indexName,
        ScanIndexForward = true
    };

    String keyConditionExpression;
    Dictionary<string, AttributeValue> expressionAttributeValues = new
Dictionary<string, AttributeValue>();

    if (indexName == "CreateDateIndex")
    {
        Console.WriteLine("Issues filed on 2013-11-01\n");
    }
}
```

```
keyConditionExpression = "CreateDate = :v_date and
begins_with(IssueId, :v_issue)";
    expressionAttributeValues.Add(":v_date", new AttributeValue
{
    S = "2013-11-01"
});
    expressionAttributeValues.Add(":v_issue", new AttributeValue
{
    S = "A-"
});
}
else if (indexName == "TitleIndex")
{
    Console.WriteLine("Compilation errors\n");

    keyConditionExpression = "Title = :v_title and
begins_with(IssueId, :v_issue)";
        expressionAttributeValues.Add(":v_title", new AttributeValue
{
            S = "Compilation error"
});
        expressionAttributeValues.Add(":v_issue", new AttributeValue
{
            S = "A-"
});

// Select
queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
}
else if (indexName == "DueDateIndex")
{
    Console.WriteLine("Items that are due on 2013-11-30\n");

    keyConditionExpression = "DueDate = :v_date";
    expressionAttributeValues.Add(":v_date", new AttributeValue
{
    S = "2013-11-30"
});

// Select
queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
}
else
{
```

```
        Console.WriteLine("\nNo valid index name provided");
        return;
    }

    queryRequest.KeyConditionExpression = keyConditionExpression;
    queryRequest.ExpressionAttributeValues = expressionAttributeValues;

    var result = client.Query(queryRequest);
    var items = result.Items;
    foreach (var currentItem in items)
    {
        foreach (string attr in currentItem.Keys)
        {
            if (attr == "Priority")
            {
                Console.WriteLine(attr + "---> " + currentItem[attr].N);
            }
            else
            {
                Console.WriteLine(attr + "---> " + currentItem[attr].S);
            }
        }
        Console.WriteLine();
    }
}

private static void DeleteTable(string tableName)
{
    Console.WriteLine("Deleting table " + tableName + "...");
    client.DeleteTable(new DeleteTableRequest
    {
        TableName = tableName
    });
    WaitForTableToDelete(tableName);
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try

```

```
{  
    var res = client.DescribeTable(new DescribeTableRequest  
    {  
        TableName = tableName  
    });  
  
    Console.WriteLine("Table name: {0}, status: {1}",  
        res.Table.TableName,  
        res.Table.TableStatus);  
    status = res.Table.TableStatus;  
}  
catch (ResourceNotFoundException)  
{  
    // DescribeTable is eventually consistent. So you might  
    // get resource not found. So we handle the potential exception.  
}  
} while (status != "ACTIVE");  
}  
  
private static void WaitForTableToDelete(string tableName)  
{  
    bool tablePresent = true;  
  
    while (tablePresent)  
    {  
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.  
        try  
        {  
            var res = client.DescribeTable(new DescribeTableRequest  
            {  
                TableName = tableName  
            });  
  
            Console.WriteLine("Table name: {0}, status: {1}",  
                res.Table.TableName,  
                res.Table.TableStatus);  
        }  
        catch (ResourceNotFoundException)  
        {  
            tablePresent = false;  
        }  
    }  
}
```

```
}
```

Working with Global Secondary Indexes: AWS CLI

You can use the AWS CLI to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes.

Topics

- [Create a table with a Global Secondary Index](#)
- [Add a Global Secondary Index to an existing table](#)
- [Describe a table with a Global Secondary Index](#)
- [Query a Global Secondary Index](#)

Create a table with a Global Secondary Index

Global secondary indexes may be created at the same time you create a table. To do this, use the `create-table` parameter and provide your specifications for one or more global secondary indexes. The following example creates a table named `GameScores` with a global secondary index called `GameTitleIndex`. The base table has a partition key of `UserId` and a sort key of `GameTitle`, allowing you to find an individual user's best score for a specific game efficiently, whereas the GSI has a partition key of `GameTitle` and a sort key of `TopScore`, allowing you to quickly find the overall highest score for a particular game.

```
aws dynamodb create-table \
  --table-name GameScores \
  --attribute-definitions AttributeName=UserId,AttributeType=S \
    AttributeName=GameTitle,AttributeType=S \
    AttributeName=TopScore,AttributeType=N \
  --key-schema AttributeName=UserId,KeyType=HASH \
    AttributeName=GameTitle,KeyType=RANGE \
  --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --global-secondary-indexes \
  "[
    {
      \"IndexName\": \"GameTitleIndex\",
      \"KeySchema\": [{\"AttributeName\": \"GameTitle\", \"KeyType\": \"HASH\"}, \
        {\"AttributeName\": \"TopScore\", \"KeyType\": \"RANGE\"}]
    }]
  ]"
```

```
  \"Projection\":{  
    \"ProjectionType\":\"INCLUDE\",  
    \"NonKeyAttributes\":[\"UserId\"]  
  },  
  \"ProvisionedThroughput\": {  
    \"ReadCapacityUnits\": 10,  
    \"WriteCapacityUnits\": 5  
  }  
}  
]  
]"
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table. You can use [describe-table](#) to determine the status of the table creation.

Add a Global Secondary Index to an existing table

Global secondary indexes may also be added or modified after table creation. To do this, use the update-table parameter and provide your specifications for one or more global secondary indexes. The following example uses the same schema as the previous example, but assumes that the table has already been created and we're adding the GSI later.

```
aws dynamodb update-table \  
  --table-name GameScores \  
  --attribute-definitions AttributeName=TopScore,AttributeType=N \  
  --global-secondary-index-updates \  
  "[  
    {  
      \"Create\": {  
        \"IndexName\": \"GameTitleIndex\",  
        \"KeySchema\": [{\"AttributeName\": \"GameTitle\", \"KeyType\": \"HASH\"},  
                      {\"AttributeName\": \"TopScore\", \"KeyType\": \"RANGE\"}],  
        \"Projection\":{  
          \"ProjectionType\":\"INCLUDE\",  
          \"NonKeyAttributes\":[\"UserId\"]  
        }  
      }  
    }  
  ]"
```

Describe a table with a Global Secondary Index

To get information about Global Secondary Indexes on a table, use the `describe-table` parameter. For each index, you can access its name, key schema, and projected attributes.

```
aws dynamodb describe-table --table-name GameScores
```

Query a Global Secondary Index

You can use the query operation on a global secondary index in much the same way that you query a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `GameTitleIndex` and the index sort key is `GameTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute projections](#).

```
aws dynamodb query --table-name GameScores\  
  --index-name GameTitleIndex \  
  --key-condition-expression "GameTitle = :v_game" \  
  --expression-attribute-values '{":v_game":{"S":"Alien Adventure"} }'
```

Local Secondary Indexes

Some applications only need to query data using the base table's primary key. However, there might be situations where an alternative sort key would be helpful. To give your application a choice of sort keys, you can create one or more local secondary indexes on an Amazon DynamoDB table and issue Query or Scan requests against these indexes.

Topics

- [Scenario: Using a Local Secondary Index](#)
- [Attribute projections](#)
- [Creating a Local Secondary Index](#)
- [Reading data from a Local Secondary Index](#)
- [Item writes and Local Secondary Indexes](#)
- [Provisioned throughput considerations for Local Secondary Indexes](#)

- [Storage considerations for Local Secondary Indexes](#)
- [Item collections in Local Secondary Indexes](#)
- [Working with Local Secondary Indexes: Java](#)
- [Working with Local Secondary Indexes: .NET](#)
- [Working with Local Secondary Indexes: AWS CLI](#)

Scenario: Using a Local Secondary Index

As an example, consider the `Thread` table that is defined in [Creating tables and loading data for code examples in DynamoDB](#). This table is useful for an application such as the [AWS discussion forums](#). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)

Thread

ForumName	Subject	LastPostDateTime	Replies	
"S3"	"aaa"	"2015-03-15:17:24:31"	12	...
"S3"	"bbb"	"2015-01-22:23:18:01"	3	...
"S3"	"ccc"	"2015-02-31:13:14:21"	4	...
"S3"	"ddd"	"2015-01-03:09:21:11"	9	...
"EC2"	"yyy"	"2015-02-12:11:07:56"	18	...
"EC2"	"zzz"	"2015-01-18:07:33:42"	0	...
"RDS"	"rrr"	"2015-01-19:01:13:24"	3	...
"RDS"	"sss"	"2015-03-11:06:53:00"	11	...
"RDS"	"ttt"	"2015-10-22:12:19:44"	5	...
...

DynamoDB stores all of the items with the same partition key value continuously. In this example, given a particular `ForumName`, a `Query` operation could immediately locate all of the threads for that forum. Within a group of items with the same partition key value, the items are sorted by sort key value. If the sort key (`Subject`) is also provided in the query, DynamoDB can narrow down the results that are returned—for example, returning all of the threads in the "S3" forum that have a `Subject` beginning with the letter "a".

Some requests might require more complex data access patterns. For example:

- Which forum threads get the most views and replies?
- Which thread in a particular forum has the largest number of messages?

- How many threads were posted in a particular forum within a particular time period?

To answer these questions, the `Query` action would not be sufficient. Instead, you would have to Scan the entire table. For a table with millions of items, this would consume a large amount of provisioned read throughput and take a long time to complete.

However, you can specify one or more local secondary indexes on non-key attributes, such as `Replies` or `LastPostDateTime`.

A *local secondary index* maintains an alternate sort key for a given partition key value. A local secondary index also contains a copy of some or all of the attributes from its base table. You specify which attributes are projected into the local secondary index when you create the table. The data in a local secondary index is organized by the same partition key as the base table, but with a different sort key. This lets you access data items efficiently across this different dimension. For greater query or scan flexibility, you can create up to five local secondary indexes per table.

Suppose that an application needs to find all of the threads that have been posted within the last three months in a particular forum. Without a local secondary index, the application would have to Scan the entire `Thread` table and discard any posts that were not within the specified time frame. With a local secondary index, a `Query` operation could use `LastPostDateTime` as a sort key and find the data quickly.

The following diagram shows a local secondary index named `LastPostIndex`. Note that the partition key is the same as that of the `Thread` table, but the sort key is `LastPostDateTime`.

LastPostIndex

ForumName	LastPostDateTime	Subject
"S3"	"2015-01-03:09:21:11"	"ddd"
"S3"	"2015-01-22:23:18:01"	"bbb"
"S3"	"2015-02-31:13:14:21"	"ccc"
"S3"	"2015-03-15:17:24:31"	"aaa"
"EC2"	"2015-01-18:07:33:42"	"zzz"
"EC2"	"2015-02-12:11:07:56"	"yyy"
"RDS"	"2015-01-19:01:13:24"	"rrr"
"RDS"	"2015-02-22:12:19:44"	"ttt"
"RDS"	"2015-03-11:06:53:00"	"sss"

*** *** ***

Every local secondary index must meet the following conditions:

- The partition key is the same as that of its base table.
- The sort key consists of exactly one scalar attribute.
- The sort key of the base table is projected into the index, where it acts as a non-key attribute.

In this example, the partition key is ForumName and the sort key of the local secondary index is LastPostDateTime. In addition, the sort key value from the base table (in this example, Subject) is projected into the index, but it is not a part of the index key. If an application needs a list that is based on ForumName and LastPostDateTime, it can issue a Query request against LastPostIndex. The query results are sorted by LastPostDateTime, and can be returned in ascending or descending order. The query can also apply key conditions, such as returning only items that have a LastPostDateTime within a particular time span.

Every local secondary index automatically contains the partition and sort keys from its base table; you can optionally project non-key attributes into the index. When you query the index, DynamoDB can retrieve these projected attributes efficiently. When you query a local secondary index, the

query can also retrieve attributes that are *not* projected into the index. DynamoDB automatically fetches these attributes from the base table, but at a greater latency and with higher provisioned throughput costs.

For any local secondary index, you can store up to 10 GB of data per distinct partition key value. This figure includes all of the items in the base table, plus all of the items in the indexes, that have the same partition key value. For more information, see [Item collections in Local Secondary Indexes](#).

Attribute projections

With LastPostIndex, an application could use ForumName and LastPostDateTime as query criteria. However, to retrieve any additional attributes, DynamoDB must perform additional read operations against the Thread table. These extra reads are known as *fetches*, and they can increase the total amount of provisioned throughput required for a query.

Suppose that you wanted to populate a webpage with a list of all the threads in "S3" and the number of replies for each thread, sorted by the last reply date/time beginning with the most recent reply. To populate this list, you would need the following attributes:

- Subject
- Replies
- LastPostDateTime

The most efficient way to query this data and to avoid fetch operations would be to project the Replies attribute from the table into the local secondary index, as shown in this diagram.

LastPostIndex

ForumName	LastPostDateTime	Subject	Replies
"S3"	"2015-01-03:09:21:11"	"ddd"	9
"S3"	"2015-01-22:23:18:01"	"bbb"	3
"S3"	"2015-02-31:13:14:21"	"ccc"	4
"S3"	"2015-03-15:17:24:31"	"aaa"	12
"EC2"	"2015-01-18:07:33:42"	"zzz"	0
"EC2"	"2015-02-12:11:07:56"	"yyy"	18
"RDS"	"2015-01-19:01:13:24"	"rrr"	3
"RDS"	"2015-02-22:12:19:44"	"ttt"	5
"RDS"	"2015-03-11:06:53:00"	"sss"	11
...

A *projection* is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, you need to specify the attributes that will be projected into the index. DynamoDB provides three different options for this:

- *KEYS_ONLY* – Each item in the index consists only of the table partition key and sort key values, plus the index key values. The *KEYS_ONLY* option results in the smallest possible secondary index.
- *INCLUDE* – In addition to the attributes described in *KEYS_ONLY*, the secondary index will include other non-key attributes that you specify.
- *ALL* – The secondary index includes all of the attributes from the source table. Because all of the table data is duplicated in the index, an *ALL* projection results in the largest possible secondary index.

In the previous diagram, the non-key attribute `Replies` is projected into `LastPostIndex`. An application can query `LastPostIndex` instead of the full `Thread` table to populate a webpage with `Subject`, `Replies`, and `LastPostDateTime`. If any other non-key attributes are requested, DynamoDB would need to fetch those attributes from the `Thread` table.

From an application's point of view, fetching additional attributes from the base table is automatic and transparent, so there is no need to rewrite any application logic. However, such fetching can greatly reduce the performance advantage of using a local secondary index.

When you choose the attributes to project into a local secondary index, you must consider the tradeoff between provisioned throughput costs and storage costs:

- If you need to access just a few attributes with the lowest possible latency, consider projecting only those attributes into a local secondary index. The smaller the index, the less that it costs to store it, and the less your write costs are. If there are attributes that you occasionally need to fetch, the cost for provisioned throughput may well outweigh the longer-term cost of storing those attributes.
- If your application frequently accesses some non-key attributes, you should consider projecting those attributes into a local secondary index. The additional storage costs for the local secondary index offset the cost of performing frequent table scans.
- If you need to access most of the non-key attributes on a frequent basis, you can project these attributes—or even the entire base table—into a local secondary index. This gives you maximum flexibility and lowest provisioned throughput consumption, because no fetching would be required. However, your storage cost would increase, or even double if you are projecting all attributes.
- If your application needs to query a table infrequently, but must perform many writes or updates against the data in the table, consider projecting `KEYS_ONLY`. The local secondary index would be of minimal size, but would still be available when needed for query activity.

Creating a Local Secondary Index

To create one or more local secondary indexes on a table, use the `LocalSecondaryIndexes` parameter of the `CreateTable` operation. Local secondary indexes on a table are created when the table is created. When you delete a table, any local secondary indexes on that table are also deleted.

You must specify one non-key attribute to act as the sort key of the local secondary index. The attribute that you choose must be a scalar String, Number, or Binary. Other scalar types, document types, and set types are not allowed. For a complete list of data types, see [Data types](#).

Important

For tables with local secondary indexes, there is a 10 GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB. For more information, see [Item collection size limit](#).

You can project attributes of any data type into a local secondary index. This includes scalars, documents, and sets. For a complete list of data types, see [Data types](#).

Reading data from a Local Secondary Index

You can retrieve items from a local secondary index using the Query and Scan operations. The GetItem and BatchGetItem operations can't be used on a local secondary index.

Querying a Local Secondary Index

In a DynamoDB table, the combined partition key value and sort key value for each item must be unique. However, in a local secondary index, the sort key value does not need to be unique for a given partition key value. If there are multiple items in the local secondary index that have the same sort key value, a Query operation returns all of the items that have the same partition key value. In the response, the matching items are not returned in any particular order.

You can query a local secondary index using either eventually consistent or strongly consistent reads. To specify which type of consistency you want, use the ConsistentRead parameter of the Query operation. A strongly consistent read from a local secondary index always returns the latest updated values. If the query needs to fetch additional attributes from the base table, those attributes will be consistent with respect to the index.

Example

Consider the following data returned from a Query that requests data from the discussion threads in a particular forum.

```
{  
  "TableName": "Thread",
```

```
"IndexName": "LastPostIndex",
"ConsistentRead": false,
"ProjectionExpression": "Subject, LastPostDateTime, Replies, Tags",
"KeyConditionExpression":
    "ForumName = :v_forum and LastPostDateTime between :v_start and :v_end",
"ExpressionAttributeValues": {
    ":v_start": {"S": "2015-08-31T00:00:00.000Z"},
    ":v_end": {"S": "2015-11-31T00:00:00.000Z"},
    ":v_forum": {"S": "EC2"}
}
}
```

In this query:

- DynamoDB accesses `LastPostIndex`, using the `ForumName` partition key to locate the index items for "EC2". All of the index items with this key are stored adjacent to each other for rapid retrieval.
- Within this forum, DynamoDB uses the index to look up the keys that match the specified `LastPostDateTime` condition.
- Because the `Replies` attribute is projected into the index, DynamoDB can retrieve this attribute without consuming any additional provisioned throughput.
- The `Tags` attribute is not projected into the index, so DynamoDB must access the `Thread` table and fetch this attribute.
- The results are returned, sorted by `LastPostDateTime`. The index entries are sorted by partition key value and then by sort key value, and `Query` returns them in the order they are stored. (You can use the `ScanIndexForward` parameter to return the results in descending order.)

Because the `Tags` attribute is not projected into the local secondary index, DynamoDB must consume additional read capacity units to fetch this attribute from the base table. If you need to run this query often, you should project `Tags` into `LastPostIndex` to avoid fetching from the base table. However, if you needed to access `Tags` only occasionally, the additional storage cost for projecting `Tags` into the index might not be worthwhile.

Scanning a Local Secondary Index

You can use `Scan` to retrieve all of the data from a local secondary index. You must provide the base table name and the index name in the request. With a `Scan`, DynamoDB reads all of

the data in the index and returns it to the application. You can also request that only some of the data be returned, and that the remaining data should be discarded. To do this, use the `FilterExpression` parameter of the Scan API. For more information, see [Filter expressions for scan](#).

Item writes and Local Secondary Indexes

DynamoDB automatically keeps all local secondary indexes synchronized with their respective base tables. Applications never write directly to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

When you create a local secondary index, you specify an attribute to serve as the sort key for the index. You also specify a data type for that attribute. This means that whenever you write an item to the base table, if the item defines an index key attribute, its type must match the index key schema's data type. In the case of `LastPostIndex`, the `LastPostDateTime` sort key in the index is defined as a `String` data type. If you try to add an item to the `Thread` table and specify a different data type for `LastPostDateTime` (such as `Number`), DynamoDB returns a `ValidationException` because of the data type mismatch.

There is no requirement for a one-to-one relationship between the items in a base table and the items in a local secondary index. In fact, this behavior can be advantageous for many applications.

A table with many local secondary indexes incurs higher costs for write activity than tables with fewer indexes. For more information, see [Provisioned throughput considerations for Local Secondary Indexes](#).

Important

For tables with local secondary indexes, there is a 10 GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB. For more information, see [Item collection size limit](#).

Provisioned throughput considerations for Local Secondary Indexes

When you create a table in DynamoDB, you provision read and write capacity units for the table's expected workload. That workload includes read and write activity on the table's local secondary indexes.

To view the current rates for provisioned throughput capacity, see [Amazon DynamoDB pricing](#).

Read capacity units

When you query a local secondary index, the number of read capacity units consumed depends on how the data is accessed.

As with table queries, an index query can use either eventually consistent or strongly consistent reads depending on the value of `ConsistentRead`. One strongly consistent read consumes one read capacity unit; an eventually consistent read consumes only half of that. Thus, by choosing eventually consistent reads, you can reduce your read capacity unit charges.

For index queries that request only index keys and projected attributes, DynamoDB calculates the provisioned read activity in the same way as it does for queries against tables. The only difference is that the calculation is based on the sizes of the index entries, rather than the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned; the result is then rounded up to the next 4 KB boundary. For more information about how DynamoDB calculates provisioned throughput usage, see [Managing settings on DynamoDB provisioned capacity tables](#).

For index queries that read attributes that are not projected into the local secondary index, DynamoDB needs to fetch those attributes from the base table, in addition to reading the projected attributes from the index. These fetches occur when you include any non-projected attributes in the `Select` or `ProjectionExpression` parameters of the `Query` operation. Fetching causes additional latency in query responses, and it also incurs a higher provisioned throughput cost: In addition to the reads from the local secondary index described previously, you are charged for read capacity units for every base table item fetched. This charge is for reading each entire item from the table, not just the requested attributes.

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all the attribute names and values across all of the items returned. However, if a `Query` against a local secondary index causes DynamoDB to fetch item attributes from the base table, the maximum size of the data in the results might be lower. In this case, the result size is the sum of:

- The size of the matching items in the index, rounded up to the next 4 KB.
- The size of each matching item in the base table, with each item individually rounded up to the next 4 KB.

Using this formula, the maximum size of the results returned by a `Query` operation is still 1 MB.

For example, consider a table where the size of each item is 300 bytes. There is a local secondary index on that table, but only 200 bytes of each item is projected into the index. Now suppose that you Query this index, that the query requires table fetches for each item, and that the query returns 4 items. DynamoDB sums up the following:

- The size of the matching items in the index: $200 \text{ bytes} \times 4 \text{ items} = 800 \text{ bytes}$; this is then rounded up to 4 KB.
- The size of each matching item in the base table: $(300 \text{ bytes, rounded up to 4 KB}) \times 4 \text{ items} = 16 \text{ KB}$.

The total size of the data in the result is therefore 20 KB.

Write capacity units

When an item in a table is added, updated, or deleted, updating the local secondary indexes consumes provisioned write capacity units for the table. The total provisioned throughput cost for a write is the sum of write capacity units consumed by writing to the table and those consumed by updating the local secondary indexes.

The cost of writing an item to a local secondary index depends on several factors:

- If you write a new item to the table that defines an indexed attribute, or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required: one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1 KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries need to return and projecting only those attributes into the index.

Storage considerations for Local Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any local secondary indexes in which those attributes should appear. Your AWS account is charged for storage of the item in the base table and also for storage of attributes in any local secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- The size in bytes of the base table primary key (partition key and sort key)
- The size in bytes of the index key attribute
- The size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a local secondary index, you can estimate the average size of an item in the index and then multiply by the number of items in the index.

If a table contains an item where a particular attribute is not defined, but that attribute is defined as an index sort key, then DynamoDB does not write any data for that item to the index.

Item collections in Local Secondary Indexes

Note

This section pertains only to tables that have local secondary indexes.

In DynamoDB, an *item collection* is any group of items that have the same partition key value in a table and all of its local secondary indexes. In the examples used throughout this section, the partition key for the Thread table is ForumName, and the partition key for LastPostIndex is also ForumName. All the table and index items with the same ForumName are part of the same item collection. For example, in the Thread table and the LastPostIndex local secondary index, there is an item collection for forum EC2 and a different item collection for forum RDS.

The following diagram shows the item collection for forum S3.

Thread

ForumName	Subject	LastPostDateTime	Thread	
ForumName: "S3"	"S3"	"aaa"	"2015-03-15:17:24:31"	12
	"S3"	"bbb"	"2015-01-22:23:18:01"	3
	"S3"	"ccc"	"2015-02-31:13:14:21"	4
	"S3"	"ddd"	"2015-01-03:09:21:11"	9
	"EC2"	"yyy"	"2015-02-12:11:07:56"	18
	"EC2"	"zzz"	"2015-01-18:07:33:42"	0
	"RDS"	"rrr"	"2015-01-19:01:13:24"	3
	"RDS"	"sss"	"2015-03-11:06:53:00"	11
	"RDS"	"ttt"	"2015-10-22:12:19:44"	5

LastPostIndex

ForumName	LastPostDateTime	Subject	Replies
ForumName: "S3"	"2015-01-03:09:21:11"	"ddd"	9
	"2015-01-22:23:18:01"	"bbb"	3
	"2015-02-31:13:14:21"	"ccc"	4
	"2015-03-15:17:24:31"	"aaa"	12
	"2015-01-18:07:33:42"	"zzz"	0
	"2015-02-12:11:07:56"	"yyy"	18
	"2015-01-19:01:13:24"	"rrr"	3
	"2015-02-22:12:19:44"	"ttt"	5
	"2015-03-11:06:53:00"	"sss"	11

In this diagram, the item collection consists of all the items in Thread and LastPostIndex where the ForumName partition key value is "S3". If there were other local secondary indexes on the table, any items in those indexes with ForumName equal to "S3" would also be part of the item collection.

You can use any of the following operations in DynamoDB to return information about item collections:

- BatchWriteItem
- DeleteItem
- PutItem
- UpdateItem
- TransactWriteItems

Each of these operations supports the `ReturnItemCollectionMetrics` parameter. When you set this parameter to `SIZE`, you can view information about the size of each item collection in the index.

Example

The following is an example from the output of an `UpdateItem` operation on the `Thread` table, with `ReturnItemCollectionMetrics` set to `SIZE`. The item that was updated had a `ForumName` value of "EC2", so the output includes information about that item collection.

```
{  
    ItemCollectionMetrics: {  
        ItemCollectionKey: {  
            ForumName: "EC2"  
        },  
        SizeEstimateRangeGB: [0.0, 1.0]  
    }  
}
```

The `SizeEstimateRangeGB` object shows that the size of this item collection is between 0 and 1 GB. DynamoDB periodically updates this size estimate, so the numbers might be different next time the item is modified.

Item collection size limit

The maximum size of any item collection for a table which has one or more local secondary indexes is 10 GB. This does not apply to item collections in tables without local secondary indexes, and also does not apply to item collections in global secondary indexes. Only tables that have one or more local secondary indexes are affected.

If an item collection exceeds the 10 GB limit, DynamoDB returns an `ItemCollectionSizeLimitExceededException`, and you won't be able to add more items to the item collection or increase the sizes of items that are in the item collection. (Read and write operations that shrink the size of the item collection are still allowed.) You can still add items to other item collections.

To reduce the size of an item collection, you can do one of the following:

- Delete any unnecessary items with the partition key value in question. When you delete these items from the base table, DynamoDB also removes any index entries that have the same partition key value.
- Update the items by removing attributes or by reducing the size of the attributes. If these attributes are projected into any local secondary indexes, DynamoDB also reduces the size of the corresponding index entries.
- Create a new table with the same partition key and sort key, and then move items from the old table to the new table. This might be a good approach if a table has historical data that is infrequently accessed. You might also consider archiving this historical data to Amazon Simple Storage Service (Amazon S3).

When the total size of the item collection drops below 10 GB, you can once again add items with the same partition key value.

We recommend as a best practice that you instrument your application to monitor the sizes of your item collections. One way to do so is to set the `ReturnItemCollectionMetrics` parameter to `SIZE` whenever you use `BatchWriteItem`, `DeleteItem`, `PutItem`, or `UpdateItem`. Your application should examine the `ReturnItemCollectionMetrics` object in the output and log an error message whenever an item collection exceeds a user-defined limit (8 GB, for example). Setting a limit that is less than 10 GB would provide an early warning system so you know that an item collection is approaching the limit in time to do something about it.

Item collections and partitions

In a table with one or more local secondary indexes, each item collection is stored in one partition. The total size of such an item collection is limited to the capability of that partition: 10 GB. For an application where the data model includes item collections which are unbounded in size, or where you might reasonably expect some item collections to grow beyond 10 GB in the future, you should consider using a global secondary index instead.

You should design your applications so that table data is evenly distributed across distinct partition key values. For tables with local secondary indexes, your applications should not create "hot spots" of read and write activity within a single item collection on a single partition.

Working with Local Secondary Indexes: Java

You can use the AWS SDK for Java Document API to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the indexes.

The following are the common steps for table operations using the AWS SDK for Java Document API.

1. Create an instance of the DynamoDB class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.
3. Call the appropriate method provided by the client that you created in the preceding step.

Topics

- [Create a table with a Local Secondary Index](#)
- [Describe a table with a Local Secondary Index](#)
- [Query a Local Secondary Index](#)
- [Example: Local Secondary Indexes using the Java document API](#)

Create a table with a Local Secondary Index

Local secondary indexes must be created at the same time you create a table. To do this, use the `createTable` method and provide your specifications for one or more local secondary

indexes. The following Java code example creates a table to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex`, facilitates queries by album title.

The following are the steps to create a table with a local secondary index, using the DynamoDB document API.

1. Create an instance of the DynamoDB class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the local secondary index, you must provide the index name, the name and data type for the index sort key, the key schema for the index, and the attribute projection.

3. Call the `createTable` method by providing the request object as a parameter.

The following Java code example demonstrates the preceding steps. The code creates a table (`Music`) with a secondary index on the `AlbumTitle` attribute. The table partition key and sort key, plus the index sort key, are the only attributes projected into the index.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName);

//ProvisionedThroughput
createTableRequest.setProvisionedThroughput(new
ProvisionedThroughput().withReadCapacityUnits((long)5).withWriteCapacityUnits((long)5));

//AttributeDefinitions
ArrayList<AttributeDefinition> attributeDefinitions= new
ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Artist").withAttributeType("S"));
attributeDefinitions.add(new
AttributeDefinition().withAttributeName("SongTitle").withAttributeType("S"));
attributeDefinitions.add(new
AttributeDefinition().withAttributeName("AlbumTitle").withAttributeType("S"));
```

```
createTableRequest.setAttributeDefinitions(attributeDefinitions);

//KeySchema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new
    KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH)); // Partition key
tableKeySchema.add(new
    KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE)); //Sort key

createTableRequest.setKeySchema(tableKeySchema);

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new
    KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH)); // Partition key
indexKeySchema.add(new
    KeySchemaElement().withAttributeName("AlbumTitle").withKeyType(KeyType.RANGE)); // Sort key

Projection projection = new Projection().withProjectionType(ProjectionType.INCLUDE);
ArrayList<String> nonKeyAttributes = new ArrayList<String>();
nonKeyAttributes.add("Genre");
nonKeyAttributes.add("Year");
projection.setNonKeyAttributes(nonKeyAttributes);

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
    .withIndexName("AlbumTitleIndex").withKeySchema(indexKeySchema).withProjection(projection);

ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
    ArrayList<LocalSecondaryIndex>();
localSecondaryIndexes.add(localSecondaryIndex);
createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `describeTable` method. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access local secondary index information of a table using the AWS SDK for Java Document API.

1. Create an instance of the DynamoDB class.
2. Create an instance of the Table class. You must provide the table name.
3. Call the `describeTable` method on the Table object.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);

TableDescription tableDescription = table.describe();

List<LocalSecondaryIndexDescription> localSecondaryIndexes
    = tableDescription.getLocalSecondaryIndexes();

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

Iterator<LocalSecondaryIndexDescription> lsiIter = localSecondaryIndexes.iterator();
while (lsiIter.hasNext()) {

    LocalSecondaryIndexDescription lsiDescription = lsiIter.next();
    System.out.println("Info for index " + lsiDescription.getIndexName() + ":");
    Iterator<KeySchemaElement> kseIter = lsiDescription.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
    Projection projection = lsiDescription.getProjection();
```

```
System.out.println("\tThe projection type is: " + projection.getProjectionType());
if (projection.getProjectionType().toString().equals("INCLUDE")) {
    System.out.println("\t\tThe non-key projected attributes are: " +
projection.getNonKeyAttributes());
}
}
```

Query a Local Secondary Index

You can use the Query operation on a local secondary index in much the same way that you Query a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute projections](#).

The following are the steps to query a local secondary index using the AWS SDK for Java Document API.

1. Create an instance of the DynamoDB class.
2. Create an instance of the Table class. You must provide the table name.
3. Create an instance of the Index class. You must provide the index name.
4. Call the query method of the Index class.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);
Index index = table.getIndex("AlbumTitleIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Artist = :v_artist and AlbumTitle = :v_title")
    .WithValueMap(new ValueMap()
```

```
.withString(":v_artist", "Acme Band")
.withString(":v_title", "Songs About Life"));

ItemCollection<QueryOutcome> items = index.query(spec);

Iterator<Item> itemsIter = items.iterator();

while (itemsIter.hasNext()) {
    Item item = itemsIter.next();
    System.out.println(item.toJSONPretty());
}
```

Example: Local Secondary Indexes using the Java document API

The following Java code example shows how to work with local secondary indexes in Amazon DynamoDB. The example creates a table named `CustomerOrders` with a partition key of `CustomerId` and a sort key of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex` — The sort key is `OrderCreationDate`, and the following attributes are projected into the index:
 - `ProductCategory`
 - `ProductName`
 - `OrderStatus`
 - `ShipmentTrackingId`
- `IsOpenIndex` — The sort key is `IsOpen`, and all of the table attributes are projected into the index.

After the `CustomerOrders` table is created, the program loads the table with data representing customer orders. It then queries the data using the local secondary indexes. Finally, the program deletes the `CustomerOrders` table.

For step-by-step instructions for testing the following sample, see [Java code examples](#).

Example

```
package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.PutItemOutcome;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.LocalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ReturnConsumedCapacity;
import com.amazonaws.services.dynamodbv2.model.Select;

public class DocumentAPILocalSecondaryIndexExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    public static String tableName = "CustomerOrders";

    public static void main(String[] args) throws Exception {

        createTable();
        loadData();

        query(null);
        query("IsOpenIndex");
        query("OrderCreationDateIndex");

        deleteTable(tableName);

    }

    public static void createTable() {
```

```
        CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
                    .withProvisionedThroughput(
                        new
ProvisionedThroughput().withReadCapacityUnits((long) 1)

.withWriteCapacityUnits((long) 1));

        // Attribute definitions for table partition and sort keys
        ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
        attributeDefinitions
                    .add(new
AttributeDefinition().withAttributeName("CustomerId").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("OrderId").withAttributeType("N"));

        // Attribute definition for index primary key attributes
        attributeDefinitions
                    .add(new
AttributeDefinition().withAttributeName("OrderCreationDate")
                            .withAttributeType("N"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("IsOpen").withAttributeType("N"));

        createTableRequest.setAttributeDefinitions(attributeDefinitions);

        // Key schema for table
        ArrayList<KeySchemaElement> tableKeySchema = new
ArrayList<KeySchemaElement>();
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("CustomerId").withKeyType(KeyType.HASH)); // Partition

                    // key
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("OrderId").withKeyType(KeyType.RANGE)); // Sort

                    // key

        createTableRequest.setKeySchema(tableKeySchema);
```

```
ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
ArrayList<LocalSecondaryIndex>();

// OrderCreationDateIndex
LocalSecondaryIndex orderCreationDateIndex = new LocalSecondaryIndex()
    .withIndexName("OrderCreationDateIndex");

// Key schema for OrderCreationDateIndex
ArrayList<KeySchemaElement> indexKeySchema = new
ArrayList<KeySchemaElement>();
indexKeySchema.add(new
KeySchemaElement().withAttributeName("CustomerId").withKeyType(KeyType.HASH)); // Partition

                                // key
indexKeySchema.add(new
KeySchemaElement().withAttributeName("OrderCreationDate")
    .withKeyType(KeyType.RANGE)); // Sort
                                // key

orderCreationDateIndex.setKeySchema(indexKeySchema);

// Projection (with list of projected attributes) for
// OrderCreationDateIndex
Projection projection = new
Projection().withProjectionType(ProjectionType.INCLUDE);
ArrayList<String> nonKeyAttributes = new ArrayList<String>();
nonKeyAttributes.add("ProductCategory");
nonKeyAttributes.add("ProductName");
projection.setNonKeyAttributes(nonKeyAttributes);

orderCreationDateIndex.setProjection(projection);

localSecondaryIndexes.add(orderCreationDateIndex);

// IsOpenIndex
LocalSecondaryIndex isOpenIndex = new
LocalSecondaryIndex().withIndexName("IsOpenIndex");

// Key schema for IsOpenIndex
indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new
KeySchemaElement().withAttributeName("CustomerId").withKeyType(KeyType.HASH)); // Partition
```

```
// key
indexKeySchema.add(new
KeySchemaElement().withAttributeName("IsOpen").withKeyType(KeyType.RANGE)); // Sort

// key

// Projection (all attributes) for IsOpenIndex
projection = new Projection().withProjectionType(ProjectionType.ALL);

isOpenIndex.setKeySchema(indexKeySchema);
isOpenIndex.setProjection(projection);

localSecondaryIndexes.add(isOpenIndex);

// Add index definitions to CreateTable request
createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

System.out.println("Creating table " + tableName + "...");
System.out.println(dynamoDB.createTable(createTableRequest));

// Wait for table to become active
System.out.println("Waiting for " + tableName + " to become
ACTIVE...");

try {
    Table table = dynamoDB.getTable(tableName);
    table.waitForActive();
} catch (InterruptedException e) {
    e.printStackTrace();
}

}

public static void query(String indexName) {

    Table table = dynamoDB.getTable(tableName);

System.out.println("\n*****\n");
    System.out.println("Querying table " + tableName + "...");

    QuerySpec querySpec = new
QuerySpec().withConsistentRead(true).withScanIndexForward(true)

    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);
```

```
if (indexName == "IsOpenIndex") {

    System.out.println("\nUsing index: '" + indexName + "' Bob's
orders that are open.");
    System.out.println("Only a user-specified list of attributes
are returned\n");
    Index index = table.getIndex(indexName);

    querySpec.withKeyConditionExpression("CustomerId = :v_custid
and IsOpen = :v_isopen")
        .withValueMap(new
ValueMap().withString(":v_custid", "bob@example.com")
        .withNumber(":v_isopen", 1));

    querySpec.withProjectionExpression(
        "OrderCreationDate, ProductCategory,
ProductName, OrderStatus");

    ItemCollection<QueryOutcome> items = index.query(querySpec);
    Iterator<Item> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }

} else if (indexName == "OrderCreationDateIndex") {
    System.out.println("\nUsing index: '" + indexName
        + "' Bob's orders that were placed after
01/31/2015.");
    System.out.println("Only the projected attributes are returned
\n");
    Index index = table.getIndex(indexName);

    querySpec.withKeyConditionExpression(
        "CustomerId = :v_custid and OrderCreationDate
>= :v_orddate")
        .withValueMap(
        new
ValueMap().withString(":v_custid", "bob@example.com")

        .withNumber(":v_orddate",
```

```
20150131));  
  
        querySpec.withSelect(Select.ALL_PROJECTED_ATTRIBUTES);  
  
        ItemCollection<QueryOutcome> items = index.query(querySpec);  
        Iterator<Item> iterator = items.iterator();  
  
        System.out.println("Query: printing results...");  
  
        while (iterator.hasNext()) {  
            System.out.println(iterator.next().toJSONPretty());  
        }  
  
    } else {  
        System.out.println("\nNo index: All of Bob's orders, by  
OrderId:\n");  
  
        querySpec.withKeyConditionExpression("CustomerId = :v_custid")  
                .withValueMap(new  
ValueMap().withString(":v_custid", "bob@example.com"));  
  
        ItemCollection<QueryOutcome> items = table.query(querySpec);  
        Iterator<Item> iterator = items.iterator();  
  
        System.out.println("Query: printing results...");  
  
        while (iterator.hasNext()) {  
            System.out.println(iterator.next().toJSONPretty());  
        }  
  
    }  
  
}  
  
public static void deleteTable(String tableName) {  
  
    Table table = dynamoDB.getTable(tableName);  
    System.out.println("Deleting table " + tableName + "...");  
    table.delete();  
  
    // Wait for table to be deleted  
    System.out.println("Waiting for " + tableName + " to be deleted...");  
    try {  
}
```

```
        table.waitForDelete();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void loadData() {

    Table table = dynamoDB.getTable(tableName);

    System.out.println("Loading data into table " + tableName + "...");

    Item item = new Item().withPrimaryKey("CustomerId",
"alice@example.com").withNumber("OrderId", 1)
                    .withNumber("IsOpen",
1).withNumber("OrderCreationDate", 20150101)
                    .withString("ProductCategory", "Book")
                    .withString("ProductName", "The Great Outdoors")
                    .withString("OrderStatus", "PACKING ITEMS");
    // no ShipmentTrackingId attribute

    PutItemOutcome putItemOutcome = table.putItem(item);

    item = new Item().withPrimaryKey("CustomerId",
"alice@example.com").withNumber("OrderId", 2)
                    .withNumber("IsOpen",
1).withNumber("OrderCreationDate", 20150221)
                    .withString("ProductCategory", "Bike")
                    .withString("ProductName", "Super Mountain")
                    .withString("OrderStatus", "ORDER RECEIVED");
    // no ShipmentTrackingId attribute

    putItemOutcome = table.putItem(item);

    item = new Item().withPrimaryKey("CustomerId",
"alice@example.com").withNumber("OrderId", 3)
                    // no IsOpen attribute
                    .withNumber("OrderCreationDate",
20150304).withString("ProductCategory", "Music")
                    .withString("ProductName", "A Quiet
Interlude").withString("OrderStatus", "IN TRANSIT")
                    .withString("ShipmentTrackingId", "176493");

    putItemOutcome = table.putItem(item);
```

```
        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 1)
                // no IsOpen attribute
                .withNumber("OrderCreationDate",
20150111).withString("ProductCategory", "Movie")
                .withString("ProductName", "Calm Before The Storm")
                .withString("OrderStatus", "SHIPPING DELAY")
                .withString("ShipmentTrackingId", "859323");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 2)
                // no IsOpen attribute
                .withNumber("OrderCreationDate",
20150124).withString("ProductCategory", "Music")
                .withString("ProductName", "E-Z
Listening").withString("OrderStatus", "DELIVERED")
                .withString("ShipmentTrackingId", "756943");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 3)
                // no IsOpen attribute
                .withNumber("OrderCreationDate",
20150221).withString("ProductCategory", "Music")
                .withString("ProductName", "Symphony
9").withString("OrderStatus", "DELIVERED")
                .withString("ShipmentTrackingId", "645193");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 4)
                .withNumber("IsOpen",
1).withNumber("OrderCreationDate", 20150222)
                .withString("ProductCategory", "Hardware")
                .withString("ProductName", "Extra Heavy Hammer")
                .withString("OrderStatus", "PACKING ITEMS");
        // no ShipmentTrackingId attribute

        putItemOutcome = table.putItem(item);
```

```
        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 5)
                /* no IsOpen attribute */
                .withNumber("OrderCreationDate",
20150309).withString("ProductCategory", "Book")
                .withString("ProductName", "How To
Cook").withString("OrderStatus", "IN TRANSIT")
                .withString("ShipmentTrackingId", "440185");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 6)
                // no IsOpen attribute
                .withNumber("OrderCreationDate",
20150318).withString("ProductCategory", "Luggage")
                .withString("ProductName", "Really Big
Suitcase").withString("OrderStatus", "DELIVERED")
                .withString("ShipmentTrackingId", "893927");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 7)
                /* no IsOpen attribute */
                .withNumber("OrderCreationDate",
20150324).withString("ProductCategory", "Golf")
                .withString("ProductName", "PGA Pro
II").withString("OrderStatus", "OUT FOR DELIVERY")
                .withString("ShipmentTrackingId", "383283");

        putItemOutcome = table.putItem(item);
        assert putItemOutcome != null;
    }

}
```

Working with Local Secondary Indexes: .NET

Topics

- [Create a table with a Local Secondary Index](#)

- [Describe a table with a Local Secondary Index](#)
- [Query a Local Secondary Index](#)
- [Example: Local Secondary Indexes using the AWS SDK for .NET low-level API](#)

You can use the AWS SDK for .NET low-level API to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the indexes. These operations map to the corresponding low-level DynamoDB API actions. For more information, see [.NET code examples](#).

The following are the common steps for table operations using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

For example, create a `CreateTableRequest` object to create a table and an `QueryRequest` object to query a table or an index.
3. Run the appropriate method provided by the client that you created in the preceding step.

Create a table with a Local Secondary Index

Local secondary indexes must be created at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more local secondary indexes. The following C# code example creates a table to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex`, facilitates queries by album title.

The following are the steps to create a table with a local secondary index, using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the local secondary index, you must provide the index name, the name and data type of the index sort key, the key schema for the index, and the attribute projection.

3. Run the `CreateTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps. The code creates a table (Music) with a secondary index on the AlbumTitle attribute. The table partition key and sort key, plus the index sort key, are the only attributes projected into the index.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

CreateTableRequest createTableRequest = new CreateTableRequest()
{
    TableName = tableName
};

//ProvisionedThroughput
createTableRequest.ProvisionedThroughput = new ProvisionedThroughput()
{
    ReadCapacityUnits = (long)5,
    WriteCapacityUnits = (long)5
};

//AttributeDefinitions
List<AttributeDefinition> attributeDefinitions = new List<AttributeDefinition>();

attributeDefinitions.Add(new AttributeDefinition()
{
   AttributeName = "Artist",
   AttributeType = "S"
});

attributeDefinitions.Add(new AttributeDefinition()
{
   AttributeName = "SongTitle",
   AttributeType = "S"
});

attributeDefinitions.Add(new AttributeDefinition()
{
   AttributeName = "AlbumTitle",
   AttributeType = "S"
});

createTableRequest.AttributeDefinitions = attributeDefinitions;

//KeySchema
```

```
List<KeySchemaElement> tableKeySchema = new List<KeySchemaElement>();

tableKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType =
    "HASH" }); //Partition key
tableKeySchema.Add(new KeySchemaElement() { AttributeName = "SongTitle", KeyType =
    "RANGE" }); //Sort key

createTableRequest.KeySchema = tableKeySchema;

List<KeySchemaElement> indexKeySchema = new List<KeySchemaElement>();
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType =
    "HASH" }); //Partition key
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "AlbumTitle", KeyType =
    "RANGE" }); //Sort key

Projection projection = new Projection() { ProjectionType = "INCLUDE" };

List<string> nonKeyAttributes = new List<string>();
nonKeyAttributes.Add("Genre");
nonKeyAttributes.Add("Year");
projection.NonKeyAttributes = nonKeyAttributes;

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
{
    IndexName = "AlbumTitleIndex",
    KeySchema = indexKeySchema,
    Projection = projection
};

List<LocalSecondaryIndex> localSecondaryIndexes = new List<LocalSecondaryIndex>();
localSecondaryIndexes.Add(localSecondaryIndex);
createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

CreateTableResponse result = client.CreateTable(createTableRequest);
Console.WriteLine(result.CreateTableResult.TableDescription.TableName);
Console.WriteLine(result.CreateTableResult.TableDescription.TableStatus);
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `DescribeTable` API. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access local secondary index information a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `DescribeTableRequest` class to provide the request information. You must provide the table name.
3. Run the `describeTable` method by providing the request object as a parameter.
- 4.

The following C# code example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest()
{ TableName = tableName });
List<LocalSecondaryIndexDescription> localSecondaryIndexes =
    response.DescribeTableResult.Table.LocalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.
foreach (LocalSecondaryIndexDescription lsiDescription in localSecondaryIndexes)
{
    Console.WriteLine("Info for index " + lsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in lsiDescription.KeySchema)
    {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
    }

    Projection projection = lsiDescription.Projection;

    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);
```

```
if (projection.ProjectionType.ToString().Equals("INCLUDE"))
{
    Console.WriteLine("\t\tThe non-key projected attributes are:");

    foreach (String s in projection.NonKeyAttributes)
    {
        Console.WriteLine("\t\t" + s);
    }
}
```

Query a Local Secondary Index

You can use Query on a local secondary index in much the same way you Query a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex`, and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute projections](#)

The following are the steps to query a local secondary index using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class to provide the request information.
3. Run the `query` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps.

Example

```
QueryRequest queryRequest = new QueryRequest
{
    TableName = "Music",
    IndexName = "AlbumTitleIndex",
    Select = "ALL_ATTRIBUTES",
    ScanIndexForward = true,
    KeyConditionExpression = "Artist = :v_artist and AlbumTitle = :v_title",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
```

```
{  
    ":v_artist", new AttributeValue {S = "Acme Band"}},  
    ":v_title", new AttributeValue {S = "Songs About Life"}}  
,  
};  
  
QueryResponse response = client.Query(queryRequest);  
  
foreach (var attrs in response.Items)  
{  
    foreach (var attrib in attrs)  
    {  
        Console.WriteLine(attrib.Key + " ---> " + attrib.Value.S);  
    }  
    Console.WriteLine();  
}
```

Example: Local Secondary Indexes using the AWS SDK for .NET low-level API

The following C# code example shows how to work with local secondary indexes in Amazon DynamoDB. The example creates a table named `CustomerOrders` with a partition key of `CustomerId` and a sort key of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex` — The sort key is `OrderCreationDate`, and the following attributes are projected into the index:
 - `ProductCategory`
 - `ProductName`
 - `OrderStatus`
 - `ShipmentTrackingId`
- `IsOpenIndex` — The sort key is `IsOpen`, and all of the table attributes are projected into the index.

After the `CustomerOrders` table is created, the program loads the table with data representing customer orders. It then queries the data using the local secondary indexes. Finally, the program deletes the `CustomerOrders` table.

For step-by-step instructions for testing the following example, see [.NET code examples](#).

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelLocalSecondaryIndexExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "CustomerOrders";

        static void Main(string[] args)
        {
            try
            {
                CreateTable();
                LoadData();

                Query(null);
                Query("IsOpenIndex");
                Query("OrderCreationDateIndex");

                DeleteTable(tableName);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void CreateTable()
        {
            var createTableRequest =
                new CreateTableRequest()
```

```
{  
    TableName = tableName,  
    ProvisionedThroughput =  
        new ProvisionedThroughput()  
    {  
        ReadCapacityUnits = (long)1,  
        WriteCapacityUnits = (long)1  
    }  
};  
  
var attributeDefinitions = new List<AttributeDefinition>()  
{  
    // Attribute definitions for table primary key  
    { new AttributeDefinition() {  
        AttributeName = "CustomerId", AttributeType = "S"  
    } },  
    { new AttributeDefinition() {  
        AttributeName = "OrderId", AttributeType = "N"  
    } },  
    // Attribute definitions for index primary key  
    { new AttributeDefinition() {  
        AttributeName = "OrderCreationDate", AttributeType = "N"  
    } },  
    { new AttributeDefinition() {  
        AttributeName = "IsOpen", AttributeType = "N"  
    } }  
};  
  
createTableRequest.AttributeDefinitions = attributeDefinitions;  
  
// Key schema for table  
var tableKeySchema = new List<KeySchemaElement>()  
{  
    { new KeySchemaElement() {  
        AttributeName = "CustomerId", KeyType = "HASH"  
    } },  
        //Partition key  
    { new KeySchemaElement() {  
        AttributeName = "OrderId", KeyType = "RANGE"  
    } }  
        //Sort key  
};  
  
createTableRequest.KeySchema = tableKeySchema;  
  
var localSecondaryIndexes = new List<LocalSecondaryIndex>();
```

```
// OrderCreationDateIndex
LocalSecondaryIndex orderCreationDateIndex = new LocalSecondaryIndex()
{
    IndexName = "OrderCreationDateIndex"
};

// Key schema for OrderCreationDateIndex
var indexKeySchema = new List<KeySchemaElement>()
{
    { new KeySchemaElement() {
        AttributeName = "CustomerId", KeyType = "HASH"
    } }, //Partition key
    { new KeySchemaElement() {
        AttributeName = "OrderCreationDate", KeyType = "RANGE"
    } } //Sort key
};

orderCreationDateIndex.KeySchema = indexKeySchema;

// Projection (with list of projected attributes) for
// OrderCreationDateIndex
var projection = new Projection()
{
    ProjectionType = "INCLUDE"
};

var nonKeyAttributes = new List<string>()
{
    "ProductCategory",
    "ProductName"
};
projection.NonKeyAttributes = nonKeyAttributes;

orderCreationDateIndex.Projection = projection;

localSecondaryIndexes.Add(orderCreationDateIndex);

// IsOpenIndex
LocalSecondaryIndex isOpenIndex
    = new LocalSecondaryIndex()
    {
        IndexName = "IsOpenIndex"
    };
}
```

```
// Key schema for IsOpenIndex
indexKeySchema = new List<KeySchemaElement>()
{
    { new KeySchemaElement() {
        AttributeName = "CustomerId", KeyType = "HASH"
    }, //Partition key
    { new KeySchemaElement() {
        AttributeName = "IsOpen", KeyType = "RANGE"
    }} //Sort key
};

// Projection (all attributes) for IsOpenIndex
projection = new Projection()
{
    ProjectionType = "ALL"
};

isOpenIndex.KeySchema = indexKeySchema;
isOpenIndex.Projection = projection;

localSecondaryIndexes.Add(isOpenIndex);

// Add index definitions to CreateTable request
createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

Console.WriteLine("Creating table " + tableName + "...");
client.CreateTable(createTableRequest);
WaitUntilTableReady(tableName);
}

public static void Query(string indexName)
{

Console.WriteLine("\n*****\n");
Console.WriteLine("Querying table " + tableName + "...");

QueryRequest queryRequest = new QueryRequest()
{
    TableName = tableName,
    ConsistentRead = true,
    ScanIndexForward = true,
    ReturnConsumedCapacity = "TOTAL"
};
```

```
String keyConditionExpression = "CustomerId = :v_customerId";
Dictionary<string, AttributeValue> expressionAttributeValues = new
Dictionary<string, AttributeValue> {
    {":v_customerId", new AttributeValue {
        S = "bob@example.com"
    }}
};

if (indexName == "IsOpenIndex")
{
    Console.WriteLine("\nUsing index: '" + indexName
        + "' Bob's orders that are open.");
    Console.WriteLine("Only a user-specified list of attributes are
returned\n");
    queryRequest.IndexName = indexName;

    keyConditionExpression += " and IsOpen = :v_isOpen";
    expressionAttributeValues.Add(":v_isOpen", new AttributeValue
    {
        N = "1"
    });

    // ProjectionExpression
    queryRequest.ProjectionExpression = "OrderCreationDate,
ProductCategory, ProductName, OrderStatus";
}
else if (indexName == "OrderCreationDateIndex")
{
    Console.WriteLine("\nUsing index: '" + indexName
        + "' Bob's orders that were placed after 01/31/2013.");
    Console.WriteLine("Only the projected attributes are returned\n");
    queryRequest.IndexName = indexName;

    keyConditionExpression += " and OrderCreationDate > :v_Date";
    expressionAttributeValues.Add(":v_Date", new AttributeValue
    {
        N = "20130131"
    });

    // Select
    queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
```

```
        }
    else
    {
        Console.WriteLine("\nNo index: All of Bob's orders, by OrderId:\n");
    }
queryRequest.KeyConditionExpression = keyConditionExpression;
queryRequest.ExpressionAttributeValues = expressionAttributeValues;

var result = client.Query(queryRequest);
var items = result.Items;
foreach (var currentItem in items)
{
    foreach (string attr in currentItem.Keys)
    {
        if (attr == "OrderId" || attr == "IsOpen"
            || attr == "OrderCreationDate")
        {
            Console.WriteLine(attr + "---> " + currentItem[attr].N);
        }
        else
        {
            Console.WriteLine(attr + "---> " + currentItem[attr].S);
        }
    }
    Console.WriteLine();
}
Console.WriteLine("\nConsumed capacity: " +
result.ConsumedCapacity.CapacityUnits + "\n");
}

private static void DeleteTable(string tableName)
{
    Console.WriteLine("Deleting table " + tableName + "...");
    client.DeleteTable(new DeleteTableRequest()
    {
        TableName = tableName
    });
    WaitForTableToDelete(tableName);
}

public static void LoadData()
{
    Console.WriteLine("Loading data into table " + tableName + "...");
```

```
Dictionary<string, AttributeValue> item = new Dictionary<string,
AttributeValue>();

item["CustomerId"] = new AttributeValue
{
    S = "alice@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "1"
};
item["IsOpen"] = new AttributeValue
{
    N = "1"
};
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130101"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Book"
};
item["ProductName"] = new AttributeValue
{
    S = "The Great Outdoors"
};
item["OrderStatus"] = new AttributeValue
{
    S = "PACKING ITEMS"
};
/* no ShipmentTrackingId attribute */
PutItemRequest putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "alice@example.com"
}
```

```
};

item["OrderId"] = new AttributeValue
{
    N = "2"
};

item["IsOpen"] = new AttributeValue
{
    N = "1"
};

item["OrderCreationDate"] = new AttributeValue
{
    N = "20130221"
};

item["ProductCategory"] = new AttributeValue
{
    S = "Bike"
};

item["ProductName"] = new AttributeValue
{
    S = "Super Mountain"
};

item["OrderStatus"] = new AttributeValue
{
    S = "ORDER RECEIVED"
};

/* no ShipmentTrackingId attribute */
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};

client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "alice@example.com"
};

item["OrderId"] = new AttributeValue
{
    N = "3"
};

/* no IsOpen attribute */
```

```
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130304"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Music"
};
item["ProductName"] = new AttributeValue
{
    S = "A Quiet Interlude"
};
item["OrderStatus"] = new AttributeValue
{
    S = "IN TRANSIT"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "176493"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "1"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130111"
};
item["ProductCategory"] = new AttributeValue
{
```

```
S = "Movie"
};

item["ProductName"] = new AttributeValue
{
    S = "Calm Before The Storm"
};
item["OrderStatus"] = new AttributeValue
{
    S = "SHIPPING DELAY"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "859323"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "2"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130124"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Music"
};
item["ProductName"] = new AttributeValue
{
    S = "E-Z Listening"
};
```

```
item["OrderStatus"] = new AttributeValue
{
    S = "DELIVERED"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "756943"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "3"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130221"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Music"
};
item["ProductName"] = new AttributeValue
{
    S = "Symphony 9"
};
item["OrderStatus"] = new AttributeValue
{
    S = "DELIVERED"
};
item["ShipmentTrackingId"] = new AttributeValue
{
```

```
S = "645193"
};

putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "4"
};
item["IsOpen"] = new AttributeValue
{
    N = "1"
};
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130222"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Hardware"
};
item["ProductName"] = new AttributeValue
{
    S = "Extra Heavy Hammer"
};
item["OrderStatus"] = new AttributeValue
{
    S = "PACKING ITEMS"
};
/* no ShipmentTrackingId attribute */
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
```

```
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue
    {
        S = "bob@example.com"
    };
    item["OrderId"] = new AttributeValue
    {
        N = "5"
    };
    /* no IsOpen attribute */
    item["OrderCreationDate"] = new AttributeValue
    {
        N = "20130309"
    };
    item["ProductCategory"] = new AttributeValue
    {
        S = "Book"
    };
    item["ProductName"] = new AttributeValue
    {
        S = "How To Cook"
    };
    item["OrderStatus"] = new AttributeValue
    {
        S = "IN TRANSIT"
    };
    item["ShipmentTrackingId"] = new AttributeValue
    {
        S = "440185"
    };
    putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue
```

```
{  
    S = "bob@example.com"  
};  
item["OrderId"] = new AttributeValue  
{  
    N = "6"  
};  
/* no IsOpen attribute */  
item["OrderCreationDate"] = new AttributeValue  
{  
    N = "20130318"  
};  
item["ProductCategory"] = new AttributeValue  
{  
    S = "Luggage"  
};  
item["ProductName"] = new AttributeValue  
{  
    S = "Really Big Suitcase"  
};  
item["OrderStatus"] = new AttributeValue  
{  
    S = "DELIVERED"  
};  
item["ShipmentTrackingId"] = new AttributeValue  
{  
    S = "893927"  
};  
putItemRequest = new PutItemRequest  
{  
    TableName = tableName,  
    Item = item,  
    ReturnItemCollectionMetrics = "SIZE"  
};  
client.PutItem(putItemRequest);  
  
item = new Dictionary<string, AttributeValue>();  
item["CustomerId"] = new AttributeValue  
{  
    S = "bob@example.com"  
};  
item["OrderId"] = new AttributeValue  
{  
    N = "7"
```

```
};

/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130324"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Golf"
};
item["ProductName"] = new AttributeValue
{
    S = "PGA Pro II"
};
item["OrderStatus"] = new AttributeValue
{
    S = "OUT FOR DELIVERY"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "383283"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });
        }
    }
}
```

```
        Console.WriteLine("Table name: {0}, status: {1}",
                           res.Table.TableName,
                           res.Table.TableStatus);
        status = res.Table.TableStatus;
    }
    catch (ResourceNotFoundException)
    {
        // DescribeTable is eventually consistent. So you might
        // get resource not found. So we handle the potential exception.
    }
} while (status != "ACTIVE");

}

private static void WaitForTableToDelete(string tableName)
{
    bool tablePresent = true;

    while (tablePresent)
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                           res.Table.TableName,
                           res.Table.TableStatus);
        }
        catch (ResourceNotFoundException)
        {
            tablePresent = false;
        }
    }
}
}
```

Working with Local Secondary Indexes: AWS CLI

You can use the AWS CLI to create an Amazon DynamoDB table with one or more Local Secondary Indexes, describe the indexes on the table, and perform queries using the indexes.

Topics

- [Create a table with a Local Secondary Index](#)
- [Describe a table with a Local Secondary Index](#)
- [Query a Local Secondary Index](#)

Create a table with a Local Secondary Index

Local Secondary Indexes must be created at the same time you create a table. To do this, use the `create-table` parameter and provide your specifications for one or more Local Secondary Indexes. The following example creates a table (`Music`) to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex` on the `AlbumTitle` attribute facilitates queries by album title.

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions AttributeName=Artist,AttributeType=S
  AttributeName=SongTitle,AttributeType=S \
    AttributeName=AlbumTitle,AttributeType=S \
  --key-schema AttributeName=Artist,KeyType=HASH
  AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --local-secondary-indexes \
    "[{\\"IndexName\\": \\"AlbumTitleIndex\\",
    \\"KeySchema\\": [{\\"AttributeName\\":\\"Artist\\",\\"KeyType\\":\\"HASH\\"},
      {\\"AttributeName\\":\\"AlbumTitle\\",\\"KeyType\\":\\"RANGE\\"}],
    \\"Projection\\":{\\"ProjectionType\\":\\"INCLUDE\\", \\"NonKeyAttributes\\":[\\"Genre\\",
      \\"Year\\"]}}]"
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table. You can use [describe-table](#) to determine the status of the table creation.

Describe a table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `describe-table` parameter. For each index, you can access its name, key schema, and projected attributes.

```
aws dynamodb describe-table --table-name Music
```

Query a Local Secondary Index

You can use the `query` operation on a Local Secondary Index in much the same way that you query a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute projections](#).

```
aws dynamodb query \  
  --table-name Music \  
  --index-name AlbumTitleIndex \  
  --key-condition-expression "Artist = :v_artist and AlbumTitle = :v_title" \  
  --expression-attribute-values '{":v_artist":{"S":"Acme Band"},":v_title":  
  {"S":"Songs About Life"} }'
```

Managing complex workflows with DynamoDB transactions

Amazon DynamoDB transactions simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables. Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB, helping you to maintain data correctness in your applications.

You can use the DynamoDB transactional read and write APIs to manage complex business workflows that require adding, updating, or deleting multiple items as a single, all-or-nothing operation. For example, a video game developer can ensure that players' profiles are updated correctly when they exchange items in a game or make in-game purchases.

With the transaction write API, you can group multiple `Put`, `Update`, `Delete`, and `ConditionCheck` actions. You can then submit the actions as a single `TransactWriteItems`

operation that either succeeds or fails as a unit. The same is true for multiple Get actions, which you can group and submit as a single `TransactGetItems` operation.

There is no additional cost to enable transactions for your DynamoDB tables. You pay only for the reads or writes that are part of your transaction. DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction. These two underlying read/write operations are visible in your Amazon CloudWatch metrics.

To get started with DynamoDB transactions, download the latest AWS SDK or the AWS Command Line Interface (AWS CLI). Then follow the [DynamoDB transactions example](#).

The following sections provide a detailed overview of the transaction APIs and how you can use them in DynamoDB.

Topics

- [Amazon DynamoDB Transactions: How it works](#)
- [Using IAM with DynamoDB transactions](#)
- [DynamoDB transactions example](#)

Amazon DynamoDB Transactions: How it works

With Amazon DynamoDB transactions, you can group multiple actions together and submit them as a single all-or-nothing `TransactWriteItems` or `TransactGetItems` operation. The following sections describe API operations, capacity management, best practices, and other details about using transactional operations in DynamoDB.

Topics

- [TransactWriteItems API](#)
- [TransactGetItems API](#)
- [Isolation levels for DynamoDB transactions](#)
- [Transaction conflict handling in DynamoDB](#)
- [Using transactional APIs in DynamoDB Accelerator \(DAX\)](#)
- [Capacity management for transactions](#)
- [Best practices for transactions](#)
- [Using transactional APIs with global tables](#)

- [DynamoDB Transactions vs. the AWSLabs transactions client library](#)

TransactWriteItems API

TransactWriteItems is a synchronous and idempotent write operation that groups up to 100 write actions in a single all-or-nothing operation. These actions can target up to 100 distinct items in one or more DynamoDB tables within the same AWS account and in the same Region. The aggregate size of the items in the transaction cannot exceed 4 MB. The actions are completed atomically so that either all of them succeed or none of them succeeds.

Note

- A TransactWriteItems operation differs from a BatchWriteItem operation in that all the actions it contains must be completed successfully, or no changes are made at all. With a BatchWriteItem operation, it is possible that only some of the actions in the batch succeed while the others do not.
- Transactions cannot be performed using indexes.

You can't target the same item with multiple operations within the same transaction. For example, you can't perform a ConditionCheck and also an Update action on the same item in the same transaction.

You can add the following types of actions to a transaction:

- Put — Initiates a PutItem operation to create a new item or replace an old item with a new item, conditionally or without specifying any condition.
- Update — Initiates an UpdateItem operation to edit an existing item's attributes or add a new item to the table if it does not already exist. Use this action to add, delete, or update attributes on an existing item conditionally or without a condition.
- Delete — Initiates a DeleteItem operation to delete a single item in a table identified by its primary key.
- ConditionCheck — Checks that an item exists or checks the condition of specific attributes of the item.

Once a transaction completes, the changes made within that transaction are propagated to global secondary indexes (GSIs), streams, and backups. Since propagation is not immediate or instantaneous, if a table is restored from backup ([RestoreTableFromBackup](#)) or exported to a point in time ([ExportTableToPointInTime](#)) mid-propagation, it might contain some but not all of the changes made during a recent transaction.

Idempotency

You can optionally include a client token when you make a `TransactWriteItems` call to ensure that the request is *idempotent*. Making your transactions idempotent helps prevent application errors if the same operation is submitted multiple times due to a connection time-out or other connectivity issue.

If the original `TransactWriteItems` call was successful, then subsequent `TransactWriteItems` calls with the same client token return successfully without making any changes. If the `ReturnConsumedCapacity` parameter is set, the initial `TransactWriteItems` call returns the number of write capacity units consumed in making the changes. Subsequent `TransactWriteItems` calls with the same client token return the number of read capacity units consumed in reading the item.

Important points about idempotency

- A client token is valid for 10 minutes after the request that uses it finishes. After 10 minutes, any request that uses the same client token is treated as a new request. You should not reuse the same client token for the same request after 10 minutes.
- If you repeat a request with the same client token within the 10-minute idempotency window but change some other request parameter, DynamoDB returns an `IdempotentParameterMismatch` exception.

Error handling for writing

Write transactions don't succeed under the following circumstances:

- When a condition in one of the condition expressions is not met.
- When a transaction validation error occurs because more than one action in the same `TransactWriteItems` operation targets the same item.

- When a `TransactWriteItems` request conflicts with an ongoing `TransactWriteItems` operation on one or more items in the `TransactWriteItems` request. In this case, the request fails with a `TransactionCanceledException`.
- When there is insufficient provisioned capacity for the transaction to be completed.
- When an item size becomes too large (larger than 400 KB), or a local secondary index (LSI) becomes too large, or a similar validation error occurs because of changes made by the transaction.
- When there is a user error, such as an invalid data format.

For more information about how conflicts with `TransactWriteItems` operations are handled, see [Transaction conflict handling in DynamoDB](#).

TransactGetItems API

`TransactGetItems` is a synchronous read operation that groups up to 100 Get actions together. These actions can target up to 100 distinct items in one or more DynamoDB tables within the same AWS account and Region. The aggregate size of the items in the transaction can't exceed 4 MB.

The Get actions are performed atomically so that either all of them succeed or all of them fail:

- Get — Initiates a `GetItem` operation to retrieve a set of attributes for the item with the given primary key. If no matching item is found, Get does not return any data.

Error handling for reading

Read transactions don't succeed under the following circumstances:

- When a `TransactGetItems` request conflicts with an ongoing `TransactWriteItems` operation on one or more items in the `TransactGetItems` request. In this case, the request fails with a `TransactionCanceledException`.
- When there is insufficient provisioned capacity for the transaction to be completed.
- When there is a user error, such as an invalid data format.

For more information about how conflicts with `TransactGetItems` operations are handled, see [Transaction conflict handling in DynamoDB](#).

Isolation levels for DynamoDB transactions

The isolation levels of transactional operations (`TransactWriteItems` or `TransactGetItems`) and other operations are as follows.

SERIALIZABLE

Serializable isolation ensures that the results of multiple concurrent operations are the same as if no operation begins until the previous one has finished.

There is serializable isolation between the following types of operation:

- Between any transactional operation and any standard write operation (`PutItem`, `UpdateItem`, or `DeleteItem`).
- Between any transactional operation and any standard read operation (`GetItem`).
- Between a `TransactWriteItems` operation and a `TransactGetItems` operation.

Although there is serializable isolation between transactional operations, and each individual standard write in a `BatchWriteItem` operation, there is no serializable isolation between the transaction and the `BatchWriteItem` operation as a unit.

Similarly, the isolation level between a transactional operation and individual `GetItems` in a `BatchGetItem` operation is serializable. But the isolation level between the transaction and the `BatchGetItem` operation as a unit is *read-committed*.

A single `GetItem` request is serializable with respect to a `TransactWriteItems` request in one of two ways, either before or after the `TransactWriteItems` request. Multiple `GetItem` requests, against keys in a concurrent `TransactWriteItems` requests can be run in any order, and therefore the results are *read-committed*.

For example, if `GetItem` requests for item A and item B are run concurrently with a `TransactWriteItems` request that modifies both item A and item B, there are four possibilities:

- Both `GetItem` requests are run before the `TransactWriteItems` request.
- Both `GetItem` requests are run after the `TransactWriteItems` request.
- `GetItem` request for item A is run before the `TransactWriteItems` request. For item B the `GetItem` is run after `TransactWriteItems`.
- `GetItem` request for item B is run before the `TransactWriteItems` request. For item A the `GetItem` is run after `TransactWriteItems`.

You should use `TransactGetItems` if you prefer serializable isolation level for multiple `GetItem` requests.

If a non-transactional read is made on multiple items that were part of the same transaction write request in-flight, it is possible that you'll be able to read new state of some of the items and old state of the other items. You'll be able to read new state of all items that were part of the transaction write request only when a successful response is received for the transactional write.

READ-COMMITTED

Read-committed isolation ensures that read operations always return committed values for an item - the read will never present a view to the item representing a state from a transactional write which did not ultimately succeed. Read-committed isolation does not prevent modifications of the item immediately after the read operation.

The isolation level is read-committed between any transactional operation and any read operation that involves multiple standard reads (`BatchGetItem`, `Query`, or `Scan`). If a transactional write updates an item in the middle of a `BatchGetItem`, `Query`, or `Scan` operation, the subsequent part of the read operation returns the newly committed value (with `ConsistentRead`) or possibly a prior committed value (eventually consistent reads).

Operation summary

To summarize, the following table shows the isolation levels between a transaction operation (`TransactWriteItems` or `TransactGetItems`) and other operations.

Operation	Isolation Level
<code>DeleteItem</code>	<i>Serializable</i>
<code>PutItem</code>	<i>Serializable</i>
<code>UpdateItem</code>	<i>Serializable</i>
<code>GetItem</code>	<i>Serializable</i>
<code>BatchGetItem</code>	<i>Read-committed*</i>
<code>BatchWriteItem</code>	<i>NOT Serializable*</i>
<code>Query</code>	<i>Read-committed</i>

Operation	Isolation Level
Scan	<i>Read-committed</i>
Other transactional operation	<i>Serializable</i>

Levels marked with an asterisk (*) apply to the operation as a unit. However, individual actions within those operations have a *serializable* isolation level.

Transaction conflict handling in DynamoDB

A transactional conflict can occur during concurrent item-level requests on an item within a transaction. Transaction conflicts can occur in the following scenarios:

- A PutItem, UpdateItem, or DeleteItem request for an item conflicts with an ongoing TransactWriteItems request that includes the same item.
- An item within a TransactWriteItems request is part of another ongoing TransactWriteItems request.
- An item within a TransactGetItems request is part of an ongoing TransactWriteItems, BatchWriteItem, PutItem, UpdateItem, or DeleteItem request.

Note

- When a PutItem, UpdateItem, or DeleteItem request is rejected, the request fails with a TransactionConflictException.
- If any item-level request within TransactWriteItems or TransactGetItems is rejected, the request fails with a TransactionCanceledException. If that request fails, AWS SDKs do not retry the request.

If you are using the AWS SDK for Java, the exception contains the list of [CancellationReasons](#), ordered according to the list of items in the TransactItems request parameter. For other languages, a string representation of the list is included in the exception's error message.

- If an ongoing TransactWriteItems or TransactGetItems operation conflicts with a concurrent GetItem request, both operations can succeed.

The [TransactionConflict CloudWatch metric](#) is incremented for each failed item-level request.

Using transactional APIs in DynamoDB Accelerator (DAX)

`TransactWriteItems` and `TransactGetItems` are both supported in DynamoDB Accelerator (DAX) with the same isolation levels as in DynamoDB.

`TransactWriteItems` writes through DAX. DAX passes a `TransactWriteItems` call to DynamoDB and returns the response. To populate the cache after the write, DAX calls `TransactGetItems` in the background for each item in the `TransactWriteItems` operation, which consumes additional read capacity units. (For more information, see [Capacity management for transactions](#).) This functionality enables you to keep your application logic simple and use DAX for both transactional operations and nontransactional ones.

`TransactGetItems` calls are passed through DAX without the items being cached locally. This is the same behavior as for strongly consistent read APIs in DAX.

Capacity management for transactions

There is no additional cost to enable transactions for your DynamoDB tables. You pay only for the reads or writes that are part of your transaction. DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction. The two underlying read/write operations are visible in your Amazon CloudWatch metrics.

Plan for the additional reads and writes that are required by transactional APIs when you are provisioning capacity to your tables. For example, suppose that your application runs one transaction per second, and each transaction writes three 500-byte items in your table. Each item requires two write capacity units (WCUs): one to prepare the transaction and one to commit the transaction. Therefore, you would need to provision six WCUs to the table.

If you were using DynamoDB Accelerator (DAX) in the previous example, you would also use two read capacity units (RCUs) for each item in the `TransactWriteItems` call. So you would need to provision six additional RCUs to the table.

Similarly, if your application runs one read transaction per second, and each transaction reads three 500-byte items in your table, you would need to provision six read capacity units (RCUs) to the table. Reading each item requires two RCUs: one to prepare the transaction and one to commit the transaction.

Also, default SDK behavior is to retry transactions in case of a `TransactionInProgressException` exception. Plan for the additional read-capacity units (RCUs) that these retries consume. The same is true if you are retrying transactions in your own code using a `ClientRequestToken`.

Best practices for transactions

Consider the following recommended practices when using DynamoDB transactions.

- Enable automatic scaling on your tables, or ensure that you have provisioned enough throughput capacity to perform the two read or write operations for every item in your transaction.
- If you are not using an AWS provided SDK, include a `ClientRequestToken` attribute when you make a `TransactWriteItems` call to ensure that the request is idempotent.
- Don't group operations together in a transaction if it's not necessary. For example, if a single transaction with 10 operations can be broken up into multiple transactions without compromising the application correctness, we recommend splitting up the transaction. Simpler transactions improve throughput and are more likely to succeed.
- Multiple transactions updating the same items simultaneously can cause conflicts that cancel the transactions. We recommend following DynamoDB best practices for data modeling to minimize such conflicts.
- If a set of attributes is often updated across multiple items as part of a single transaction, consider grouping the attributes into a single item to reduce the scope of the transaction.
- Avoid using transactions for ingesting data in bulk. For bulk writes, it is better to use `BatchWriteItem`.

Using transactional APIs with global tables

Operations contained within a DynamoDB transaction are only guaranteed transactional in the region where the transaction is originally executed. Transactionality is not preserved when changes applied within a transaction are replicated across Regions to global tables replicas.

DynamoDB Transactions vs. the AWSLabs transactions client library

DynamoDB transactions provide a more cost-effective, robust, and performant replacement for the [AWSLabs](#) transactions client library. We suggest that you update your applications to use the native, server-side transaction APIs.

Using IAM with DynamoDB transactions

You can use AWS Identity and Access Management (IAM) to restrict the actions that transactional operations can perform in Amazon DynamoDB. For more information about using IAM policies in DynamoDB, see [Identity-based policies for DynamoDB](#).

Permissions for Put, Update, Delete, and Get actions are governed by the permissions used for the underlying PutItem, UpdateItem, DeleteItem, and GetItem operations. For the ConditionCheck action, you can use the dynamodb:ConditionCheck permission in IAM policies.

The following are examples of IAM policies that you can use to configure the DynamoDB transactions.

Example 1: Allow transactional operations

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:ConditionCheckItem",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:GetItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:*:*:table/table04"  
            ]  
        }  
    ]  
}
```

Example 2: Allow only transactional operations

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb:ConditionCheckItem"],  
            "Resource": [  
                "arn:aws:dynamodb:*:*:table/table04"  
            ]  
        }  
    ]  
}
```

```
"Effect": "Allow",
"Action": [
    "dynamodb:ConditionCheckItem",
    "dynamodb:PutItem",
    "dynamodb:UpdateItem",
    "dynamodb>DeleteItem",
    "dynamodb:GetItem"
],
"Resource": [
    "arn:aws:dynamodb:*:*:table/table04"
],
"Condition": {
    "ForAnyValue:StringEquals": {
        "dynamodb:EnclosingOperation": [
            "TransactWriteItems",
            "TransactGetItems"
        ]
    }
}
]
}
```

Example 3: Allow nontransactional reads and writes, and block transactional reads and writes

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb>DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
```

```
        "dynamodb:EnclosingOperation": [
            "TransactWriteItems",
            "TransactGetItems"
        ],
    }
},
{
    "Effect": "Allow",
    "Action": [
        "dynamodb:PutItem",
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:UpdateItem"
    ],
    "Resource": [
        "arn:aws:dynamodb:*:*:table/table04"
    ]
}
]
```

Example 4: Prevent information from being returned on a ConditionCheck failure

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/table01",
            "Condition": {
                "StringEqualsIfExists": {
                    "dynamodb:ReturnValues": "NONE"
                }
            }
        }
    ]
}
```

```
    ]  
}
```

DynamoDB transactions example

As an example of a situation in which Amazon DynamoDB transactions can be useful, consider this sample Java application for an online marketplace.

The application has three DynamoDB tables in the backend:

- **Customers** — This table stores details about the marketplace customers. Its primary key is a `CustomerId` unique identifier.
- **ProductCatalog** — This table stores details such as price and availability about the products for sale in the marketplace. Its primary key is a `ProductId` unique identifier.
- **Orders** — This table stores details about orders from the marketplace. Its primary key is an `OrderId` unique identifier.

Making an order

The following code snippets illustrate how to use DynamoDB transactions to coordinate the multiple steps that are required to create and process an order. Using a single all-or-nothing operation ensures that if any part of the transaction fails, no actions in the transaction are run and no changes are made.

In this example, you set up an order from a customer whose `customerId` is `09e8e9c8-ec48`. You then run it as a single transaction using the following simple order-processing workflow:

1. Determine that the customer ID is valid.
2. Make sure that the product is `IN_STOCK`, and update the product status to `SOLD`.
3. Make sure that the order does not already exist, and create the order.

Validate the customer

First, define an action to verify that a customer with `customerId` equal to `09e8e9c8-ec48` exists in the customer table.

```
final String CUSTOMER_TABLE_NAME = "Customers";  
final String CUSTOMER_PARTITION_KEY = "CustomerId";
```

```
final String customerId = "09e8e9c8-ec48";
final HashMap<String, AttributeValue> customerItemKey = new HashMap<>();
customerItemKey.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));

ConditionCheck checkCustomerValid = new ConditionCheck()
    .withTableName(CUSTOMER_TABLE_NAME)
    .withKey(customerItemKey)
    .withConditionExpression("attribute_exists(" + CUSTOMER_PARTITION_KEY + ")");
```

Update the product status

Next, define an action to update the product status to SOLD if the condition that the product status is currently set to IN_STOCK is true. Setting the `ReturnValuesOnConditionCheckFailure` parameter returns the item if the item's product status attribute was not equal to IN_STOCK.

```
final String PRODUCT_TABLE_NAME = "ProductCatalog";
final String PRODUCT_PARTITION_KEY = "ProductId";
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
expressionAttributeValues.put(":new_status", new AttributeValue("SOLD"));
expressionAttributeValues.put(":expected_status", new AttributeValue("IN_STOCK"));

Update markItemSold = new Update()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey)
    .withUpdateExpression("SET ProductStatus = :new_status")
    .withExpressionAttributeValues(expressionAttributeValues)
    .withConditionExpression("ProductStatus = :expected_status")

    .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD);
```

Create the order

Lastly, create the order as long as an order with that OrderId does not already exist.

```
final String ORDER_PARTITION_KEY = "OrderId";
final String ORDER_TABLE_NAME = "Orders";

HashMap<String, AttributeValue> orderItem = new HashMap<>();
orderItem.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));
orderItem.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));
```

```
orderItem.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));
orderItem.put("OrderStatus", new AttributeValue("CONFIRMED"));
orderItem.put("OrderTotal", new AttributeValue("100"));

Put createOrder = new Put()
    .withTableName(ORDER_TABLE_NAME)
    .withItem(orderItem)

    .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD)
    .withConditionExpression("attribute_not_exists(" + ORDER_PARTITION_KEY + ")");
```

Run the transaction

The following example illustrates how to run the actions defined previously as a single all-or-nothing operation.

```
Collection<TransactWriteItem> actions = Arrays.asList(
    new TransactWriteItem().withConditionCheck(checkCustomerValid),
    new TransactWriteItem().withUpdate(markItemSold),
    new TransactWriteItem().withPut(createOrder));

TransactWriteItemsRequest placeOrderTransaction = new TransactWriteItemsRequest()
    .withTransactItems(actions)
    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

// Run the transaction and process the result.
try {
    client.transactWriteItems(placeOrderTransaction);
    System.out.println("Transaction Successful");

} catch (ResourceNotFoundException rnf) {
    System.err.println("One of the table involved in the transaction is not found"
+ rnf.getMessage());
} catch (InternalServerException ise) {
    System.err.println("Internal Server Error" + ise.getMessage());
} catch (TransactionCanceledException tce) {
    System.out.println("Transaction Canceled " + tce.getMessage());
}
```

Reading the order details

The following example shows how to read the completed order transactionally across the Orders and ProductCatalog tables.

```
HashMap<String,AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

HashMap<String,AttributeValue> orderKey = new HashMap<>();
orderKey.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));

Get readProductSold = new Get()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey);
Get readCreatedOrder = new Get()
    .withTableName(ORDER_TABLE_NAME)
    .withKey(orderKey);

Collection<TransactGetItem> getActions = Arrays.asList(
    new TransactGetItem().withGet(readProductSold),
    new TransactGetItem().withGet(readCreatedOrder));

TransactGetItemsRequest readCompletedOrder = new TransactGetItemsRequest()
    .withTransactItems(getActions)
    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

// Run the transaction and process the result.
try {
    TransactGetItemsResult result = client.transactGetItems(readCompletedOrder);
    System.out.println(result.getResponses());
} catch (ResourceNotFoundException rnf) {
    System.err.println("One of the table involved in the transaction is not found" +
        rnf.getMessage());
} catch (InternalServerException ise) {
    System.err.println("Internal Server Error" + ise.getMessage());
} catch (TransactionCanceledException tce) {
    System.err.println("Transaction Canceled" + tce.getMessage());
}
```

Additional examples

- [Using transactions from DynamoDBMapper](#)

Change data capture with Amazon DynamoDB

Many applications benefit from capturing changes to items stored in a DynamoDB table, at the point in time when such changes occur. The following are some example use cases:

- A popular mobile app modifies data in a DynamoDB table, at the rate of thousands of updates per second. Another application captures and stores data about these updates, providing near-real-time usage metrics for the mobile app.
- A financial application modifies stock market data in a DynamoDB table. Different applications running in parallel track these changes in real time, compute value-at-risk, and automatically rebalance portfolios based on stock price movements.
- Sensors in transportation vehicles and industrial equipment send data to a DynamoDB table. Different applications monitor performance and send messaging alerts when a problem is detected, predict any potential defects by applying machine learning algorithms, and compress and archive data to Amazon Simple Storage Service (Amazon S3).
- An application automatically sends notifications to the mobile devices of all friends in a group as soon as one friend uploads a new picture.
- A new customer adds data to a DynamoDB table. This event invokes another application that sends a welcome email to the new customer.

DynamoDB supports streaming of item-level change data capture records in near-real time. You can build applications that consume these streams and take action based on the contents.

The following video will give you an introductory look at the change data capture concept.

[Table capacity modes](#)

Topics

- [Streaming options for change data capture](#)
- [Using Kinesis Data Streams to capture changes to DynamoDB](#)
- [Change data capture for DynamoDB Streams](#)

Streaming options for change data capture

DynamoDB offers two streaming models for change data capture: Kinesis Data Streams for DynamoDB and DynamoDB Streams.

To help you choose the right solution for your application, the following table summarizes the features of each streaming model.

Properties	Kinesis Data Streams for DynamoDB	DynamoDB Streams
Data retention	Up to 1 year .	24 hours.
Kinesis Client Library (KCL) support	Supports KCL versions 1.X and 2.X .	Supports KCL version 1.X .
Number of consumers	Up to 5 simultaneous consumers per shard, or up to 20 simultaneous consumers per shard with enhanced fan-out .	Up to 2 simultaneous consumers per shard.
Throughput quotas	Unlimited.	Subject to throughput quotas by DynamoDB table and AWS Region.
Record delivery model	Pull model over HTTP using GetRecords and with enhanced fan-out , Kinesis Data Streams pushes the records over HTTP/2 by using SubscribeToShard .	Pull model over HTTP using GetRecords .
Ordering of records	The timestamp attribute on each stream record can be used to identify the actual order in which changes occurred in the DynamoDB table.	For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.
Duplicate records	Duplicate records might occasionally appear in the stream.	No duplicate records appear in the stream.

Properties	Kinesis Data Streams for DynamoDB	DynamoDB Streams
Stream processing options	Process stream records using AWS Lambda , Amazon Managed Service for Apache Flink , Kinesis data firehose , or AWS Glue streaming ETL .	Process stream records using AWS Lambda or DynamoDB Streams Kinesis adapter .
Durability level	Availability zones to provide automatic failover without interruption.	Availability zones to provide automatic failover without interruption.

You can enable both streaming models on the same DynamoDB table.

The following video talks more about the differences between the two options.

[DynamoDB Streams vs Kinesis Data Streams](#)

Using Kinesis Data Streams to capture changes to DynamoDB

You can use Amazon Kinesis Data Streams to capture changes to Amazon DynamoDB.

Kinesis Data Streams captures item-level modifications in any DynamoDB table and replicates them to a [Kinesis data stream](#). Your applications can access this stream and view item-level changes in near-real time. You can continuously capture and store terabytes of data per hour. You can take advantage of longer data retention time—and with enhanced fan-out capability, you can simultaneously reach two or more downstream applications. Other benefits include additional audit and security transparency.

Kinesis Data Streams also gives you access to [Amazon Data Firehose](#) and [Amazon Managed Service for Apache Flink](#). These services can help you build applications that power real-time dashboards, generate alerts, implement dynamic pricing and advertising, and implement sophisticated data analytics and machine learning algorithms.

Note

Using Kinesis data streams for DynamoDB is subject to both [Kinesis Data Streams pricing](#) for the data stream and [DynamoDB pricing](#) for the source table.

How Kinesis Data Streams works with DynamoDB

When a Kinesis data stream is enabled for a DynamoDB table, the table sends out a data record that captures any changes to that table's data. This data record includes:

- The specific time any item was recently created, updated, or deleted
- That item's primary key
- A snapshot of the record before the modification
- A snapshot of the record after the modification

These data records are captured and published in near-real time. After they are written to the Kinesis data stream, they can be read just like any other record. You can use the Kinesis Client Library, use AWS Lambda, call the Kinesis Data Streams API, and use other connected services. For more information, see [Reading Data from Amazon Kinesis Data Streams](#) in the Amazon Kinesis Data Streams Developer Guide.

These changes to data are also captured asynchronously. Kinesis has no performance impact on a table that it's streaming from. The stream records stored in your Kinesis data stream are also encrypted at rest. For more information, see [Data Protection in Amazon Kinesis Data Streams](#).

The Kinesis data stream records might appear in a different order than when the item changes occurred. The same item notifications might also appear more than once in the stream. You can check the `ApproximateCreationDateTime` attribute to identify the order that the item modifications occurred in, and to identify duplicate records.

When you enable a Kinesis data stream as a streaming destination of a DynamoDB table, you can configure the precision of `ApproximateCreationDateTime` values in either milliseconds or microseconds. By default, `ApproximateCreationDateTime` indicates the time of the change in milliseconds. Additionally, you can change this value on an active streaming destination. After such an update, stream records written to Kinesis will have `ApproximateCreationDateTime` values of the desired precision.

Binary values written to DynamoDB must be encoded in [base64-encoded format](#). However, when data records are written to a Kinesis data stream, these encoded binary values are encoded with base64-encoding a second time. When reading these records from a Kinesis data stream, in order to retrieve the raw binary values, applications must decode these values twice.

DynamoDB charges for using Kinesis Data Streams in change data capture units. 1 KB of change per single item counts as one change data capture unit. The KB of change in each item is calculated by the larger of the “before” and “after” images of the item written to the stream, using the same logic as [capacity unit consumption for write operations](#). Similar to how DynamoDB [on-demand](#) mode works, you don't need to provision capacity throughput for change data capture units.

Turning on a Kinesis data stream for your DynamoDB table

You can enable or disable streaming to Kinesis from your existing DynamoDB table by using the AWS Management Console, the AWS SDK, or the AWS Command Line Interface (AWS CLI).

- You can only stream data from DynamoDB to Kinesis Data Streams in the same AWS account and AWS Region as your table.
- You can only stream data from a DynamoDB table to one Kinesis data stream.

Making changes to a Kinesis Data Streams destination on your DynamoDB table

By default, all Kinesis data stream records include an `ApproximateCreationDateTime` attribute. This attribute represents a timestamp in milliseconds of the approximate time when each record was created. You can change the precision of these values by using the <https://console.aws.amazon.com/kinesis>, the SDK or the AWS CLI

Getting started with Kinesis Data Streams for Amazon DynamoDB

This section describes how to use Kinesis Data Streams for Amazon DynamoDB tables with the Amazon DynamoDB console, the AWS Command Line Interface (AWS CLI), and the API.

All of these examples use the Music DynamoDB table that was created as part of the [Getting started with DynamoDB](#) tutorial.

To learn more about how to build consumers and connect your Kinesis data stream to other AWS services, see [Reading data from Kinesis Data Streams](#) in the *Amazon Kinesis Data Streams developer guide*.

Note

When you're first using KDS shards, we recommend setting your shards to scale up and down with usage patterns. After you have accumulated more data on usage patterns, you can adjust the shards in your stream to match.

Console

1. Sign in to the AWS Management Console and open the Kinesis console at <https://console.aws.amazon.com/kinesis/>.
2. Choose **Create data stream** and follow the instructions to create a stream called **samplestream**.
3. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
4. In the navigation pane on the left side of the console, choose **Tables**.
5. Choose the **Music** table.
6. Choose the **Exports and streams** tab.

Music

Overview | Indexes | Monitor | Global tables | Backups | **Exports and streams** | Additional settings

Exports to S3 (0) [Info](#)

Showing all export jobs from the last 90 days.

Find exports

No exports

[Export to S3](#)

Amazon Kinesis data stream details

Edit record timestamp precision | Turn off

Status: **On** | Destination stream: **test**

Record timestamp precision: Microsecond

7. (Optional) Under **Amazon Kinesis data stream details**, you can change the record timestamp precision from microsecond (default) to millisecond.
8. Choose **samplestream** from the dropdown list.
9. Choose the **Turn On** button.

AWS CLI

1. Create a Kinesis Data Streams named `samplestream` by using the [create-stream command](#).

```
aws kinesis create-stream --stream-name samplestream --shard-count 3
```

See [Shard management considerations for Kinesis Data Streams](#) before setting the number of shards for the Kinesis data stream.

2. Check that the Kinesis stream is active and ready for use by using the [describe-stream command](#).

```
aws kinesis describe-stream --stream-name samplestream
```

3. Enable Kinesis streaming on the DynamoDB table by using the DynamoDB enable-kinesis-streaming-destination command. Replace the stream-arn value with the one that was returned by describe-stream in the previous step. Optionally, enable streaming with a more granular (microsecond) precision of timestamp values returned on each record.

Enable streaming with microsecond timestamp precision:

```
aws dynamodb enable-kinesis-streaming-destination \
--table-name Music \
--stream-arn arn:aws:kinesis:us-west-2:12345678901:stream/samplestream
--enable-kinesis-streaming-configuration
ApproximateCreationDateTimePrecision=MICROSECOND
```

Or enable streaming with default timestamp precision (millisecond):

```
aws dynamodb enable-kinesis-streaming-destination \
--table-name Music \
--stream-arn arn:aws:kinesis:us-west-2:12345678901:stream/samplestream
```

4. Check if Kinesis streaming is active on the table by using the DynamoDB describe-kinesis-streaming-destination command.

```
aws dynamodb describe-kinesis-streaming-destination --table-name Music
```

5. Write data to the DynamoDB table by using the put-item command, as described in the [DynamoDB Developer Guide](#).

```
aws dynamodb put-item \
--table-name Music \
--item \
'{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"}, "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}'

aws dynamodb put-item \
--table-name Music \
--item \
```

```
'{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"},  
"AlbumTitle": {"S": "Songs About Life"}, "Awards": {"N": "10"} }'
```

6. Use the Kinesis [get-records](#) CLI command to retrieve the Kinesis stream contents. Then use the following code snippet to deserialize the stream content.

```
/**  
 * Takes as input a Record fetched from Kinesis and does arbitrary processing as  
an example.  
 */  
public void processRecord(Record kinesisRecord) throws IOException {  
    ByteBuffer kdsRecordByteBuffer = kinesisRecord.getData();  
    JsonNode rootNode = OBJECT_MAPPER.readTree(kdsRecordByteBuffer.array());  
    JsonNode dynamoDBRecord = rootNode.get("dynamodb");  
    JsonNode oldItemImage = dynamoDBRecord.get("OldImage");  
    JsonNode newItemImage = dynamoDBRecord.get("NewImage");  
    Instant recordTimestamp = fetchTimestamp(dynamoDBRecord);  
  
    /**  
     * Say for example our record contains a String attribute named "stringName"  
and we want to fetch the value  
     * of this attribute from the new item image. The following code fetches  
this value.  
     */  
    JsonNode attributeNode = newItemImage.get("stringName");  
    JsonNode attributeValueNode = attributeNode.get("S"); // Using DynamoDB "S"  
type attribute  
    String attributeValue = attributeValueNode.textValue();  
    System.out.println(attributeValue);  
}  
  
private Instant fetchTimestamp(JsonNode dynamoDBRecord) {  
    JsonNode timestampJson = dynamoDBRecord.get("ApproximateCreationDateTime");  
    JsonNode timestampPrecisionJson =  
dynamoDBRecord.get("ApproximateCreationDateTimePrecision");  
    if (timestampPrecisionJson != null &&  
timestampPrecisionJson.equals("MICROSECOND")) {  
        return Instant.EPOCH.plus(timestampJson.longValue(), ChronoUnit.MICROS);  
    }  
    return Instant.ofEpochMilli(timestampJson.longValue());  
}
```

Java

1. Follow the instructions in the Kinesis Data Streams developer guide to [create](#) a Kinesis data stream named `samplestream` using Java.

See [Shard management considerations for Kinesis Data Streams](#) before setting the number of shards for the Kinesis data stream.

2. Use the following code snippet to enable Kinesis streaming on the DynamoDB table. Optionally, enable streaming with a more granular (microsecond) precision of timestamp values returned on each record.

Enable streaming with microsecond timestamp precision:

```
EnableKinesisStreamingConfiguration enableKdsConfig =
    EnableKinesisStreamingConfiguration.builder()

    .approximateCreationDateTimePrecision(ApproximateCreationDateTimePrecision.MICROSECOND)
    .build();

EnableKinesisStreamingDestinationRequest enableKdsRequest =
    EnableKinesisStreamingDestinationRequest.builder()
    .tableName(tableName)
    .streamArn(kdsArn)
    .enableKinesisStreamingConfiguration(enableKdsConfig)
    .build();

EnableKinesisStreamingDestinationResponse enableKdsResponse =
    ddbClient.enableKinesisStreamingDestination(enableKdsRequest);
```

Or enable streaming with default timestamp precision (millisecond):

```
EnableKinesisStreamingDestinationRequest enableKdsRequest =
    EnableKinesisStreamingDestinationRequest.builder()
    .tableName(tableName)
    .streamArn(kdsArn)
    .build();

EnableKinesisStreamingDestinationResponse enableKdsResponse =
    ddbClient.enableKinesisStreamingDestination(enableKdsRequest);
```

3. Follow the instructions in the *Kinesis Data Streams developer guide* to [read](#) from the created data stream.
4. Use the following code snippet to deserialize the stream content

```
/**  
 * Takes as input a Record fetched from Kinesis and does arbitrary processing as  
an example.  
 */  
public void processRecord(Record kinesisRecord) throws IOException {  
    ByteBuffer kdsRecordByteBuffer = kinesisRecord.getData();  
    JsonNode rootNode = OBJECT_MAPPER.readTree(kdsRecordByteBuffer.array());  
    JsonNode dynamoDBRecord = rootNode.get("dynamodb");  
    JsonNode oldItemImage = dynamoDBRecord.get("OldImage");  
    JsonNode newItemImage = dynamoDBRecord.get("NewImage");  
    Instant recordTimestamp = fetchTimestamp(dynamoDBRecord);  
  
    /**  
     * Say for example our record contains a String attribute named "stringName"  
and we wanted to fetch the value  
     * of this attribute from the new item image, the below code would fetch  
this.  
     */  
    JsonNode attributeNode = newItemImage.get("stringName");  
    JsonNode attributeValueNode = attributeNode.get("S"); // Using DynamoDB "S"  
type attribute  
    String attributeValue = attributeValueNode.textValue();  
    System.out.println(attributeValue);  
}  
  
private Instant fetchTimestamp(JsonNode dynamoDBRecord) {  
    JsonNode timestampJson = dynamoDBRecord.get("ApproximateCreationDateTime");  
    JsonNode timestampPrecisionJson =  
dynamoDBRecord.get("ApproximateCreationDateTimePrecision");  
    if (timestampPrecisionJson != null &&  
timestampPrecisionJson.equals("MICROSECOND")) {  
        return Instant.EPOCH.plus(timestampJson.longValue(), ChronoUnit.MICROS);  
    }  
    return Instant.ofEpochMilli(timestampJson.longValue());  
}
```

Making changes to an active Amazon Kinesis data stream

This section describes how to make changes to an active Kinesis Data Streams for DynamoDB setup by using the console, AWS CLI and the API.

AWS Management Console

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>
2. Go to your table.
3. Choose **Exports and Streams**.

AWS CLI

1. Call `describe-kinesis-streaming-destination` to confirm that the stream is ACTIVE.
2. Call `UpdateKinesisStreamingDestination`, such as in this example:

```
aws dynamodb update-kinesis-streaming-destination --table-name
    enable_test_table --stream-arn arn:aws:kinesis:us-east-1:12345678901:stream/
    enable_test_stream --update-kinesis-streaming-configuration
        ApproximateCreationDateTimePrecision=MICROSECOND
```

3. Call `describe-kinesis-streaming-destination` to confirm that the stream is UPDATING.
4. Call `describe-kinesis-streaming-destination` periodically until the streaming status is ACTIVE again. It typically takes up to 5 minutes for the timestamp precision updates to take effect. Once this status updates, that indicates that the update is complete and the new precision value will be applied on future records.
5. Write to the table using `putItem`.
6. Use the Kinesis `get-records` command to get the stream contents.
7. Confirm that the `ApproximateCreationDateTime` of the writes have the desired precision.

Java API

1. Provide a code snippet that constructs an `UpdateKinesisStreamingDestination` request and an `UpdateKinesisStreamingDestination` response.
2. Provide a code snippet that constructs a `DescribeKinesisStreamingDestination` request and a `DescribeKinesisStreamingDestination` response.

3. Call `describe-kinesis-streaming-destination` periodically until the streaming status is ACTIVE again, indicating that the update is complete and the new precision value will be applied on future records.
4. Perform writes to the table.
5. Read from the stream and deserialize the stream content.
6. Confirm that the `ApproximateCreationDateTime` of the writes have the desired precision.

Configuring shards and monitoring change data capture with Kinesis Data Streams in DynamoDB

Shard management considerations for Kinesis Data Streams

A Kinesis data stream counts its throughput in [shards](#). In Amazon Kinesis Data streams, you can choose between an **on-demand** mode and a **provisioned** mode for your data streams.

We recommend using on-demand mode for your Kinesis Data Stream if your DynamoDB write workload is highly variable and unpredictable. With on-demand mode, there is no capacity planning required as Kinesis Data Streams automatically manages the shards in order to provide the necessary throughput.

For predictable workloads, you can use provisioned mode for your Kinesis Data Stream. With provisioned mode, you must specify the number of shards for the data stream to accommodate the change data capture records from DynamoDB. To determine the number of shards that the Kinesis data stream will need to support your DynamoDB table, you need the following input values:

- The average size of your DynamoDB table's record in bytes (`average_record_size_in_bytes`).
- The maximum number of write operations that your DynamoDB table will perform per second. This includes create, delete, and update operations performed by your applications, as well as automatically generated operations like Time to Live generated delete operations(`write_throughput`).
- The percentage of update and overwrite operations that you perform on your table, as compared to create or delete operations (`percentage_of_updates`). Keep in mind that update and overwrite operations replicate both the old and new images of the modified item to the stream. This generates twice the DynamoDB item size.

You can calculate the number of shards (`number_of_shards`) that your Kinesis data stream needs by using the input values in the following formula:

```
number_of_shards = ceiling( max( ((write_throughput * (4+percentage_of_updates) * average_record_size_in_bytes) / 1024 / 1024), (write_throughput/1000)), 1)
```

For example, you might have a maximum throughput of 1040 write operations per second (`write_throughput`) with an average record size of 800 bytes (`average_record_size_in_bytes`). If 25 percent of those write operations are update operations (`percentage_of_updates`), then you will need two shards (`number_of_shards`) to accommodate your DynamoDB streaming throughput:

```
ceiling( max( ((1040 * (4+25/100) * 800)/ 1024 / 1024), (1040/1000)), 1).
```

Consider the following before using the formula to calculate the number of shards required with provisioned mode for Kinesis data streams:

- This formula helps estimate the number of shards that will be required to accommodate your DynamoDB change data records. It doesn't represent the total number of shards needed in your Kinesis data stream, such as the number of shards required to support additional Kinesis data stream consumers.
- You may still experience read and write throughput exceptions in the provisioned mode if you don't configure your data stream to handle your peak throughput. In this case, you must manually scale your data stream to accommodate your data traffic.
- This formula takes into consideration the additional bloat generated by DynamoDB before streaming the change logs data records to Kinesis Data Stream.

To learn more about capacity modes on Kinesis Data Stream see [Choosing the Data Stream Capacity Mode](#). To learn more about pricing difference between different capacity modes, see [Amazon Kinesis Data Streams pricing](#).

Monitoring change data capture with Kinesis Data Streams

DynamoDB provides several Amazon CloudWatch metrics to help you monitor the replication of change data capture to Kinesis. For a full list of CloudWatch metrics, see [DynamoDB Metrics and dimensions](#).

To determine whether your stream has sufficient capacity, we recommend that you monitor the following items both during stream enabling and in production:

- **ThrottledPutRecordCount:** The number of records that were throttled by your Kinesis data stream because of insufficient Kinesis data stream capacity. You might experience some throttling during exceptional usage peaks, but the ThrottledPutRecordCount should remain as low as possible. DynamoDB retries sending throttled records to the Kinesis data stream, but this might result in higher replication latency.

If you experience excessive and regular throttling, you might need to increase the number of Kinesis stream shards proportionally to the observed write throughput of your table. To learn more about determining the size of a Kinesis data stream, see [Determining the Initial Size of a Kinesis Data Stream](#).

- **AgeOfOldestUnreplicatedRecord:** The elapsed time since the oldest item-level change yet to replicate to the Kinesis data stream appeared in the DynamoDB table. Under normal operation, AgeOfOldestUnreplicatedRecord should be in the order of milliseconds. This number grows based on unsuccessful replication attempts when these are caused by customer-controlled configuration choices.

If AgeOfOldestUnreplicatedRecord metric exceeds 168 hours, replication of item-level changes from the DynamoDB table to Kinesis data stream will be automatically disabled.

Customer-controlled configuration examples that leads to unsuccessful replication attempts are an under-provisioned Kinesis data stream capacity that leads to excessive throttling, or a manual update to your Kinesis data stream's access policies that prevents DynamoDB from adding data to your data stream. To keep this metric as low as possible, you might need to ensure the right provisioning of your Kinesis data stream capacity, and make sure that DynamoDB's permissions are unchanged.

- **FailedToReplicateRecordCount:** The number of records that DynamoDB failed to replicate to your Kinesis data stream. Certain items larger than 34 KB might expand in size to change data records that are larger than the 1 MB item size limit of Kinesis Data Streams. This size expansion occurs when these larger than 34 KB items include a large number of Boolean or empty attribute values. Boolean and empty attribute values are stored as 1 byte in DynamoDB, but expand up to 5 bytes when they're serialized using standard JSON for Kinesis Data Streams replication. DynamoDB can't replicate such change records to your Kinesis data stream. DynamoDB skips these change data records, and automatically continues replicating subsequent records.

You can create Amazon CloudWatch alarms that send an Amazon Simple Notification Service (Amazon SNS) message for notification when any of the preceding metrics exceed a specific threshold. For more information, see [Creating CloudWatch alarms to monitor DynamoDB](#).

Using IAM policies for Amazon Kinesis Data Streams and Amazon DynamoDB

The first time that you enable Amazon Kinesis Data Streams for Amazon DynamoDB, DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role, AWSServiceRoleForDynamoDBKinesisDataStreamsReplication, allows DynamoDB to manage the replication of item-level changes to Kinesis Data Streams on your behalf. Don't delete this service-linked role.

For more information about service-linked roles, see [Using service-linked roles in the IAM User Guide](#).

To enable Amazon Kinesis Data Streams for Amazon DynamoDB, you must have the following permissions on the table:

- dynamodb:EnableKinesisStreamingDestination
- kinesis>ListStreams
- kinesis:PutRecords
- kinesis:DescribeStream

To describe Amazon Kinesis Data Streams for Amazon DynamoDB for a given DynamoDB table, you must have the following permissions on the table.

- dynamodb:DescribeKinesisStreamingDestination
- kinesis:DescribeStreamSummary
- kinesis:DescribeStream

To disable Amazon Kinesis Data Streams for Amazon DynamoDB, you must have the following permissions on the table.

- dynamodb:DisableKinesisStreamingDestination

To update Amazon Kinesis Data Streams for Amazon DynamoDB, you must have the following permissions on the table.

- `dynamodb:UpdateKinesisStreamingDestination`

The following examples show how to use IAM policies to grant permissions for Amazon Kinesis Data Streams for Amazon DynamoDB.

Example: Enable Amazon Kinesis Data Streams for Amazon DynamoDB

The following IAM policy grants permissions to enable Amazon Kinesis Data Streams for Amazon DynamoDB for the Music table. It does not grant permissions to disable, update or describe Kinesis Data Streams for DynamoDB for the Music table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iam:CreateServiceLinkedRole",  
            "Resource": "arn:aws:iam::*:role/aws-service-role/  
kinesisreplication.dynamodb.amazonaws.com/  
AWSServiceRoleForDynamoDBKinesisDataStreamsReplication",  
            "Condition": {"StringLike": {"iam:AWSServiceName":  
"kinesisreplication.dynamodb.amazonaws.com"}}  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
  
                "dynamodb:EnableKinesisStreamingDestination"  
  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:12345678901:table/Music"  
        }  
    ]  
}
```

Example: Update Amazon Kinesis Data Streams for Amazon DynamoDB

The following IAM policy grants permissions to update Amazon Kinesis Data Streams for Amazon DynamoDB for the Music table. It does not grant permissions to enable, disable or describe Amazon Kinesis Data Streams for Amazon DynamoDB for the Music table.

```
{
```

```
"Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
        "Action": [
          "dynamodb:UpdateKinesisStreamingDestination"
        ],
      "Resource": "arn:aws:dynamodb:us-west-2:12345678901:table/Music"
    }
  ]
}
```

Example: Disable Amazon Kinesis Data Streams for Amazon DynamoDB

The following IAM policy grants permissions to disable Amazon Kinesis Data Streams for Amazon DynamoDB for the Music table. It does not grant permissions to enable, update or describe Amazon Kinesis Data Streams for Amazon DynamoDB for the Music table.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
        "Action": [
          "dynamodb:DisableKinesisStreamingDestination"
        ],
      "Resource": "arn:aws:dynamodb:us-west-2:12345678901:table/Music"
    }
  ]
}
```

Example: Selectively apply permissions for Amazon Kinesis Data Streams for Amazon DynamoDB based on resource

The following IAM policy grants permissions to enable and describe Amazon Kinesis Data Streams for Amazon DynamoDB for the Music table, and denies permissions to disable Amazon Kinesis Data Streams for Amazon DynamoDB for the Orders table.

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
{  
    "Effect": "Allow",  
    "Action": [  
        "dynamodb:EnableKinesisStreamingDestination",  
        "dynamodb:DescribeKinesisStreamingDestination"  
    ],  
    "Resource": "arn:aws:dynamodb:us-west-2:12345678901:table/Music"  
},  
{  
    "Effect": "Deny",  
    "Action": [  
        "dynamodb:DisableKinesisStreamingDestination"  
    ],  
    "Resource": "arn:aws:dynamodb:us-west-2:12345678901:table/Orders"  
}  
]  
}
```

Using service-linked roles for Kinesis Data Streams for DynamoDB

Amazon Kinesis Data Streams for Amazon DynamoDB uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to Kinesis Data Streams for DynamoDB. Service-linked roles are predefined by Kinesis Data Streams for DynamoDB and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up Kinesis Data Streams for DynamoDB easier because you don't have to manually add the necessary permissions. Kinesis Data Streams for DynamoDB defines the permissions of its service-linked roles, and unless defined otherwise, only Kinesis Data Streams for DynamoDB can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for Kinesis Data Streams for DynamoDB

Kinesis Data Streams for DynamoDB uses the service-linked role named **AWSServiceRoleForDynamoDBKinesisDataStreamsReplication**. The purpose of the service-linked

role is to allow Amazon DynamoDB to manage the replication of item-level changes to Kinesis Data Streams, on your behalf.

The `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` service-linked role trusts the following services to assume the role:

- `kinesisreplication.dynamodb.amazonaws.com`

The role permissions policy allows Kinesis Data Streams for DynamoDB to complete the following actions on the specified resources:

- Action: `Put records` and `describe` on Kinesis stream
- Action: `Generate data keys` on AWS KMS in order to put data on Kinesis streams that are encrypted using User-Generated AWS KMS keys.

For the exact contents of the policy document, see [DynamoDBKinesisReplicationServiceRolePolicy](#).

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

Creating a service-linked role for Kinesis Data Streams for DynamoDB

You don't need to manually create a service-linked role. When you enable Kinesis Data Streams for DynamoDB in the AWS Management Console, the AWS CLI, or the AWS API, Kinesis Data Streams for DynamoDB creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you enable Kinesis Data Streams for DynamoDB, Kinesis Data Streams for DynamoDB creates the service-linked role for you again.

Editing a service-linked role for Kinesis Data Streams for DynamoDB

Kinesis Data Streams for DynamoDB does not allow you to edit the `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

Deleting a service-linked role for Kinesis Data Streams for DynamoDB

You can also use the IAM console, the AWS CLI or the AWS API to manually delete the service-linked role. To do this, you must first manually clean up the resources for your service-linked role and then you can manually delete it.

Note

If the Kinesis Data Streams for DynamoDB service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

Change data capture for DynamoDB Streams

DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near-real time.

Encryption at rest encrypts the data in DynamoDB streams. For more information, see [DynamoDB encryption at rest](#).

A *DynamoDB stream* is an ordered flow of information about changes to items in a DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. A *stream record* contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams helps ensure the following:

- Each stream record appears exactly once in the stream.
- For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

DynamoDB Streams writes stream records in near-real time so that you can build applications that consume these streams and take action based on the contents.

Topics

- [Endpoints for DynamoDB Streams](#)
- [Enabling a stream](#)
- [Reading and processing a stream](#)
- [DynamoDB Streams and Time to Live](#)
- [Using the DynamoDB Streams Kinesis adapter to process stream records](#)
- [DynamoDB Streams low-level API: Java example](#)
- [DynamoDB Streams and AWS Lambda triggers](#)

Endpoints for DynamoDB Streams

AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application must access a DynamoDB endpoint. To read and process DynamoDB Streams records, your application must access a DynamoDB Streams endpoint in the same Region.

The naming convention for DynamoDB Streams endpoints is `streams.dynamodb.<region>.amazonaws.com`. For example, if you use the endpoint `dynamodb.us-west-2.amazonaws.com` to access DynamoDB, you would use the endpoint `streams.dynamodb.us-west-2.amazonaws.com` to access DynamoDB Streams.

Note

For a complete list of DynamoDB and DynamoDB Streams Regions and endpoints, see [Regions and endpoints](#) in the *AWS General Reference*.

The AWS SDKs provide separate clients for DynamoDB and DynamoDB Streams. Depending on your requirements, your application can access a DynamoDB endpoint, a DynamoDB Streams

endpoint, or both at the same time. To connect to both endpoints, your application must instantiate two clients—one for DynamoDB and one for DynamoDB Streams.

Enabling a stream

You can enable a stream on a new table when you create it using the AWS CLI or one of the AWS SDKs. You can also enable or disable a stream on an existing table, or change the settings of a stream. DynamoDB Streams operates asynchronously, so there is no performance impact on a table if you enable a stream.

The easiest way to manage DynamoDB Streams is by using the AWS Management Console.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. On the DynamoDB console dashboard, choose **Tables** and select an existing table.
3. Choose the **Exports and streams** tab.
4. In the **DynamoDB stream details** section, choose **Turn on**.
5. On the **Turn on DynamoDB stream** page, choose the information that will be written to the stream whenever the data in the table is modified:
 - **Key attributes only** — Only the key attributes of the modified item.
 - **New image** — The entire item, as it appears after it was modified.
 - **Old image** — The entire item, as it appeared before it was modified.
 - **New and old images** — Both the new and the old images of the item.

When the settings are as you want them, choose **Turn on stream**.

6. (Optional) To disable an existing stream, choose **Turn off** under **DynamoDB stream details**.

You can also use the `CreateTable` or `UpdateTable` API operations to enable or modify a stream. The `StreamSpecification` parameter determines how the stream is configured:

- `StreamEnabled` — Specifies whether a stream is enabled (`true`) or disabled (`false`) for the table.
- `StreamViewType` — Specifies the information that will be written to the stream whenever data in the table is modified:
 - `KEYS_ONLY` — Only the key attributes of the modified item.

- NEW_IMAGE — The entire item, as it appears after it was modified.
- OLD_IMAGE — The entire item, as it appeared before it was modified.
- NEW_AND_OLD_IMAGES — Both the new and the old images of the item.

You can enable or disable a stream at any time. However, you receive a `ResourceInUseException` if you try to enable a stream on a table that already has a stream. You receive a `ValidationException` if you try to disable a stream on a table that doesn't have a stream.

When you set `StreamEnabled` to `true`, DynamoDB creates a new stream with a unique stream descriptor assigned to it. If you disable and then re-enable a stream on the table, a new stream is created with a different stream descriptor.

Every stream is uniquely identified by an Amazon Resource Name (ARN). The following is an example ARN for a stream on a DynamoDB table named `TestTable`.

```
arn:aws:dynamodb:us-west-2:111122223333:table/TestTable/stream/2015-05-11T21:21:33.291
```

To determine the latest stream descriptor for a table, issue a DynamoDB `DescribeTable` request and look for the `LatestStreamArn` element in the response.

 **Note**

It is not possible to edit a `StreamViewType` once a stream has been setup. If you need to make changes to a stream after it has been setup, you must disable the current stream and create a new one.

Reading and processing a stream

To read and process a stream, your application must connect to a DynamoDB Streams endpoint and issue API requests.

A stream consists of *stream records*. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, reflecting the order in which the record was published to the stream.

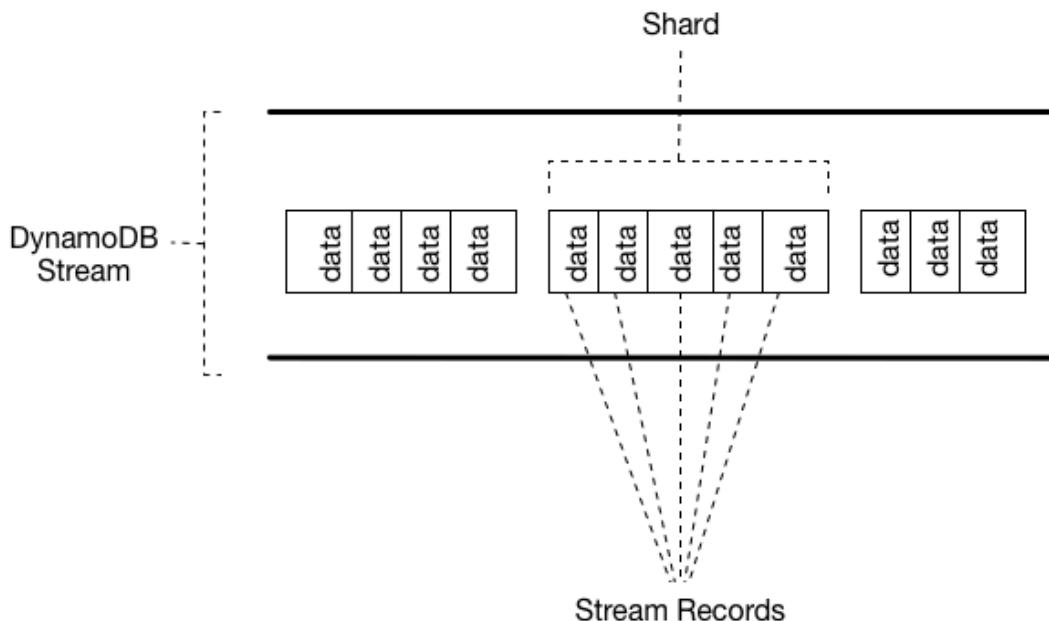
Stream records are organized into groups, or *shards*. Each shard acts as a container for multiple stream records, and contains information required for accessing and iterating through these records. The stream records within a shard are removed automatically after 24 hours.

Shards are ephemeral: They are created and deleted automatically, as needed. Any shard can also split into multiple new shards; this also occurs automatically. (It's also possible for a parent shard to have just one child shard.) A shard might split in response to high levels of write activity on its parent table, so that applications can process records from multiple shards in parallel.

If you disable a stream, any shards that are open will be closed. The data in the stream will continue to be readable for 24 hours.

Because shards have a lineage (parent and children), an application must always process a parent shard before it processes a child shard. This helps ensure that the stream records are also processed in the correct order. (If you use the DynamoDB Streams Kinesis Adapter, this is handled for you. Your application processes the shards and stream records in the correct order. It automatically handles new or expired shards, in addition to shards that split while the application is running. For more information, see [Using the DynamoDB Streams Kinesis adapter to process stream records.](#))

The following diagram shows the relationship between a stream, shards in the stream, and stream records in the shards.



Note

If you perform a `PutItem` or `UpdateItem` operation that does not change any data in an item, DynamoDB Streams does *not* write a stream record for that operation.

To access a stream and process the stream records within, you must do the following:

- Determine the unique ARN of the stream that you want to access.
- Determine which shards in the stream contain the stream records that you are interested in.
- Access the shards and retrieve the stream records that you want.

Note

No more than two processes at most should be reading from the same stream's shard at the same time. Having more than two readers per shard can result in throttling.

The DynamoDB Streams API provides the following actions for use by application programs:

- [ListStreams](#) — Returns a list of stream descriptors for the current account and endpoint. You can optionally request just the stream descriptors for a particular table name.
- [DescribeStream](#) — Returns detailed information about a given stream. The output includes a list of shards associated with the stream, including the shard IDs.
- [GetShardIterator](#) — Returns a *shard iterator*, which describes a location within a shard. You can request that the iterator provide access to the oldest point, the newest point, or a particular point in the stream.
- [GetRecords](#) — Returns the stream records from within a given shard. You must provide the shard iterator returned from a `GetShardIterator` request.

For complete descriptions of these API operations, including example requests and responses, see the [Amazon DynamoDB Streams API Reference](#).

Data retention limit for DynamoDB Streams

All data in DynamoDB Streams is subject to a 24-hour lifetime. You can retrieve and analyze the last 24 hours of activity for any given table. However, data that is older than 24 hours is susceptible to trimming (removal) at any moment.

If you disable a stream on a table, the data in the stream continues to be readable for 24 hours. After this time, the data expires and the stream records are automatically deleted. There is no mechanism for manually deleting an existing stream. You must wait until the retention limit expires (24 hours), and all the stream records will be deleted.

DynamoDB Streams and Time to Live

You can back up, or otherwise process, items that are deleted by [Time to Live](#) (TTL) by enabling Amazon DynamoDB Streams on the table and processing the streams records of the expired items. For more information, see [Reading and processing a stream](#).

The streams record contains a user identity field `Records[<index>].userIdentity`.

Items that are deleted by the Time to Live process after expiration have the following fields:

- `Records[<index>].userIdentity.type`
"Service"
- `Records[<index>].userIdentity.principalId`
"dynamodb.amazonaws.com"

Note

When you use TTL in a global table, the region the TTL was performed in will have the `userIdentity` field set. This field won't be set in other regions when the delete is replicated.

The following JSON shows the relevant portion of a single streams record.

```
"Records": [  
  {
```

```
...  
  "userIdentity": {  
    "type": "Service",  
    "principalId": "dynamodb.amazonaws.com"  
  }  
...  
}  
]
```

Using DynamoDB Streams and Lambda to archive TTL deleted items

Combining [DynamoDB Time to Live \(TTL\)](#), [DynamoDB Streams](#), and [AWS Lambda](#) can help simplify archiving data, reduce DynamoDB storage costs, and reduce code complexity. Using Lambda as the stream consumer provides many advantages, most notably the cost reduction compared to other consumers such as Kinesis Client Library (KCL). You aren't charged for GetRecords API calls on your DynamoDB stream when using Lambda to consume events, and Lambda can provide event filtering by identifying JSON patterns in a stream event. With event-pattern content filtering, you can define up to five different filters to control which events are sent to Lambda for processing. This helps reduce invocations of your Lambda functions, simplifies code, and reduces overall cost.

While DynamoDB Streams contains all data modifications, such as Create, Modify, and Remove actions, this can result in unwanted invocations of your archive Lambda function. For example, say you have a table with 2 million data modifications per hour flowing into the stream, but less than 5 percent of these are item deletes that will expire through the TTL process and need to be archived. With [Lambda event source filters](#), the Lambda function will only invoke 100,000 times per hour. The result with event filtering is that you're charged only for the needed invocations instead of the 2 million invocations you would have without event filtering.

Event filtering is applied to the [Lambda event source mapping](#), which is a resource that reads from a chosen event—the DynamoDB stream—and invokes a Lambda function. In the following diagram, you can see how a Time to Live deleted item is consumed by a Lambda function using streams and event filters.



DynamoDB Time to Live event filter pattern

Adding the following JSON to your event source mapping [filter criteria](#) allows invocation of your Lambda function only for TTL deleted items:

```
{  
    "Filters": [  
        {  
            "Pattern": { "userIdentity": { "type": ["Service"], "principalId":  
                ["dynamodb.amazonaws.com"] } }  
        }  
    ]  
}
```

Create an AWS Lambda event source mapping

Use the following code snippets to create a filtered event source mapping which you can connect to a table's DynamoDB stream. Each code block includes the event filter pattern.

AWS CLI

```
aws lambda create-event-source-mapping \  
--event-source-arn 'arn:aws:dynamodb:eu-west-1:012345678910:table/test/  
stream/2021-12-10T00:00:00.000' \  
--batch-size 10 \  
--enabled \  
--function-name test_func \  
--starting-position LATEST \  
--filter-criteria '{"Filters": [{"Pattern": {"userIdentity": {"type": ["Service"], "principalId": ["dynamodb.amazonaws.com"]}}}]}'
```

Java

```
LambdaClient client = LambdaClient.builder()  
    .region(Region.EU_WEST_1)  
    .build();  
  
Filter userIdentity = Filter.builder()  
    .pattern("{\"userIdentity\":{\"type\": [\"Service\"], \"principalId\":  
        [\"dynamodb.amazonaws.com\"]}}")  
    .build();
```

```
FilterCriteria filterCriteria = FilterCriteria.builder()
    .filters(userIdentity)
    .build();

CreateEventSourceMappingRequest mappingRequest =
CreateEventSourceMappingRequest.builder()
    .eventSourceArn("arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000")
    .batchSize(10)
    .enabled(Boolean.TRUE)
    .functionName("test_func")
    .startingPosition("LATEST")
    .filterCriteria(filterCriteria)
    .build();

try{
    CreateEventSourceMappingResponse eventSourceMappingResponse =
client.createEventSourceMapping(mappingRequest);
    System.out.println("The mapping ARN is
"+eventSourceMappingResponse.eventSourceArn());

}catch (ServiceException e){
    System.out.println(e.getMessage());
}
```

Node

```
const client = new LambdaClient({ region: "eu-west-1" });

const input = {
  EventSourceArn: "arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000",
  BatchSize: 10,
  Enabled: true,
  FunctionName: "test_func",
  StartingPosition: "LATEST",
  FilterCriteria: { "Filters": [{ "Pattern": "{\"userIdentity\":{\"type\":\"[\\"Service\\\"],\\\"principalId\\\":["dynamodb.amazonaws.com\"]}}"} ] }
}

const command = new CreateEventSourceMappingCommand(input);
```

```
try {
    const results = await client.send(command);
    console.log(results);
} catch (err) {
    console.error(err);
}
```

Python

```
session = boto3.session.Session(region_name = 'eu-west-1')
client = session.client('lambda')

try:
    response = client.create_event_source_mapping(
        EventSourceArn='arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000',
        BatchSize=10,
        Enabled=True,
        FunctionName='test_func',
        StartingPosition='LATEST',
        FilterCriteria={
            'Filters': [
                {
                    'Pattern': "{\"userIdentity\":{\"type\":[\"Service\"],"
                    \"principalId\":[\"dynamodb.amazonaws.com\"]}}"
                },
            ]
        }
    )
    print(response)
except Exception as e:
    print(e)
```

JSON

```
{
    "userIdentity": {
        "type": ["Service"],
        "principalId": ["dynamodb.amazonaws.com"]
    }
}
```

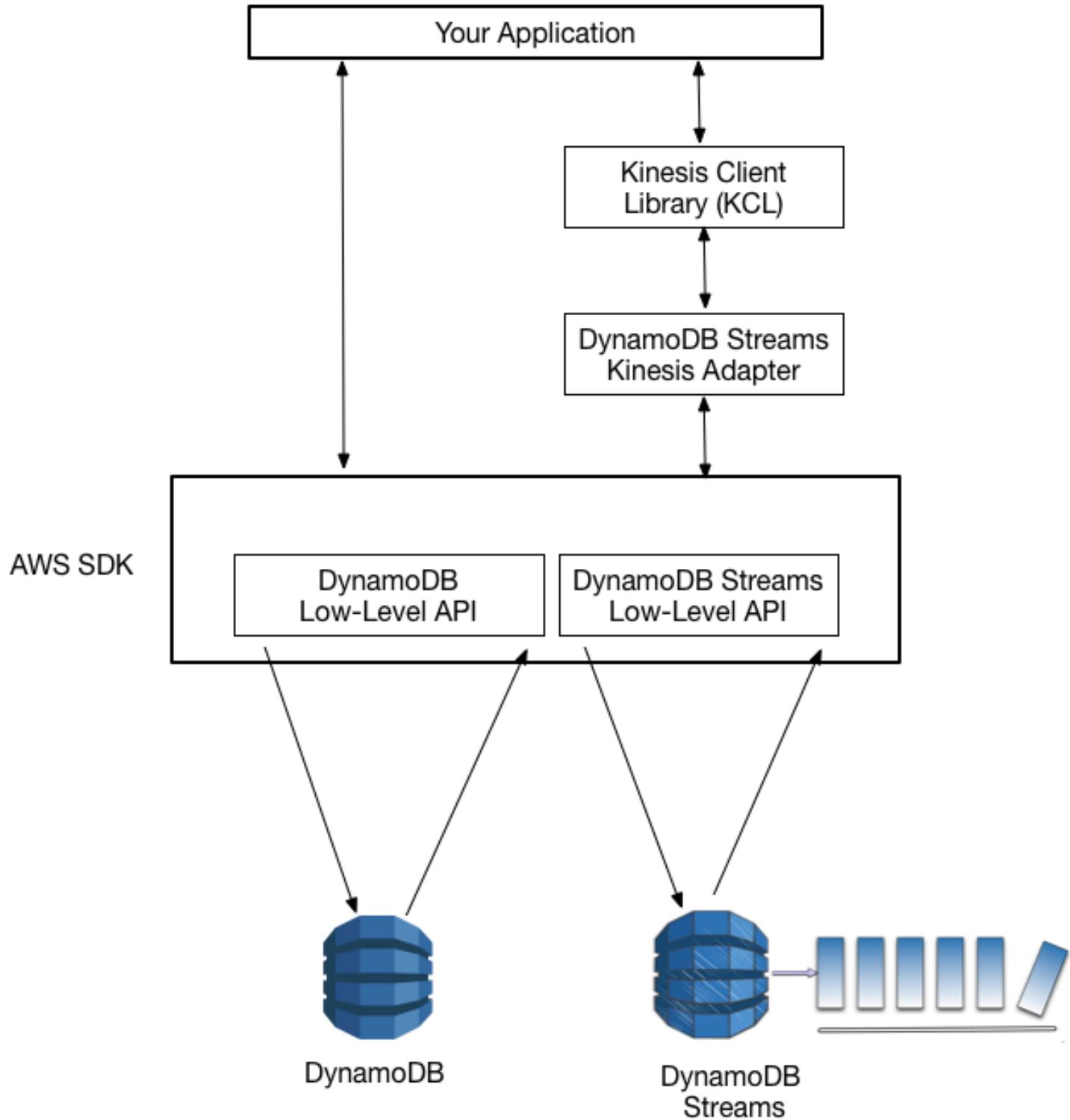
Using the DynamoDB Streams Kinesis adapter to process stream records

Using the Amazon Kinesis Adapter is the recommended way to consume streams from Amazon DynamoDB. The DynamoDB Streams API is intentionally similar to that of Kinesis Data Streams, a service for real-time processing of streaming data at massive scale. In both services, data streams are composed of shards, which are containers for stream records. Both services' APIs contain `ListStreams`, `DescribeStream`, `GetShards`, and `GetShardIterator` operations. (Although these DynamoDB Streams actions are similar to their counterparts in Kinesis Data Streams, they are not 100 percent identical.)

You can write applications for Kinesis Data Streams using the Kinesis Client Library (KCL). The KCL simplifies coding by providing useful abstractions above the low-level Kinesis Data Streams API. For more information about the KCL, see the [Developing consumers using the Kinesis client library](#) in the *Amazon Kinesis Data Streams Developer Guide*.

As a DynamoDB Streams user, you can use the design patterns found within the KCL to process DynamoDB Streams shards and stream records. To do this, you use the DynamoDB Streams Kinesis Adapter. The Kinesis Adapter implements the Kinesis Data Streams interface so that the KCL can be used for consuming and processing records from DynamoDB Streams. For instructions on how to set up and install the DynamoDB Streams Kinesis Adapter, see the [GitHub repository](#) .

The following diagram shows how these libraries interact with one another.



With the DynamoDB Streams Kinesis Adapter in place, you can begin developing against the KCL interface, with the API calls seamlessly directed at the DynamoDB Streams endpoint.

When your application starts, it calls the KCL to instantiate a worker. You must provide the worker with configuration information for the application, such as the stream descriptor and AWS credentials, and the name of a record processor class that you provide. As it runs the code in the record processor, the worker performs the following tasks:

- Connects to the stream
- Enumerates the shards within the stream
- Coordinates shard associations with other workers (if any)
- Instantiates a record processor for every shard it manages
- Pulls records from the stream
- Pushes the records to the corresponding record processor
- Checkpoints processed records
- Balances shard-worker associations when the worker instance count changes
- Balances shard-worker associations when shards are split

 **Note**

For a description of the KCL concepts listed here, see [Developing consumers using the Kinesis client library](#) in the *Amazon Kinesis Data Streams Developer Guide*.

For more information on using streams with AWS Lambda see [DynamoDB Streams and AWS Lambda triggers](#)

Walkthrough: DynamoDB Streams Kinesis adapter

This section is a walkthrough of a Java application that uses the Amazon Kinesis Client Library and the Amazon DynamoDB Streams Kinesis Adapter. The application shows an example of data replication, in which write activity from one table is applied to a second table, with both tables' contents staying in sync. For the source code, see [Complete program: DynamoDB Streams Kinesis adapter](#).

The program does the following:

1. Creates two DynamoDB tables named KCL-Demo-src and KCL-Demo-dst. Each of these tables has a stream enabled on it.

2. Generates update activity in the source table by adding, updating, and deleting items. This causes data to be written to the table's stream.
3. Reads the records from the stream, reconstructs them as DynamoDB requests, and applies the requests to the destination table.
4. Scans the source and destination tables to ensure that their contents are identical.
5. Cleans up by deleting the tables.

These steps are described in the following sections, and the complete application is shown at the end of the walkthrough.

Topics

- [Step 1: Create DynamoDB tables](#)
- [Step 2: Generate update activity in source table](#)
- [Step 3: Process the stream](#)
- [Step 4: Ensure that both tables have identical contents](#)
- [Step 5: Clean up](#)
- [Complete program: DynamoDB Streams Kinesis adapter](#)

Step 1: Create DynamoDB tables

The first step is to create two DynamoDB tables—a source table and a destination table. The `StreamViewType` on the source table's stream is `NEW_IMAGE`. This means that whenever an item is modified in this table, the item's "after" image is written to the stream. In this way, the stream keeps track of all write activity on the table.

The following example shows the code that is used for creating both tables.

```
java.util.List<AttributeDefinition> attributeDefinitions = new
    ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

java.util.List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new
    KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition
```

```
// key

ProvisionedThroughput provisionedThroughput = new
ProvisionedThroughput().withReadCapacityUnits(2L)
.withWriteCapacityUnits(2L);

StreamSpecification streamSpecification = new StreamSpecification();
streamSpecification.setStreamEnabled(true);
streamSpecification.setStreamViewType(StreamViewType.NEW_IMAGE);
CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
.withAttributeDefinitions(attributeDefinitions).withKeySchema(keySchema)

.withProvisionedThroughput(provisionedThroughput).withStreamSpecification(streamSpecification)
```

Step 2: Generate update activity in source table

The next step is to generate some write activity on the source table. While this activity is taking place, the source table's stream is also updated in near-real time.

The application defines a helper class with methods that call the PutItem, UpdateItem, and DeleteItem API operations for writing the data. The following code example shows how these methods are used.

```
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "101", "test1");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "101", "test2");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "101");
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "102", "demo3");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "102", "demo4");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "102");
```

Step 3: Process the stream

Now the program begins processing the stream. The DynamoDB Streams Kinesis Adapter acts as a transparent layer between the KCL and the DynamoDB Streams endpoint, so that the code can fully use KCL rather than having to make low-level DynamoDB Streams calls. The program performs the following tasks:

- It defines a record processor class, `StreamsRecordProcessor`, with methods that comply with the KCL interface definition: `initialize`, `processRecords`, and `shutdown`. The `processRecords` method contains the logic required for reading from the source table's stream and writing to the destination table.
- It defines a class factory for the record processor class (`StreamsRecordProcessorFactory`). This is required for Java programs that use the KCL.
- It instantiates a new KCL Worker, which is associated with the class factory.
- It shuts down the Worker when record processing is complete.

To learn more about the KCL interface definition, see [Developing consumers using the Kinesis client library](#) in the *Amazon Kinesis Data Streams Developer Guide*.

The following code example shows the main loop in `StreamsRecordProcessor`. The case statement determines what action to perform, based on the `OperationType` that appears in the stream record.

```
for (Record record : records) {  
    String data = new String(record.getData().array(), Charset.forName("UTF-8"));  
    System.out.println(data);  
    if (record instanceof RecordAdapter) {  
        com.amazonaws.services.dynamodbv2.model.Record streamRecord =  
        ((RecordAdapter) record)  
            .getInternalObject();  
  
        switch (streamRecord.getEventName()) {  
            case "INSERT":  
            case "MODIFY":  
                StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName,  
                    streamRecord.getDynamodb().getNewImage());  
                break;  
            case "REMOVE":  
                StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName,  
                    streamRecord.getDynamodb().getKeys().get("Id").getN());  
        }  
    }  
    checkpointCounter += 1;  
    if (checkpointCounter % 10 == 0) {  
        try {  
            checkpointer.checkpoint();  
        } catch (Exception e) {  
            logger.error("Error during checkpoint: " + e.getMessage());  
        }  
    }  
}
```

```
        }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Step 4: Ensure that both tables have identical contents

At this point, the source and destination tables' contents are in sync. The application issues Scan requests against both tables to verify that their contents are, in fact, identical.

The DemoHelper class contains a ScanTable method that calls the low-level Scan API. The following example shows how this is used.

```
if (StreamsAdapterDemoHelper.scanTable(dynamoDBClient, srcTable).getItems()
    .equals(StreamsAdapterDemoHelper.scanTable(dynamoDBClient, destTable).getItems()))
{
    System.out.println("Scan result is equal.");
}
else {
    System.out.println("Tables are different!");
}
```

Step 5: Clean up

The demo is complete, so the application deletes the source and destination tables. See the following code example. Even after the tables are deleted, their streams remain available for up to 24 hours, after which they are automatically deleted.

```
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(srcTable));
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(destTable));
```

Complete program: DynamoDB Streams Kinesis adapter

The following is the complete Java program that performs the tasks described in [Walkthrough: DynamoDB Streams Kinesis adapter](#). When you run it, you should see output similar to the following.

```
Creating table KCL-Demo-src
```

```
Creating table KCL-Demo-dest
Table is active.
Creating worker for stream: arn:aws:dynamodb:us-west-2:111122223333:table/KCL-Demo-src/
stream/2015-05-19T22:48:56.601
Starting worker...
Scan result is equal.
Done.
```

Important

To run this program, ensure that the client application has access to DynamoDB and Amazon CloudWatch using policies. For more information, see [Identity-based policies for DynamoDB](#).

The source code consists of four .java files:

- StreamsAdapterDemo.java
- StreamsRecordProcessor.java
- StreamsRecordProcessorFactory.java
- StreamsAdapterDemoHelper.java

StreamsAdapterDemo.java

```
package com.amazonaws.codesamples;

import com.amazonaws.auth.AWS CredentialsProvider;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreams;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreamsClientBuilder;
import com.amazonaws.services.dynamodbv2.model.DeleteTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import
com.amazonaws.services.dynamodbv2.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
```

```
import com.amazonaws.services.dynamodbv2.streamsadapter.StreamsWorkerFactory;
import
com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.InitialPositionInStream;
import
com.amazonaws.services.kinesis.clientlibrary.lib.worker.KinesisClientLibConfiguration;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;

public class StreamsAdapterDemo {
    private static Worker worker;
    private static KinesisClientLibConfiguration workerConfig;
    private static IRecordProcessorFactory recordProcessorFactory;

    private static AmazonDynamoDB dynamoDBClient;
    private static AmazonCloudWatch cloudWatchClient;
    private static AmazonDynamoDBStreams dynamoDBStreamsClient;
    private static AmazonDynamoDBStreamsAdapterClient adapterClient;

    private static String tablePrefix = "KCL-Demo";
    private static String streamArn;

    private static Regions awsRegion = Regions.US_EAST_2;

    private static AWSCredentialsProvider awsCredentialsProvider =
DefaultAWSCredentialsProviderChain.getInstance();

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        System.out.println("Starting demo...");

        dynamoDBClient = AmazonDynamoDBClientBuilder.standard()
            .withRegion(awsRegion)
            .build();
        cloudWatchClient = AmazonCloudWatchClientBuilder.standard()
            .withRegion(awsRegion)
            .build();
        dynamoDBStreamsClient = AmazonDynamoDBStreamsClientBuilder.standard()
            .withRegion(awsRegion)
            .build();
        adapterClient = new AmazonDynamoDBStreamsAdapterClient(dynamoDBStreamsClient);
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";
    }
}
```

```
recordProcessorFactory = new StreamsRecordProcessorFactory(dynamoDBClient,
destTable);

setUpTables();

workerConfig = new KinesisClientLibConfiguration("streams-adapter-demo",
        streamArn,
        awsCredentialsProvider,
        "streams-demo-worker")
        .withMaxRecords(1000)
        .withIdleTimeBetweenReadsInMillis(500)
        .withInitialPositionInStream(InitialPositionInStream.TRIM_HORIZON);

System.out.println("Creating worker for stream: " + streamArn);
worker =
StreamsWorkerFactory.createDynamoDbStreamsWorker(recordProcessorFactory, workerConfig,
adapterClient,
        dynamoDBClient, cloudWatchClient);
System.out.println("Starting worker...");
Thread t = new Thread(worker);
t.start();

Thread.sleep(25000);
worker.shutdown();
t.join();

if (StreamsAdapterDemoHelper.scanTable(dynamoDBClient, srcTable).getItems()
        .equals(StreamsAdapterDemoHelper.scanTable(dynamoDBClient,
destTable).getItems())) {
    System.out.println("Scan result is equal.");
} else {
    System.out.println("Tables are different!");
}

System.out.println("Done.");
cleanupAndExit(0);
}

private static void setUpTables() {
String srcTable = tablePrefix + "-src";
String destTable = tablePrefix + "-dest";
streamArn = StreamsAdapterDemoHelper.createTable(dynamoDBClient, srcTable);
StreamsAdapterDemoHelper.createTable(dynamoDBClient, destTable);
```

```
        awaitTableCreation(srcTable);

        performOps(srcTable);
    }

    private static void awaitTableCreation(String tableName) {
        Integer retries = 0;
        Boolean created = false;
        while (!created && retries < 100) {
            DescribeTableResult result =
StreamsAdapterDemoHelper.describeTable(dynamoDBClient, tableName);
            created = result.getTable().getTableStatus().equals("ACTIVE");
            if (created) {
                System.out.println("Table is active.");
                return;
            } else {
                retries++;
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // do nothing
                }
            }
        }
        System.out.println("Timeout after table creation. Exiting...");
        cleanupAndExit(1);
    }

    private static void performOps(String tableName) {
        StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "101", "test1");
        StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "101", "test2");
        StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "101");
        StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "102", "demo3");
        StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "102", "demo4");
        StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "102");
    }

    private static void cleanupAndExit(Integer returnValue) {
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";
        dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(srcTable));
        dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(destTable));
        System.exit(returnValue);
    }
}
```

```
}
```

StreamsRecordProcessor.java

```
package com.amazonaws.codesamples;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.streamsadapter.model.RecordAdapter;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;
import com.amazonaws.services.kinesis.model.Record;

import java.nio.charset.Charset;

public class StreamsRecordProcessor implements IRecordProcessor {
    private Integer checkpointCounter;

    private final AmazonDynamoDB dynamoDBClient;
    private final String tableName;

    public StreamsRecordProcessor(AmazonDynamoDB dynamoDBClient2, String tableName) {
        this.dynamoDBClient = dynamoDBClient2;
        this.tableName = tableName;
    }

    @Override
    public void initialize(InitializationInput initializationInput) {
        checkpointCounter = 0;
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        for (Record record : processRecordsInput.getRecords()) {
            String data = new String(record.getData().array(),
CharsetName("UTF-8"));
            System.out.println(data);
            if (record instanceof RecordAdapter) {
```

```
com.amazonaws.services.dynamodbv2.model.Record streamRecord =
((RecordAdapter) record)
    .getInternalObject();

    switch (streamRecord.getEventName()) {
        case "INSERT":
        case "MODIFY":
            StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName,
                streamRecord.getDynamodb().getNewImage());
            break;
        case "REMOVE":
            StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName,
                streamRecord.getDynamodb().getKeys().get("Id").getN());
    }
}

checkpointCounter += 1;
if (checkpointCounter % 10 == 0) {
    try {
        processRecordsInput.getCheckpointer().checkpoint();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

}

@Override
public void shutdown(ShutdownInput shutdownInput) {
    if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
        try {
            shutdownInput.getCheckpointer().checkpoint();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

StreamsRecordProcessorFactory.java

```
package com.amazonaws.codesamples;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class StreamsRecordProcessorFactory implements IRecordProcessorFactory {
    private final String tableName;
    private final AmazonDynamoDB dynamoDBClient;

    public StreamsRecordProcessorFactory(AmazonDynamoDB dynamoDBClient, String
tableName) {
        this.tableName = tableName;
        this.dynamoDBClient = dynamoDBClient;
    }

    @Override
    public IRecordProcessor createProcessor() {
        return new StreamsRecordProcessor(dynamoDBClient, tableName);
    }
}
```

StreamsAdapterDemoHelper.java

```
package com.amazonaws.codesamples;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.model.AttributeAction;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.AttributeValueUpdate;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.CreateTableResult;
import com.amazonaws.services.dynamodbv2.model.DeleteItemRequest;
```

```
import com.amazonaws.services.dynamodbv2.model.DescribeTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.PutItemRequest;
import com.amazonaws.services.dynamodbv2.model.ResourceInUseException;
import com.amazonaws.services.dynamodbv2.model.ScanRequest;
import com.amazonaws.services.dynamodbv2.model.ScanResult;
import com.amazonaws.services.dynamodbv2.model.StreamSpecification;
import com.amazonaws.services.dynamodbv2.model.StreamViewType;
import com.amazonaws.services.dynamodbv2.model.UpdateItemRequest;

public class StreamsAdapterDemoHelper {

    /**
     * @return StreamArn
     */
    public static String createTable(AmazonDynamoDB client, String tableName) {
        java.util.List<AttributeDefinition> attributeDefinitions = new
        ArrayList<AttributeDefinition>();
        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

        java.util.List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
        keySchema.add(new
        KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition

        // key

        ProvisionedThroughput provisionedThroughput = new
        ProvisionedThroughput().withReadCapacityUnits(2L)
            .withWriteCapacityUnits(2L);

        StreamSpecification streamSpecification = new StreamSpecification();
        streamSpecification.setStreamEnabled(true);
        streamSpecification.setStreamViewType(StreamViewType.NEW_IMAGE);
        CreateTableRequest createTableRequest = new
        CreateTableRequest().withTableName(tableName)

        .withAttributeDefinitions(attributeDefinitions).withKeySchema(keySchema)

        .withProvisionedThroughput(provisionedThroughput).withStreamSpecification(streamSpecification)
```

```
try {
    System.out.println("Creating table " + tableName);
    CreateTableResult result = client.createTable(createTableRequest);
    return result.getTableDescription().getLatestStreamArn();
} catch (ResourceInUseException e) {
    System.out.println("Table already exists.");
    return describeTable(client, tableName).getTable().getLatestStreamArn();
}
}

public static DescribeTableResult describeTable(AmazonDynamoDB client, String
tableName) {
    return client.describeTable(new
DescribeTableRequest().withTableName(tableName));
}

public static ScanResult scanTable(AmazonDynamoDB dynamoDBClient, String tableName)
{
    return dynamoDBClient.scan(new ScanRequest().withTableName(tableName));
}

public static void putItem(AmazonDynamoDB dynamoDBClient, String tableName, String
id, String val) {
    java.util.Map<String, AttributeValue> item = new HashMap<String,
AttributeValue>();
    item.put("Id", new AttributeValue().withN(id));
    item.put("attribute-1", new AttributeValue().withS(val));

    PutItemRequest putItemRequest = new
PutItemRequest().withTableName(tableName).withItem(item);
    dynamoDBClient.putItem(putItemRequest);
}

public static void putItem(AmazonDynamoDB dynamoDBClient, String tableName,
    java.util.Map<String, AttributeValue> items) {
    PutItemRequest putItemRequest = new
PutItemRequest().withTableName(tableName).withItem(items);
    dynamoDBClient.putItem(putItemRequest);
}

public static void updateItem(AmazonDynamoDB dynamoDBClient, String tableName,
String id, String val) {
    java.util.Map<String, AttributeValue> key = new HashMap<String,
AttributeValue>();
```

```
key.put("Id", new AttributeValue().withN(id));

Map<String, AttributeValueUpdate> attributeUpdates = new HashMap<String,
AttributeValueUpdate>();
AttributeValueUpdate update = new
AttributeValueUpdate().withAction(AttributeAction.PUT)
    .withValue(new AttributeValue().withS(val));
attributeUpdates.put("attribute-2", update);

UpdateItemRequest updateItemRequest = new
UpdateItemRequest().withTableName(tableName).withKey(key)
    .withAttributeUpdates(attributeUpdates);
dynamoDBClient.updateItem(updateItemRequest);
}

public static void deleteItem(AmazonDynamoDB dynamoDBClient, String tableName,
String id) {
    java.util.Map<String, AttributeValue> key = new HashMap<String,
AttributeValue>();
    key.put("Id", new AttributeValue().withN(id));

    DeleteItemRequest deleteItemRequest = new
DeleteItemRequest().withTableName(tableName).withKey(key);
    dynamoDBClient.deleteItem(deleteItemRequest);
}

}
```

DynamoDB Streams low-level API: Java example

Note

The code on this page is not exhaustive and does not handle all scenarios for consuming Amazon DynamoDB Streams. The recommended way to consume stream records from DynamoDB is through the Amazon Kinesis Adapter using the Kinesis Client Library (KCL), as described in [Using the DynamoDB Streams Kinesis adapter to process stream records](#).

This section contains a Java program that shows DynamoDB Streams in action. The program does the following:

1. Creates a DynamoDB table with a stream enabled.
2. Describes the stream settings for this table.
3. Modifies data in the table.
4. Describes the shards in the stream.
5. Reads the stream records from the shards.
6. Cleans up.

When you run the program, you will see output similar to the following.

```
Issuing CreateTable request for TestTableForStreams
Waiting for TestTableForStreams to be created...
Current stream ARN for TestTableForStreams: arn:aws:dynamodb:us-
east-2:123456789012:table/TestTableForStreams/stream/2018-03-20T16:49:55.208
Stream enabled: true
Update view type: NEW_AND_OLD_IMAGES

Performing write activities on TestTableForStreams
Processing item 1 of 100
Processing item 2 of 100
Processing item 3 of 100
...
Processing item 100 of 100

Shard: {ShardId: shardId-1234567890-...,SequenceNumberRange: {StartingSequenceNumber:
01234567890...,},}
    Shard iterator: EjYFEkX2a26eVTWe...
        ApproximateCreationDateTime: Tue Mar 20 09:50:00 PDT 2018,Keys:
        {Id={N: 1,}},NewImage: {Message={S: New item!,}, Id={N: 1,}},SequenceNumber:
        10000000003218256368,SizeBytes: 24,StreamViewType: NEW_AND_OLD_IMAGES}
            {ApproximateCreationDateTime: Tue Mar 20 09:50:00 PDT 2018,Keys: {Id={N:
        1,}},NewImage: {Message={S: This item has changed,}, Id={N: 1,}},OldImage:
        {Message={S: New item!,}, Id={N: 1,}},SequenceNumber: 20000000003218256412,SizeBytes:
        56,StreamViewType: NEW_AND_OLD_IMAGES}
            {ApproximateCreationDateTime: Tue Mar 20 09:50:00 PDT 2018,Keys: {Id={N:
        1,}},OldImage: {Message={S: This item has changed,}, Id={N: 1,}},SequenceNumber:
        30000000003218256413,SizeBytes: 36,StreamViewType: NEW_AND_OLD_IMAGES}
...
Deleting the table...
Demo complete
```

Example

```
package com.amazon.codesamples;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.amazonaws.AmazonClientException;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreams;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreamsClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeAction;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.AttributeValueUpdate;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeStreamRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeStreamResult;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import com.amazonaws.services.dynamodbv2.model.GetRecordsRequest;
import com.amazonaws.services.dynamodbv2.model.GetRecordsResult;
import com.amazonaws.services.dynamodbv2.model.GetShardIteratorRequest;
import com.amazonaws.services.dynamodbv2.model.GetShardIteratorResult;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.Record;
import com.amazonaws.services.dynamodbv2.model.Shard;
import com.amazonaws.services.dynamodbv2.model.ShardIteratorType;
import com.amazonaws.services.dynamodbv2.model.StreamSpecification;
import com.amazonaws.services.dynamodbv2.model.StreamViewType;
import com.amazonaws.services.dynamodbv2.util.TableUtils;

public class StreamsLowLevelDemo {

    public static void main(String args[]) throws InterruptedException {
```

```
AmazonDynamoDB dynamoDBClient = AmazonDynamoDBClientBuilder
    .standard()
    .withRegion(Regions.US_EAST_2)
    .withCredentials(new
DefaultAWSCredentialsProviderChain())
    .build();

AmazonDynamoDBStreams streamsClient =
AmazonDynamoDBStreamsClientBuilder
    .standard()
    .withRegion(Regions.US_EAST_2)
    .withCredentials(new
DefaultAWSCredentialsProviderChain())
    .build();

// Create a table, with a stream enabled
String tableName = "TestTableForStreams";

ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>(
    Arrays.asList(new AttributeDefinition()
        .withAttributeName("Id")
        .withAttributeType("N")));

ArrayList<KeySchemaElement> keySchema = new ArrayList<>(
    Arrays.asList(new KeySchemaElement()
        .withAttributeName("Id")
        .withKeyType(KeyType.HASH))); // Partition key

StreamSpecification streamSpecification = new StreamSpecification()
    .withStreamEnabled(true)
    .withStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES);

 CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)

.withKeySchema(keySchema).withAttributeDefinitions(attributeDefinitions)
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits(10L)
        .withWriteCapacityUnits(10L))
    .withStreamSpecification(streamSpecification);

System.out.println("Issuing CreateTable request for " + tableName);
dynamoDBClient.createTable(createTableRequest);
```

```
System.out.println("Waiting for " + tableName + " to be created...");

try {
    TableUtils.waitUntilActive(dynamoDBClient, tableName);
} catch (AmazonClientException e) {
    e.printStackTrace();
}

// Print the stream settings for the table
DescribeTableResult describeTableResult =
dynamoDBClient.describeTable(tableName);
String streamArn = describeTableResult.getTable().getLatestStreamArn();
System.out.println("Current stream ARN for " + tableName + ":" +
                    describeTableResult.getTable().getLatestStreamArn());
StreamSpecification streamSpec =
describeTableResult.getTable().getStreamSpecification();
System.out.println("Stream enabled: " + streamSpec.getStreamEnabled());
System.out.println("Update view type: " +
streamSpec.getStreamViewType());
System.out.println();

// Generate write activity in the table

System.out.println("Performing write activities on " + tableName);
int maxItemCount = 100;
for (Integer i = 1; i <= maxItemCount; i++) {
    System.out.println("Processing item " + i + " of " +
maxItemCount);

    // Write a new item
    Map<String, AttributeValue> item = new HashMap<>();
    item.put("Id", new AttributeValue().withN(i.toString()));
    item.put("Message", new AttributeValue().withS("New item!"));
    dynamoDBClient.putItem(tableName, item);

    // Update the item
    Map<String, AttributeValue> key = new HashMap<>();
    key.put("Id", new AttributeValue().withN(i.toString()));
    Map<String, AttributeValueUpdate> attributeUpdates = new
HashMap<>();
    attributeUpdates.put("Message", new AttributeValueUpdate()
        .withAction(AttributeAction.PUT)
        .WithValue(new AttributeValue()
```

```
                .withS("This item has
changed")));
        dynamoDBClient.updateItem(tableName, key, attributeUpdates);

        // Delete the item
        dynamoDBClient.deleteItem(tableName, key);
    }

    // Get all the shard IDs from the stream. Note that DescribeStream
returns
    // the shard IDs one page at a time.
    String lastEvaluatedShardId = null;

    do {
        DescribeStreamResult describeStreamResult =
streamsClient.describeStream(
            new DescribeStreamRequest()
                .withStreamArn(streamArn)

        .withExclusiveStartShardId(lastEvaluatedShardId));
        List<Shard> shards =
describeStreamResult.getStreamDescription().getShards();

        // Process each shard on this page

        for (Shard shard : shards) {
            String shardId = shard.getShardId();
            System.out.println("Shard: " + shard);

            // Get an iterator for the current shard

            GetShardIteratorRequest getShardIteratorRequest = new
GetShardIteratorRequest()
                .withStreamArn(streamArn)
                .withShardId(shardId)

        .withShardIteratorType(ShardIteratorType.TRIM_HORIZON);
        GetShardIteratorResult getShardIteratorResult =
streamsClient

        .getShardIterator(getShardIteratorRequest);
        String currentShardIter =
getShardIteratorResult.getShardIterator();
```

```
// Shard iterator is not null until the Shard is sealed
// (marked as READ_ONLY).
// To prevent running the loop until the Shard is
sealed, which will be on
// average
// 4 hours, we process only the items that were written
into DynamoDB and then
// exit.
int processedRecordCount = 0;
while (currentShardIter != null && processedRecordCount
< maxItemCount) {
    System.out.println("      Shard iterator: " +
currentShardIter.substring(380));

    // Use the shard iterator to read the stream
records

    GetRecordsResult getRecordsResult =
streamsClient
        .getRecords(new
GetRecordsRequest()

.withShardIterator(currentShardIter));
    List<Record> records =
getRecordsResult.getRecords();
    for (Record record : records) {
        System.out.println("      " +
record.getDynamodb());
    }
    processedRecordCount += records.size();
    currentShardIter =
getRecordsResult.getNextShardIterator();
}
}

// If LastEvaluatedShardId is set, then there is
// at least one more page of shard IDs to retrieve
lastEvaluatedShardId =
describeStreamResult.getStreamDescription().getLastEvaluatedShardId();

} while (lastEvaluatedShardId != null);

// Delete the table
System.out.println("Deleting the table...");
```

```
dynamoDBClient.deleteTable(tableName);

System.out.println("Demo complete");

}
```

DynamoDB Streams and AWS Lambda triggers

Topics

- [Tutorial #1: Using filters to process all events with Amazon DynamoDB and AWS Lambda using the AWS CLI](#)
- [Tutorial #2: Using filters to process some events with DynamoDB and Lambda.](#)
- [Best practices with Lambda](#)

Amazon DynamoDB is integrated with AWS Lambda so that you can create *triggers*—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream Amazon Resource Name (ARN) with an AWS Lambda function that you write. All mutation actions to that DynamoDB table can then be captured as an item on the stream. For example, you can set a trigger so that when an item in a table is modified a new record immediately appears in that table's stream.

 **Note**

You can have more than two Lambda functions subscribed, but this may result in read throttling if using more than two Lambda functions to one DynamoDB stream.

The [AWS Lambda](#) service polls the stream for new records four times per second. When new stream records are available, your Lambda function is synchronously invoked. You can subscribe up to two Lambda functions to the same DynamoDB stream.

The Lambda function can send a notification, initiate a workflow, or perform many other actions that you specify. You can write a Lambda function to simply copy each stream record to persistent storage, such as Amazon S3 File Gateway (Amazon S3), and create a permanent audit trail of write activity in your table. Or suppose that you have a mobile gaming app that writes to a GameScores

table. Whenever the TopScore attribute of the GameScores table is updated, a corresponding stream record is written to the table's stream. This event could then trigger a Lambda function that posts a congratulatory message on a social media network. This function could also be written to ignore any stream records that are not updates to GameScores, or that do not modify the TopScore attribute.

If your function returns an error, Lambda retries the batch until it processes successfully or the data expires. You can also configure Lambda to retry with a smaller batch, limit the number of retries, discard records once they become too old, and other options.

As performance best practices, the Lambda function needs to be short lived. To avoid introducing unnecessary processing delays, it also should not execute complex logic. For a high velocity stream in particular, it is better to trigger an asynchronous post-processing step function workflows than synchronous long running Lambdas.

You cannot use the same Lambda trigger across different AWS accounts. Both the DynamoDB table and the Lambda functions must belong to the same AWS account.

For more information about AWS Lambda, see the [AWS Lambda Developer Guide](#).

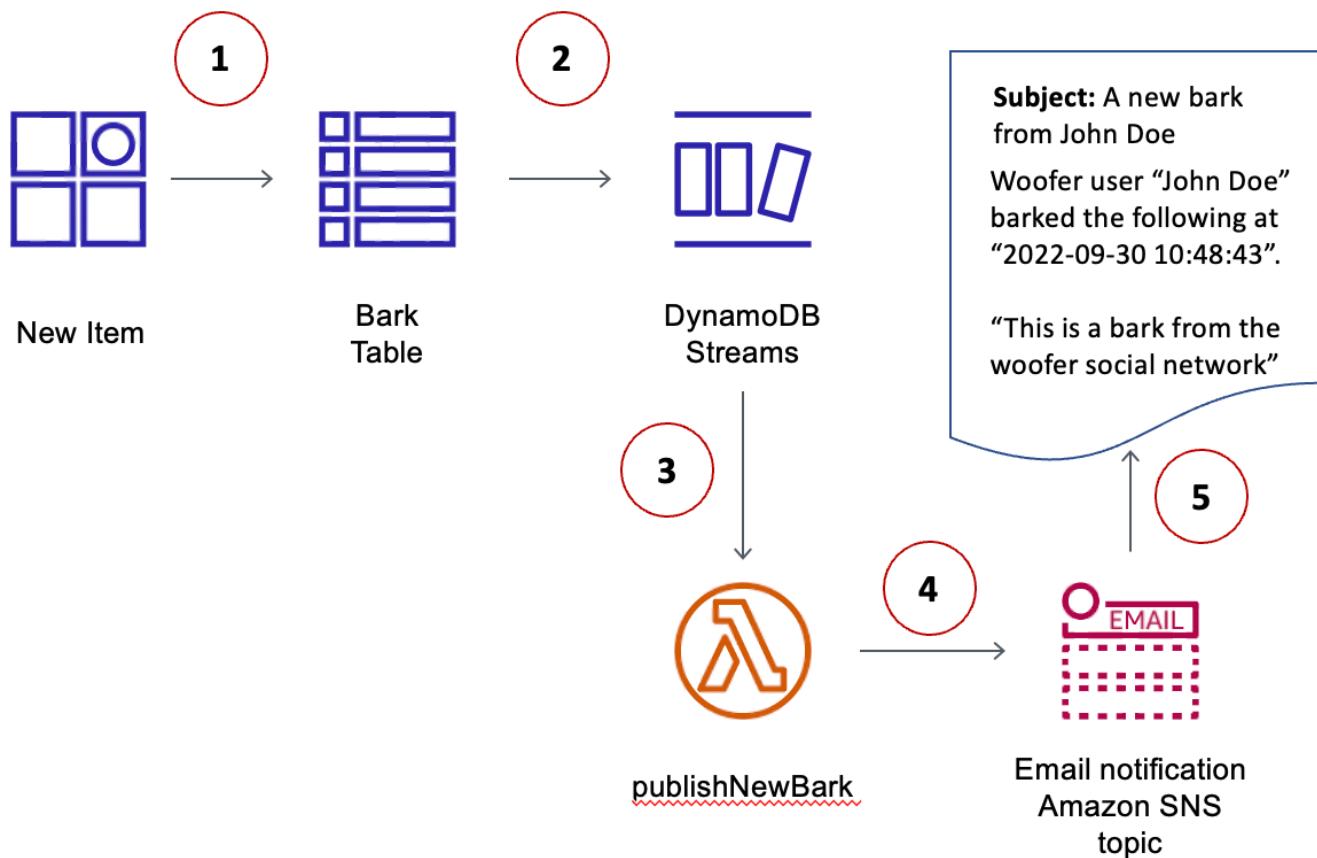
Tutorial #1: Using filters to process all events with Amazon DynamoDB and AWS Lambda using the AWS CLI

Topics

- [Step 1: Create a DynamoDB table with a stream enabled](#)
- [Step 2: Create a Lambda execution role](#)
- [Step 3: Create an Amazon SNS topic](#)
- [Step 4: Create and test a Lambda function](#)
- [Step 5: Create and test a trigger](#)

In this tutorial, you will create an AWS Lambda trigger to process a stream from a DynamoDB table.

The scenario for this tutorial is Woofer, a simple social network. Woofer users communicate using *barks* (short text messages) that are sent to other Woofer users. The following diagram shows the components and workflow for this application.



1. A user writes an item to a DynamoDB table (BarkTable). Each item in the table represents a bark.
2. A new stream record is written to reflect that a new item has been added to BarkTable.
3. The new stream record triggers an AWS Lambda function (`publishNewBark`).
4. If the stream record indicates that a new item was added to BarkTable, the Lambda function reads the data from the stream record and publishes a message to a topic in Amazon Simple Notification Service (Amazon SNS).
5. The message is received by subscribers to the Amazon SNS topic. (In this tutorial, the only subscriber is an email address.)

Before You Begin

This tutorial uses the AWS Command Line Interface AWS CLI. If you have not done so already, follow the instructions in the [AWS Command Line Interface User Guide](#) to install and configure the AWS CLI.

Step 1: Create a DynamoDB table with a stream enabled

In this step, you create a DynamoDB table (BarkTable) to store all of the barks from Woofer users. The primary key is composed of Username (partition key) and Timestamp (sort key). Both of these attributes are of type string.

BarkTable has a stream enabled. Later in this tutorial, you create a trigger by associating an AWS Lambda function with the stream.

1. Enter the following command to create the table.

```
aws dynamodb create-table \
    --table-name BarkTable \
    --attribute-definitions AttributeName=Username,AttributeType=S
    AttributeName=Timestamp,AttributeType=S \
    --key-schema AttributeName=Username,KeyType=HASH
    AttributeName=Timestamp,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

2. In the output, look for the LatestStreamArn.

```
...
"LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/
stream/timestamp
..."
```

Make a note of the *region* and the *accountID*, because you need them for the other steps in this tutorial.

Step 2: Create a Lambda execution role

In this step, you create an AWS Identity and Access Management (IAM) role (WooferLambdaRole) and assign permissions to it. This role is used by the Lambda function that you create in [Step 4: Create and test a Lambda function](#).

You also create a policy for the role. The policy contains all of the permissions that the Lambda function needs at runtime.

1. Create a file named trust-relationship.json with the following contents.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "lambda.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

2. Enter the following command to create WooferLambdaRole.

```
aws iam create-role --role-name WooferLambdaRole \  
    --path "/service-role/" \  
    --assume-role-policy-document file://trust-relationship.json
```

3. Create a file named role-policy.json with the following contents. (Replace *region* and *accountID* with your AWS Region and account ID.)

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "logs>CreateLogGroup",  
                "logs>CreateLogStream",  
                "logs>PutLogEvents"  
            ],  
            "Resource": "arn:aws:logs:region:accountID:*"  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeStream",  
                "dynamodb:GetRecords",  
                "dynamodb:GetShardIterator",  
                "dynamodb>ListStreams"  
            ],  
            "Resource": "arn:aws:dynamodb:region:accountID:  
        }  
    ]  
}
```

```
"Resource": "arn:aws:dynamodb:region:accountID:table/BarkTable/stream/*",
},
{
    "Effect": "Allow",
    "Action": [
        "sns:Publish"
    ],
    "Resource": [
        "*"
    ]
}
]
```

The policy has four statements that allow `WooferLambdaRole` to do the following:

- Run a Lambda function (`publishNewBark`). You create the function later in this tutorial.
- Access Amazon CloudWatch Logs. The Lambda function writes diagnostics to CloudWatch Logs at runtime.
- Read data from the DynamoDB stream for `BarkTable`.
- Publish messages to Amazon SNS.

4. Enter the following command to attach the policy to `WooferLambdaRole`.

```
aws iam put-role-policy --role-name WooferLambdaRole \
    --policy-name WooferLambdaRolePolicy \
    --policy-document file://role-policy.json
```

Step 3: Create an Amazon SNS topic

In this step, you create an Amazon SNS topic (`wooferTopic`) and subscribe an email address to it. Your Lambda function uses this topic to publish new barks from Woofer users.

1. Enter the following command to create a new Amazon SNS topic.

```
aws sns create-topic --name wooferTopic
```

2. Enter the following command to subscribe an email address to `wooferTopic`.
(Replace `region` and `accountID` with your AWS Region and account ID, and replace `example@example.com` with a valid email address.)

```
aws sns subscribe \
--topic-arn arn:aws:sns:region:accountID:wooferTopic \
--protocol email \
--notification-endpoint example@example.com
```

3. Amazon SNS sends a confirmation message to your email address. Choose the **Confirm subscription** link in that message to complete the subscription process.

Step 4: Create and test a Lambda function

In this step, you create an AWS Lambda function (`publishNewBark`) to process stream records from `BarkTable`.

The `publishNewBark` function processes only the stream events that correspond to new items in `BarkTable`. The function reads data from such an event, and then invokes Amazon SNS to publish it.

1. Create a file named `publishNewBark.js` with the following contents. Replace `region` and `accountID` with your AWS Region and account ID.

```
'use strict';
var AWS = require("aws-sdk");
var sns = new AWS.SNS();

exports.handler = (event, context, callback) => {

    event.Records.forEach((record) => {
        console.log('Stream record: ', JSON.stringify(record, null, 2));

        if (record.eventName == 'INSERT') {
            var who = JSON.stringify(record.dynamodb.NewImage.Username.S);
            var when = JSON.stringify(record.dynamodb.NewImage.Timestamp.S);
            var what = JSON.stringify(record.dynamodb.NewImage.Message.S);
            var params = {
                Subject: 'A new bark from ' + who,
                Message: 'Woofer user ' + who + ' barked the following at ' + when
                + ':\n\n' + what,
            };
            sns.publish(params).promise();
        }
    });
}
```

```
        TopicArn: 'arn:aws:sns:region:accountID:wooferTopic'
    };
    sns.publish(params, function(err, data) {
        if (err) {
            console.error("Unable to send message. Error JSON:",
JSON.stringify(err, null, 2));
        } else {
            console.log("Results from sending message: ",
JSON.stringify(data, null, 2));
        }
    });
});
callback(null, `Successfully processed ${event.Records.length} records.`);
};
```

2. Create a zip file to contain publishNewBark.js. If you have the zip command-line utility, you can enter the following command to do this.

```
zip publishNewBark.zip publishNewBark.js
```

3. When you create the Lambda function, you specify the Amazon Resource Name (ARN) for WooferLambdaRole, which you created in [Step 2: Create a Lambda execution role](#). Enter the following command to retrieve this ARN.

```
aws iam get-role --role-name WooferLambdaRole
```

In the output, look for the ARN for WooferLambdaRole.

```
...
"Arn": "arn:aws:iam::region:role/service-role/WooferLambdaRole"
...
```

Enter the following command to create the Lambda function. Replace *roleARN* with the ARN for WooferLambdaRole.

```
aws lambda create-function \
--region region \
--function-name publishNewBark \
--zip-file fileb://publishNewBark.zip \
```

```
--role roleARN \
--handler publishNewBark.handler \
--timeout 5 \
--runtime nodejs16.x
```

- Now test `publishNewBark` to verify that it works. To do this, you provide input that resembles a real record from DynamoDB Streams.

Create a file named `payload.json` with the following contents. Replace `region` and `accountID` with your AWS Region and account ID.

```
{
  "Records": [
    {
      "eventID": "7de3041dd709b024af6f29e4fa13d34c",
      "eventName": "INSERT",
      "eventVersion": "1.1",
      "eventSource": "aws:dynamodb",
      "awsRegion": "region",
      "dynamodb": {
        "ApproximateCreationDateTime": 1479499740,
        "Keys": {
          "Timestamp": {
            "S": "2016-11-18:12:09:36"
          },
          "Username": {
            "S": "John Doe"
          }
        },
        "NewImage": {
          "Timestamp": {
            "S": "2016-11-18:12:09:36"
          },
          "Message": {
            "S": "This is a bark from the Woofer social network"
          },
          "Username": {
            "S": "John Doe"
          }
        },
        "SequenceNumber": "13021600000000001596893679",
        "SizeBytes": 112,
        "StreamViewType": "NEW_IMAGE"
      }
    }
  ]
}
```

```
    },
    "eventSourceARN": "arn:aws:dynamodb:region:account ID:table/BarkTable/
stream/2016-11-16T20:42:48.104"
}
]
```

Enter the following command to test the publishNewBark function.

```
aws lambda invoke --function-name publishNewBark --payload file://payload.json --
cli-binary-format raw-in-base64-out output.txt
```

If the test was successful, you will see the following output.

```
{  
  "StatusCode": 200,  
  "ExecutedVersion": "$LATEST"  
}
```

In addition, the output.txt file will contain the following text.

```
"Successfully processed 1 records."
```

You will also receive a new email message within a few minutes.

Note

AWS Lambda writes diagnostic information to Amazon CloudWatch Logs. If you encounter errors with your Lambda function, you can use these diagnostics for troubleshooting purposes:

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Logs**.
3. Choose the following log group: /aws/lambda/publishNewBark
4. Choose the latest log stream to view the output (and errors) from the function.

Step 5: Create and test a trigger

In [Step 4: Create and test a Lambda function](#), you tested the Lambda function to ensure that it ran correctly. In this step, you create a *trigger* by associating the Lambda function (publishNewBark) with an event source (the BarkTable stream).

- When you create the trigger, you need to specify the ARN for the BarkTable stream. Enter the following command to retrieve this ARN.

```
aws dynamodb describe-table --table-name BarkTable
```

In the output, look for the LatestStreamArn.

```
...
"LatestStreamArn": "arn:aws:dynamodb:<region>:<accountID>:table/BarkTable/
stream/<timestamp>
..."
```

- Enter the following command to create the trigger. Replace *streamARN* with the actual stream ARN.

```
aws lambda create-event-source-mapping \
--region <region> \
--function-name publishNewBark \
--event-source <streamARN> \
--batch-size 1 \
--starting-position TRIM_HORIZON
```

- Test the trigger. Enter the following command to add an item to BarkTable.

```
aws dynamodb put-item \
--table-name BarkTable \
--item Username={"S": "Jane
Doe"},Timestamp={"S": "2016-11-18:14:32:17"},Message={"S": "Testing...1...2...3"}
```

You should receive a new email message within a few minutes.

- Open the DynamoDB console and add a few more items to BarkTable. You must specify values for the Username and Timestamp attributes. (You should also specify a value for Message, even though it is not required.) You should receive a new email message for each item you add to BarkTable.

The Lambda function processes only new items that you add to BarkTable. If you update or delete an item in the table, the function does nothing.

Note

AWS Lambda writes diagnostic information to Amazon CloudWatch Logs. If you encounter errors with your Lambda function, you can use these diagnostics for troubleshooting purposes.

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Logs**.
3. Choose the following log group: /aws/lambda/publishNewBark
4. Choose the latest log stream to view the output (and errors) from the function.

Tutorial #2: Using filters to process some events with DynamoDB and Lambda.

Topics

- [Putting it all together - AWS CloudFormation](#)
- [Putting it all together - CDK](#)

In this tutorial, you will create an AWS Lambda trigger to process only some events in a stream from a DynamoDB table.

With [Lambda event filtering](#) you can use filter expressions to control which events Lambda sends to your function for processing. You can configure up to 5 different filters per DynamoDB streams. If you are using batching windows, Lambda applies the filter criteria to each new event to see if it should be included in the current batch.

Filters are applied via structures called `FilterCriteria`. The 3 main attributes of `FilterCriteria` are `metadata properties`, `data properties` and `filter patterns`.

Here is an example structure of a DynamoDB Streams event:

{

```
"eventID": "c9fbe7d0261a5163fc6940593e41797",
"eventName": "INSERT",
"eventVersion": "1.1",
"eventSource": "aws:dynamodb",
"awsRegion": "us-east-2",
"dynamodb": {
    "ApproximateCreationDateTime": 1664559083.0,
    "Keys": {
        "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
        "PK": { "S": "COMPANY#1000" }
    },
    "NewImage": {
        "quantity": { "N": "50" },
        "company_id": { "S": "1000" },
        "fabric": { "S": "Florida Chocolates" },
        "price": { "N": "15" },
        "stores": { "N": "5" },
        "product_id": { "S": "1000" },
        "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
        "PK": { "S": "COMPANY#1000" },
        "state": { "S": "FL" },
        "type": { "S": "" }
    },
    "SequenceNumber": "700000000000888747038",
    "SizeBytes": 174,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
},
"eventSourceARN": "arn:aws:dynamodb:us-east-2:111122223333:table/chocolate-table-StreamsSampleDDBTable-LU0I6UXQY7J1/stream/2022-09-30T17:05:53.209"
}
```

The `metadata` properties are the fields of the event object. In the case of DynamoDB Streams, the `metadata` properties are fields like `dynamodb` or `eventName`.

The `data` properties are the fields of the event body. To filter on `data` properties, make sure to contain them in `FilterCriteria` within the proper key. For DynamoDB event sources, the data key is `NewImage` or `OldImage`.

Finally, the filter rules will define the filter expression that you want to apply to a specific property. Here are some examples:

Comparison operator	Example	Rule syntax (Partial)
Null	Product Type is null	{ "product_type": { "S": null } }
Empty	Product name is empty	{ "product_name": { "S": [""] } }
Equals	State equals Florida	{ "state": { "S": ["FL"] } }
And	Product state equals Florida and product category Chocolate	{ "state": { "S": ["FL"] }, "category": { "S": ["CHOCOLATE"] } }
Or	Product state is Florida or California	{ "state": { "S": ["FL", "CA"] } }
Not	Product state is not Florida	{"state": {"S": [{"anything-but": ["FL"]}]} }
Exists	Product Homemade exists	{"homemade": {"S": [{"exists": true}]} }
Does not exist	Product Homemade does not exist	{"homemade": {"S": [{"exists": false}]} }
Begins with	PK begins with COMPANY	{"PK": {"S": [{"prefix": "COMPANY"}]} }

You can specify up to 5 event filtering patterns for a Lambda function. Notice that each one of those 5 events will be evaluated as a logical OR. So if you configure two filters named `Filter_One` and `Filter_Two`, the Lambda function will execute `Filter_One OR Filter_Two`.

Note

In the [Lambda event filtering](#) page there are some options to filter and compare numeric values, however in the case of DynamoDB filter events it doesn't apply because numbers in DynamoDB are stored as strings. For example "quantity": { "N": "50" }, we know its a number because of the "N" property.

Putting it all together - AWS CloudFormation

To show event filtering functionality in practice, here is a sample CloudFormation template. This template will generate a Simple DynamoDB table with a Partition Key PK and a Sort Key SK with Amazon DynamoDB Streams enabled. It will create a lambda function and a simple Lambda Execution role that will allow write logs to Amazon Cloudwatch, and read the events from the Amazon DynamoDB Stream. It will also add the event source mapping between the DynamoDB Streams and the Lambda function, so the function can be executed every time there is an event in the Amazon DynamoDB Stream.

```
AWSTemplateFormatVersion: "2010-09-09"

Description: Sample application that presents AWS Lambda event source filtering
with Amazon DynamoDB Streams.
```

Resources:

```
StreamsSampleDDBTable:
  Type: AWS::DynamoDB::Table
  Properties:
    AttributeDefinitions:
      - AttributeName: "PK"
        AttributeType: "S"
      - AttributeName: "SK"
        AttributeType: "S"
    KeySchema:
      - AttributeName: "PK"
        KeyType: "HASH"
      - AttributeName: "SK"
        KeyType: "RANGE"
    StreamSpecification:
      StreamViewType: "NEW_AND_OLD_IMAGES"
    ProvisionedThroughput:
      ReadCapacityUnits: 5
```

```
        WriteCapacityUnits: 5

    LambdaExecutionRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: "2012-10-17"
                Statement:
                    - Effect: Allow
                    Principal:
                        Service:
                            - lambda.amazonaws.com
                    Action:
                        - sts:AssumeRole
            Path: "/"
            Policies:
                - PolicyName: root
                PolicyDocument:
                    Version: "2012-10-17"
                    Statement:
                        - Effect: Allow
                        Action:
                            - logs>CreateLogGroup
                            - logs>CreateLogStream
                            - logs>PutLogEvents
                        Resource: arn:aws:logs:*:*:*
                    - Effect: Allow
                    Action:
                        - dynamodb>DescribeStream
                        - dynamodb>GetRecords
                        - dynamodb>GetShardIterator
                        - dynamodb>ListStreams
                    Resource: !GetAtt StreamsSampleDDBTable.StreamArn

    EventSourceDDBTableStream:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: True
            EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
            FunctionName: !GetAtt ProcessEventLambda.Arn
            StartingPosition: LATEST

    ProcessEventLambda:
```

```
Type: AWS::Lambda::Function
Properties:
  Runtime: python3.7
  Timeout: 300
  Handler: index.handler
  Role: !GetAtt LambdaExecutionRole.Arn
Code:
  ZipFile: |
    import logging

    LOGGER = logging.getLogger()
    LOGGER.setLevel(logging.INFO)

    def handler(event, context):
        LOGGER.info('Received Event: %s', event)
        for rec in event['Records']:
            LOGGER.info('Record: %s', rec)

Outputs:
StreamsSampleDDBTable:
  Description: DynamoDB Table ARN created for this example
  Value: !GetAtt StreamsSampleDDBTable.Arn
StreamARN:
  Description: DynamoDB Table ARN created for this example
  Value: !GetAtt StreamsSampleDDBTable.StreamArn
```

After you deploy this cloud formation template you can insert the following Amazon DynamoDB Item:

```
{
  "PK": "COMPANY#1000",
  "SK": "PRODUCT#CHOCOLATE#DARK",
  "company_id": "1000",
  "type": "",
  "state": "FL",
  "stores": 5,
  "price": 15,
  "quantity": 50,
  "fabric": "Florida Chocolates"
}
```

Thanks to the simple lambda function included inline in this cloud formation template, you will see the events in the Amazon CloudWatch log groups for the lambda function as follows:

```
{  
    "eventID": "c9fbe7d0261a5163fcb6940593e41797",  
    "eventName": "INSERT",  
    "eventVersion": "1.1",  
    "eventSource": "aws:dynamodb",  
    "awsRegion": "us-east-2",  
    "dynamodb": {  
        "ApproximateCreationDateTime": 1664559083.0,  
        "Keys": {  
            "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },  
            "PK": { "S": "COMPANY#1000" }  
        },  
        "NewImage": {  
            "quantity": { "N": "50" },  
            "company_id": { "S": "1000" },  
            "fabric": { "S": "Florida Chocolates" },  
            "price": { "N": "15" },  
            "stores": { "N": "5" },  
            "product_id": { "S": "1000" },  
            "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },  
            "PK": { "S": "COMPANY#1000" },  
            "state": { "S": "FL" },  
            "type": { "S": "" }  
        },  
        "SequenceNumber": "700000000000888747038",  
        "SizeBytes": 174,  
        "StreamViewType": "NEW_AND_OLD_IMAGES"  
    },  
    "eventSourceARN": "arn:aws:dynamodb:us-east-2:111122223333:table/chocolate-table-StreamsSampleDDBTable-LU0I6UXQY7J1/stream/2022-09-30T17:05:53.209"  
}
```

Filter Examples

- Only products that matches a given state

This example modifies the CloudFormation template to include a filter to match all products which come from Florida, with the abbreviation “FL”.

```
EventSourceDDBTableStream:  
  Type: AWS::Lambda::EventSourceMapping  
  Properties:
```

```
BatchSize: 1
Enabled: True
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb": { "NewImage": { "state": { "S": ["FL"] } } } }'
EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
FunctionName: !GetAtt ProcessEventLambda.Arn
StartingPosition: LATEST
```

Once you redeploy the stack, you can add the following DynamoDB item to the table. Note that it will not appear in the Lambda function logs, because the product in this example is from California.

```
{
  "PK": "COMPANY#1000",
  "SK": "PRODUCT#CHOCOLATE#DARK#1000",
  "company_id": "1000",
  "fabric": "Florida Chocolates",
  "price": 15,
  "product_id": "1000",
  "quantity": 50,
  "state": "CA",
  "stores": 5,
  "type": ""
}
```

- **Only the items that starts with some values in the PK and SK**

This example modifies the CloudFormation template to include the following condition:

```
EventSourceDDBTableStream:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    BatchSize: 1
    Enabled: True
    FilterCriteria:
      Filters:
        - Pattern: '{"dynamodb": {"Keys": {"PK": { "S": [{"prefix": "COMPANY"}]}, "SK": { "S": [{"prefix": "PRODUCT"}]} }} }'
    EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
    FunctionName: !GetAtt ProcessEventLambda.Arn
    StartingPosition: LATEST
```

Notice the AND condition requires the condition to be inside the pattern, where Keys PK and SK are in the same expression separated by comma.

Either start with some values on PK and SK or is from certain state.

This example modifies the CloudFormation template to include the following conditions:

```
EventSourceDDBTableStream:  
  Type: AWS::Lambda::EventSourceMapping  
  Properties:  
    BatchSize: 1  
    Enabled: True  
    FilterCriteria:  
      Filters:  
        - Pattern: '{"dynamodb": {"Keys": {"PK": {"S": [{"prefix": "COMPANY"}]}, "SK": {"S": [{"prefix": "PRODUCT"}]}}}}'  
        - Pattern: '{ "dynamodb": { "NewImage": { "state": { "S": ["FL"] } } } }'  
  EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn  
  FunctionName: !GetAtt ProcessEventLambda.Arn  
  StartingPosition: LATEST
```

Notice the OR condition is added by introducing new patterns in the filter section.

Putting it all together - CDK

The following sample CDK project formation template walks through event filtering functionality. Before working with this CDK project you will need to [install the pre-requisites](#) including [running preparation scripts](#).

Create a CDK project

First create a new AWS CDK project, by invoking `cdk init` in an empty directory.

```
mkdir ddb_filters  
cd ddb_filters  
cdk init app --language python
```

The `cdk init` command uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Any hyphens in the folder name are converted to underscores. The name should otherwise follow the form of a Python identifier. For example, it should not start with a number or contain spaces.

To work with the new project, activate its virtual environment. This allows the project's dependencies to be installed locally in the project folder, instead of globally.

```
source .venv/bin/activate  
python -m pip install -r requirements.txt
```

Note

You may recognize this as the Mac/Linux command to activate a virtual environment. The Python templates include a batch file, `source.bat`, that allows the same command to be used on Windows. The traditional Windows command `.venv\Scripts\activate.bat` works too. If you initialized your AWS CDK project using AWS CDK Toolkit v1.70.0 or earlier, your virtual environment is in the `.env` directory instead of `.venv`.

Base Infrastructure

Open the file `./ddb_filters/ddb_filters_stack.py` with your preferred text editor. This file was auto generated when you created the AWS CDK project.

Next, add the functions `_create_ddb_table` and `_set_ddb_trigger_function`. These functions will create a DynamoDB table with partition key PK and sort key SK in provision mode on-demand mode, with Amazon DynamoDB Streams enabled by default to show New and Old images.

The Lambda function will be stored in the folder `lambda` under the file `app.py`. This file will be created later. It will include an environment variable `APP_TABLE_NAME`, which will be the name of the Amazon DynamoDB Table created by this stack. In the same function we will grant stream read permissions to the Lambda function. Finally, it will subscribe to the DynamoDB Streams as the event source for the lambda function.

At the end of the file in the `__init__` method, you will call the respective constructs to initialize them in the stack. For bigger projects that require additional components and services, it might be best to define these constructs outside the base stack.

```
import os  
import json  
  
import aws_cdk as cdk  
from aws_cdk import (
```

```
Stack,
aws_lambda as _lambda,
aws_dynamodb as dynamodb,
)
from constructs import Construct

class DdbFiltersStack(Stack):

    def _create_ddb_table(self):
        dynamodb_table = dynamodb.Table(
            self,
            "AppTable",
            partition_key=dynamodb.Attribute(
                name="PK", type=dynamodb.AttributeType.STRING
            ),
            sort_key=dynamodb.Attribute(
                name="SK", type=dynamodb.AttributeType.STRING),
            billing_mode=dynamodb.BillingMode.PAY_PER_REQUEST,
            stream=dynamodb.StreamViewType.NEW_AND_OLD_IMAGES,
            removal_policy=cdk.RemovalPolicy.DESTROY,
        )

        cdk.CfnOutput(self, "AppTableName", value=dynamodb_table.table_name)
        return dynamodb_table

    def _set_ddb_trigger_function(self, ddb_table):
        events_lambda = _lambda.Function(
            self,
            "LambdaHandler",
            runtime=_lambda.Runtime.PYTHON_3_9,
            code=_lambda.Code.from_asset("lambda"),
            handler="app.handler",
            environment={
                "APP_TABLE_NAME": ddb_table.table_name,
            },
        )

        ddb_table.grant_stream_read(events_lambda)

        event_subscription = _lambda.CfnEventSourceMapping(
            scope=self,
            id="companyInsertsOnlyEventSourceMapping",
            function_name=events_lambda.function_name,
```

```
        event_source_arn=ddb_table.table_stream_arn,
        maximum_batching_window_in_seconds=1,
        starting_position="LATEST",
        batch_size=1,
    )

def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
    super().__init__(scope, construct_id, **kwargs)

    ddb_table = self._create_ddb_table()
    self._set_ddb_trigger_function(ddb_table)
```

Now we will create a very simple lambda function that will print the logs into Amazon CloudWatch. To do this, create a new folder called `lambda`.

```
mkdir lambda
touch app.py
```

Using your favorite text editor, add the following content to the `app.py` file:

```
import logging

LOGGER = logging.getLogger()
LOGGER.setLevel(logging.INFO)

def handler(event, context):
    LOGGER.info('Received Event: %s', event)
    for rec in event['Records']:
        LOGGER.info('Record: %s', rec)
```

Ensuring you are in the `/ddb_filters/` folder, type the following command to create the sample application:

```
cdk deploy
```

At some point you will be asked to confirm if you want to deploy the solution. Accept the changes by typing Y.

```
#####
#####
```

```
# + # ${LambdaHandler/ServiceRole} # arn:${AWS::Partition}:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole #
#####
Do you wish to deploy these changes (y/n)? y
...
# Deployment time: 67.73s

Outputs:
DdbFiltersStack.AppTableName = DdbFiltersStack-AppTable815C50BC-1M1W7209V5YPP
Stack ARN:
arn:aws:cloudformation:us-east-2:111122223333:stack/
DdbFiltersStack/66873140-40f3-11ed-8e93-0a74f296a8f6
```

Once the changes are deployed, open your AWS console and add one item to your table.

```
{
    "PK": "COMPANY#1000",
    "SK": "PRODUCT#CHOCOLATE#DARK",
    "company_id": "1000",
    "type": "",
    "state": "FL",
    "stores": 5,
    "price": 15,
    "quantity": 50,
    "fabric": "Florida Chocolates"
}
```

The CloudWatch logs should now contain all the information from this entry.

Filter Examples

- **Only products that matches a given state**

Open the file `ddb_filters/ddb_filters/ddb_filters_stack.py`, and modify it to include the filter that matches all the products that are equals to "FL". This can be revised just below the `event_subscription` in line 45.

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
```

```
value={  
    "Filters": [  
        {  
            "Pattern": json.dumps(  
                {"dynamodb": {"NewImage": {"state": {"S": ["FL"]}}}}  
            )  
        },  
    ],  
},  
)
```

- Only the items that starts with some values in the PK and SK

Modify the python script to include the following condition:

```
event_subscription.add_property_override(  
    property_path="FilterCriteria",  
    value={  
        "Filters": [  
            {  
                "Pattern": json.dumps(  
                    {  
                        "dynamodb": {  
                            "Keys": {  
                                "PK": {"S": [{"prefix": "COMPANY"}]},  
                                "SK": {"S": [{"prefix": "PRODUCT"}]},  
                            }  
                        }  
                    }  
                )  
            },  
        ],  
    },  
)
```

- Either start with some values on PK and SK or is from certain state.

Modify the python script to include the following conditions:

```
event_subscription.add_property_override(  
)
```

```
property_path="FilterCriteria",
value={
    "Filters": [
        {
            "Pattern": json.dumps(
                {
                    "dynamodb": {
                        "Keys": {
                            "PK": {"S": [{"prefix": "COMPANY"}]},
                            "SK": {"S": [{"prefix": "PRODUCT"}]}
                        }
                    }
                }
            )
        },
        {
            "Pattern": json.dumps(
                {"dynamodb": {"NewImage": {"state": {"S": ["FL"]}}}}
            )
        },
    ],
},
)
```

Notice that the OR condition is added by adding more elements to the Filters array.

Cleanup

Locate the filter stack in the base of your working directory, and execute `cdk destroy`. You will be asked to confirm the resource deletion:

```
cdk destroy
Are you sure you want to delete: DdbFiltersStack (y/n)? y
```

Best practices with Lambda

An AWS Lambda function runs within a *container*—an execution environment that is isolated from other functions. When you run a function for the first time, AWS Lambda creates a new container and begins executing the function's code.

A Lambda function has a *handler* that is run once per invocation. The handler contains the main business logic for the function. For example, the Lambda function shown in [Step 4: Create and test a Lambda function](#) has a handler that can process records in a DynamoDB stream.

You can also provide initialization code that runs one time only—after the container is created, but before AWS Lambda runs the handler for the first time. The Lambda function shown in [Step 4: Create and test a Lambda function](#) has initialization code that imports the SDK for JavaScript in Node.js, and creates a client for Amazon SNS. These objects should only be defined once, outside of the handler.

After the function runs, AWS Lambda might opt to reuse the container for subsequent invocations of the function. In this case, your function handler might be able to reuse the resources that you defined in your initialization code. (You cannot control how long AWS Lambda will retain the container, or whether the container will be reused at all.)

For DynamoDB triggers using AWS Lambda, we recommend the following:

- AWS service clients should be instantiated in the initialization code, not in the handler. This allows AWS Lambda to reuse existing connections, for the duration of the container's lifetime.
- In general, you do not need to explicitly manage connections or implement connection pooling because AWS Lambda manages this for you.

A Lambda consumer for a DynamoDB stream doesn't guarantee exactly once delivery and may lead to occasional duplicates. Make sure your Lambda function code is idempotent to prevent unexpected issues from arising because of duplicate processing.

For more information, see [Best practices for working with AWS Lambda functions](#) in the *AWS Lambda Developer Guide*.

Using On-Demand backup and restore for DynamoDB

You can use the DynamoDB on-demand backup capability to create full backups of your tables for long-term retention, and archiving for regulatory compliance needs. You can back up and restore your table data anytime with a single click on the AWS Management Console or with a single API call. Backup and restore actions run with no impact on table performance or availability.

The following video will give you an introductory look at the backup and restore concept.

[Backup and restore](#)

There are two options available for creating and managing DynamoDB on-demand backups:

- AWS Backup service
- DynamoDB

With AWS Backup, you can configure backup policies and monitor activity for your AWS resources and on-premises workloads in one place. Using DynamoDB with AWS Backup, you can copy your on-demand backups across AWS accounts and Regions, add cost allocation tags to on-demand backups, and transition on-demand backups to cold storage for lower costs. To use these advanced features, you must [opt in](#) to AWS Backup. Opt-in choices apply to the specific account and AWS Region, so you might have to opt in to multiple Regions using the same account. For more information, see the [AWS Backup Developer Guide](#).

The on-demand backup and restore process scales without degrading the performance or availability of your applications. It uses a new and unique distributed technology that lets you complete backups in seconds regardless of table size. You can create backups that are consistent within seconds across thousands of partitions without worrying about schedules or long-running backup processes. All on-demand backups are cataloged, discoverable, and retained until they are explicitly deleted.

In addition, on-demand backup and restore operations don't affect performance or API latencies. Backups are preserved regardless of table deletion. For more information, see [Using DynamoDB backup and restore](#).

DynamoDB on-demand backups are available at no additional cost beyond the normal pricing that's associated with backup storage size. DynamoDB on-demand backups cannot be copied to a different account or Region. To create backup copies across AWS accounts and Regions and for other advanced features, you should use AWS Backup. If you use AWS Backup features you will be billed for them by AWS Backup. For more information about AWS Region availability and pricing, see [Amazon DynamoDB pricing](#).

Topics

- [Using AWS Backup with DynamoDB](#)
- [Using DynamoDB backup and restore](#)

Using AWS Backup with DynamoDB

Amazon DynamoDB can help you meet regulatory compliance and business continuity requirements through enhanced backup features in AWS Backup. AWS Backup is a fully managed data protection service that makes it easy to centralize and automate backups across AWS services, in the cloud, and on premises. Using this service, you can configure backup policies and monitor activity for your AWS resources in one place. To use AWS Backup, you must affirmatively [opt-in](#). Opt-in choices apply to the specific account and AWS Region, so you might have to opt in to multiple Regions using the same account. For more information, see the [AWS Backup Developer Guide](#).

Amazon DynamoDB is natively integrated with AWS Backup. You can use AWS Backup to schedule, copy, tag and life cycle your DynamoDB on-demand backups automatically. You can continue to view and restore these backups from the DynamoDB console. You can use the DynamoDB console, API, and AWS Command Line Interface (AWS CLI) to enable automatic backups for your DynamoDB tables.

 **Note**

Any backups made through DynamoDB will remain unchanged. You will still be able to create backups through the current DynamoDB workflow.

Enhanced backup features available through AWS Backup include:

Scheduled backups - You can set up regularly scheduled backups of your DynamoDB tables using backup plans.

Cross-account and cross-Region copying - You can automatically copy your backups to another backup vault in a different AWS Region or account, which allows you to support your data protection requirements.

Cold storage tiering - You can configure your backups to implement life cycle rules to delete or transition backups to colder storage. This can help you optimize your backup costs.

Tags - You can automatically tag your backups for billing and cost allocation purposes.

Encryption – DynamoDB on-demand backups managed through AWS Backup are now stored in the AWS Backup vault. This allows you to encrypt and secure your backups by using an AWS KMS key that is independent from your DynamoDB table encryption key.

Audit backups – You can use AWS Backup Audit Manager to audit the compliance of your AWS Backup policies and to find backup activity and resources that are not yet compliant with the controls that you defined. You can also use it to automatically generate an audit trail of daily and on-demand reports for your backup governance purposes.

Secure backups using the WORM model – You can use AWS Backup Vault Lock to enable a write-once-read-many (WORM) setting for your backups. With AWS Backup Vault Lock, you can add an additional layer of defense that protects backups from inadvertent or malicious delete operations, changes to backup retention periods, and updates to lifecycle settings. To learn more, see [AWS Backup Vault Lock](#).

These enhanced backup features are available in all AWS Regions. To learn more about these features, see the [AWS Backup Developer Guide](#).

Topics

- [Backing up and restoring DynamoDB tables with AWS Backup: How it works](#)
- [Creating backups of DynamoDB tables with AWS Backup](#)
- [Copying a backup of a DynamoDB table with AWS Backup](#)
- [Restoring a backup of a DynamoDB table from AWS Backup](#)
- [Deleting a backup of a DynamoDB table with AWS Backup](#)
- [Usage notes](#)

Backing up and restoring DynamoDB tables with AWS Backup: How it works

You can use the on-demand backup feature to create full backups of your Amazon DynamoDB tables. This section provides an overview of what happens during the backup and restore process.

Backups

When you create an on-demand backup with AWS Backup, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot.

Each time you create an on-demand backup, the entire table data is backed up. There is no limit to the number of on-demand backups that can be taken.

Note

Unlike DynamoDB Backups, backups made with AWS Backup are not instantaneous.

While a backup is in progress, you can't do the following:

- Pause or cancel the backup operation.
- Delete the source table of the backup.
- Disable backups on a table if a backup for that table is in progress.

AWS Backup provides automated backup schedules, retention management, and lifecycle management. This removes the need for custom scripts and manual processes. AWS Backup runs the backups and deletes them when they expire. For more information, see the [AWS Backup Developer Guide](#).

If you're using the console, any backups created using AWS Backup are listed on the **Backups** tab with the **Backup type** set to AWS_BACKUP.

Note

You can't delete backups marked with a **Backup type** of AWS_BACKUP using the DynamoDB console. To manage these backups, use the AWS Backup console.

To learn how to perform a backup, see [Backing up a DynamoDB table](#).

Restores

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore from your DynamoDB backup, or you can configure the destination table settings. When you do a restore, you can change the following table settings:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity

- Encryption settings

 **Important**

When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units that the source table had when the backup was requested. The restore process also restores the local secondary indexes and the global secondary indexes.

You can copy a backup of your DynamoDB table data to a different AWS Region and then restore it in that new Region. You can copy and then restore backups between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you copy from the source Region and the data you restore to a new table in the destination Region.

AWS Backup will restore the tables with all the original indexes.

You must manually set up the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Deletion protection settings
- Point in time recovery (PITR) settings

You can only restore the entire table data to a new table from a backup. You can write to the restored table only after it becomes active.

 **Note**

AWS Backup restores are nondestructive. You can't overwrite an existing table during a restore operation.

Service metrics show that 95 percent of customers' table restores complete in less than one hour. However, restore times are directly related to the configuration of your tables (such as the size of your tables and the number of underlying partitions) and other related variables. A best practice when planning for disaster recovery is to regularly document average restore completion times and establish how these times affect your overall Recovery Time Objective.

To learn how to perform a restore, see [Restoring a DynamoDB table from a backup](#).

You can use IAM policies for access control. For more information, see [Using IAM with DynamoDB backup and restore](#).

All backup and restore console and API actions are captured and recorded in AWS CloudTrail for logging, continuous monitoring, and auditing.

Creating backups of DynamoDB tables with AWS Backup

This section describes how to turn on AWS Backup to create on-demand and scheduled backups from your DynamoDB tables.

Topics

- [Turning on AWS Backup features](#)
- [On-demand backups](#)
- [Scheduled backups](#)

Turning on AWS Backup features

You must turn on AWS Backup to use it with DynamoDB.

To turn on AWS Backup, go through the following steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the Backup Settings window, choose **Turn on**.
4. A confirmation screen will appear. Choose **Turn on features**.

AWS Backup features are now available for your DynamoDB tables.

If you choose to turn off AWS Backup features after they've been turned on, follow these steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the Backup Settings window, choose **Turn off**.
4. A confirmation screen will appear. Choose **Turn off features**.

If you can't turn the AWS Backup features on or off, your AWS admin may need to perform those actions.

On-demand backups

To create an on-demand backup of a DynamoDB table, follow these steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. Choose **Create backup**.
4. From the dropdown menu that appears, choose **Create an on-demand backup**.
5. To create a backup managed by AWS Backup with warm storage and other basic features, choose **Default Settings**. To create a backup that can be transitioned to cold storage, or to create a backup with DynamoDB features instead of AWS Backup, choose **Customize settings**.

If you want to create this backup with previous DynamoDB features instead, choose **Customize settings** and then choose **Backup with DynamoDB**.

6. When you have completed the settings, choose **Create backup**.

Scheduled backups

To schedule a backup, follow these steps.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. From the dropdown menu that appears, choose **Schedule backups with AWS Backup**.
4. You will be taken to AWS Backup to create a backup plan.

Copying a backup of a DynamoDB table with AWS Backup

You can make a copy of a current backup. You can copy backups to multiple AWS accounts or AWS Regions on demand or automatically as part of a scheduled backup plan. You can also automate a sequence of cross-account and cross-Region copies for Amazon DynamoDB Encryption Client.

Cross-Region replication is especially valuable if you have business continuity or compliance requirements to store backups a minimum distance away from your production data.

Cross-account backups are useful for securely copying your backups to one or more AWS accounts in your organization for operational or security reasons. If your original backup is inadvertently deleted, you can copy the backup from its destination account to its source account, and then start the restore. Before you can do this, you must have two accounts that belong to the same organization in the Organizations service.

Copies inherit the source backup's configuration unless you specify otherwise, with one exception: if you specify that your new copy "Never" expire. With this setting, the new copy still inherits its source expiration date. If you want your new backup copy to be permanent, either set your source backups to never expire, or specify your new copy to expire 100 years after its creation.

 **Note**

If you're copying to another account, you must first have permission from that account.

To copy a backup, do the following:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. Select the check box next to the backup you want to copy.
 - If the backup you want to copy is grayed out, you must enable [advanced features with AWS Backup](#). Then create a new backup. You can now copy this new backup to other Regions and accounts, and copy any other new backups going forward.
4. Choose **Copy**.
5. If you want to copy the backup to another account or Region, select the check box next to **Copy the recovery point to another destination**. Then select whether you will to copy to another Region in your account, or to a different account in a different Region.

Note

To restore a backup to another Region or account, you must first copy the backup to that Region or account.

6. Select the desired vault the file will be copied into. You can also create a new backup vault if desired.
7. Choose **Copy backup**.

Restoring a backup of a DynamoDB table from AWS Backup

This section describes how to restore a backup of a DynamoDB table from AWS Backup.

Topics

- [Restoring a DynamoDB table from AWS Backup](#)
- [Restoring a DynamoDB table to another Region or account](#)

Restoring a DynamoDB table from AWS Backup

To restore your DynamoDB tables from AWS Backup, follow these steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Backups** tab.
4. Select the check box next to the previous backup that you want to restore from.
5. Choose **Restore**. You will be taken to the **Restore table from backup** screen.
6. Enter the name for the newly restored table, the encryption that this new table will have, the key you want the restore to be encrypted with, and other options.
7. When you're finished, choose **Restore**.

Restoring a DynamoDB table to another Region or account

To restore a DynamoDB table to another Region or account, you will first need to copy the backup to that new Region or account. In order to copy to another account, that account must first grant

you permission. After you have copied your DynamoDB backup to the new Region or account, it can be restored with the process in the previous section.

Deleting a backup of a DynamoDB table with AWS Backup

This section describes how to delete a backup of a DynamoDB table with AWS Backup.

A DynamoDB backup created through AWS Backup features is stored in an AWS Backup vault.

In order to delete this kind of backup, do the following:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. On the screen that follows, choose **Continue to AWS Backup**.

You will be taken to the AWS Backup console. To learn more on how to delete backups on the AWS Backup console, see [Deleting backups](#)

Usage notes

This section describes the technical differences between on-demand backups managed by AWS Backup and DynamoDB.

AWS Backup has some different workflows and behaviors than DynamoDB. These include:

Encryption - Backups created with the AWS Backup plan are stored in an encrypted vault with a key that is managed by the AWS Backup service. The vault has access control policies for additional security.

Backup ARN - The backup files created by AWS Backup will now have an AWS Backup ARN, which could impact the user permission model. Backup resource names (ARNs) will change from `arn:aws:dynamodb` to `arn:aws:backup`.

Deleting backups - Backups that are created with AWS Backup can only be deleted from the AWS Backup vault. You will not be able to delete AWS Backup files from the DynamoDB console.

Backup process - Unlike DynamoDB backups, backups made with AWS Backup are not instantaneous.

Billing - Backups of DynamoDB tables with AWS Backup features are billed from AWS Backup.

IAM roles - If you're managing access through IAM roles, you will also need to configure a new IAM role with these new permissions:

```
"dynamodb:StartAwsBackupJob",
"dynamodb:RestoreTableFromAwsBackup"
```

dynamodb:StartAwsBackupJob is needed for a successful backup with AWS Backup features, and dynamodb:RestoreTableFromAwsBackup is needed to restore from a backup made with AWS Backup features.

To see these permissions in a complete IAM policy, see Example 8 in [Using IAM](#).

Using DynamoDB backup and restore

Amazon DynamoDB supports stand-alone on-demand backup and restores features. Those features are available to you independent of whether you use AWS Backup.

You can use the DynamoDB on-demand backup capability to create full backups of your tables for long-term retention and archival for regulatory compliance needs. You can back up and restore your table data anytime with a single click on the AWS Management Console or with a single API call. Backup and restore actions run with zero impact on table performance or availability.

You can create table backups using the console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. For more information, see [Backing up a DynamoDB table](#).

For information about restoring a table from a backup, see [Restoring a DynamoDB table from a backup](#).

Backing up and restoring DynamoDB tables with DynamoDB: How it works

You can use the DynamoDB on-demand backup feature to create full backups of your Amazon DynamoDB tables. This feature is available independently from AWS backup. This section provides an overview of what happens during the DynamoDB backup and restore process.

Backups

When you create an on-demand backup with DynamoDB, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the

request to the last full table snapshot. DynamoDB backup requests are processed instantaneously and become available for restore within minutes.

 **Note**

Each time you create an on-demand backup, the entire table data is backed up. There is no limit to the number of on-demand backups that can be taken.

All backups in DynamoDB work without consuming any provisioned throughput on the table.

DynamoDB backups do not guarantee causal consistency across items; however, the skew between updates in a backup is usually much less than a second.

While a backup is in progress, you can't do the following:

- Pause or cancel the backup operation.
- Delete the source table of the backup.
- Disable backups on a table if a backup for that table is in progress.

If you don't want to create scheduling scripts and cleanup jobs, you can use AWS Backup to create backup plans with schedules and retention policies for your DynamoDB tables. AWS Backup runs the backups and deletes them when they expire. For more information, see the [AWS Backup Developer Guide](#).

In addition to AWS Backup, you can schedule periodic or future backups by using AWS Lambda functions. For more information, see the blog post [A serverless solution to schedule your Amazon DynamoDB On-Demand backup](#).

If you're using the console, any backups created using AWS Backup are listed on the **Backups** tab with the **Backup type** set to AWS.

 **Note**

You can't delete backups marked with a **Backup type** of AWS using the DynamoDB console. To manage these backups, use the AWS Backup console.

To learn how to perform a backup, see [Backing up a DynamoDB table](#).

Restores

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore from your DynamoDB backup, or you can configure the destination table settings. When you do a restore, you can change the following table settings:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity
- Encryption settings

Important

When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units as the source table, as recorded at the time the backup was requested. The restore process also restores the local secondary indexes and the global secondary indexes.

You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the backup resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data that you transfer out of the source Region and for restoring to a new table in the destination Region.

Restores can be faster and more cost-efficient if you choose to exclude some or all secondary indexes from being created on the new restored table.

You must manually set up the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings

- Deletion protection settings
- Point in time recovery (PITR) settings

You can only restore the entire table data to a new table from a backup. You can write to the restored table only after it becomes active.

 **Note**

You can't overwrite an existing table during a restore operation.

Service metrics show that 95 percent of customers' table restores complete in less than one hour. However, restore times are directly related to the configuration of your tables (such as the size of your tables and the number of underlying partitions) and other related variables. A best practice when planning for disaster recovery is to regularly document average restore completion times and establish how these times affect your overall Recovery Time Objective.

To learn how to perform a restore, see [Restoring a DynamoDB table from a backup](#).

You can use IAM policies for access control. For more information, see [Using IAM with DynamoDB backup and restore](#).

All backup and restore console and API actions are captured and recorded in AWS CloudTrail for logging, continuous monitoring, and auditing.

Backing up a DynamoDB table

This section describes how to use the Amazon DynamoDB console or the AWS Command Line Interface to back up a table.

Topics

- [Creating a table backup \(console\)](#)
- [Creating a table backup \(AWS CLI\)](#)

Creating a table backup (console)

Follow these steps to create a backup named MusicBackup for an existing Music table using the AWS Management Console.

To create a table backup

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. You can create a backup by doing one of the following:
 - On the **Backups** tab of the Music table, choose **Create backup**.
 - In the navigation pane on the left side of the console, choose **Backups**. Then choose **Create backup**.
3. Make sure that Music is the table name, and enter **MusicBackup** for the backup name. Then, choose **Create** to create the backup.

Create backup

Backup settings Info

Source table
Music

Backup name
This will be used to identify your backup.
 Between 3 and 255 characters in length. Only A-Z, a-z, 0-9, underscore characters, hyphens, and periods are allowed.

Cancel **Create backup**

Note

If you create backups using the **Backups** section in the navigation pane, the table isn't preselected for you. You have to manually choose the source table name for the backup.

While the backup is being created, the backup status is set to **Creating**. After the backup is complete, the backup status changes to **Available**.

	Name	Status	Creation Date	ARN
	MusicBackup	Available	August 23...	arn:aws:dynamodb:us-w...

Creating a table backup (AWS CLI)

Follow these steps to create a backup for an existing table Music using the AWS CLI.

To create a table backup

- Create a backup with the name MusicBackup for the Music table.

```
aws dynamodb create-backup --table-name Music \
--backup-name MusicBackup
```

While the backup is being created, the backup status is set to CREATING.

```
{
    "BackupDetails": {
        "BackupName": "MusicBackup",
        "BackupArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc",
        "BackupStatus": "CREATING",
        "BackupCreationDateTime": 1489602797.149
    }
}
```

After the backup is complete, its BackupStatus should change to AVAILABLE. To confirm this, use the describe-backup command. You can get the input value of backup-arn from the output of the previous step or by using the list-backups command.

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-
b308cd7d
```

To keep track of your backups, you can use the `list-backups` command. It lists all your backups that are in CREATING or AVAILABLE status.

```
aws dynamodb list-backups
```

The `list-backups` command and the `describe-backup` command are useful to check information about the source table of the backup.

Restoring a DynamoDB table from a backup

This section describes how to restore a table from a backup using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Note

If you want to use the AWS CLI, you must configure it first. For more information, see [Accessing DynamoDB](#).

Topics

- [Restoring a table from a backup \(console\)](#)
- [Restoring a table from a backup \(AWS CLI\)](#)

Restoring a table from a backup (console)

The following procedure shows how to restore the Music table by using the MusicBackup file that is created in the [Backing up a DynamoDB table](#) tutorial.

Note

This procedure assumes that the Music table no longer exists before restoring it using the MusicBackup file.

To restore a table from a backup

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.

2. In the navigation pane on the left side of the console, choose **Backups**.

3. In the list of backups, choose MusicBackup.

On-demand backups (1/1)					Info	C	Restore	Delete	Create backup	
<input type="text"/> Find backups by ARN or name						<	1	>		
<input checked="" type="checkbox"/>	Name	Table	Status	Creation t...	ARN					
<input checked="" type="checkbox"/>	MusicBackup	Music	Available	August 23, 2...	arn:aws:dynamodb:us-w...					

4. Choose **Restore**.

5. Enter **Music** as the new table name. Confirm the backup name and other backup details. Then choose **Restore table** to start the restore process.

Note

You can restore the table to the same AWS Region or to a different Region from where the backup resides. You can also exclude secondary indexes from being created on the new restored table. In addition, you can specify a different encryption mode.

Tables restored from backups are always created using the DynamoDB Standard table class.

Restore table from backup Info

Restoring a table from a backup will restore it as a new table.

Restore settings

Name of restored table

This name will identify your restored table.

Between 3 and 255 characters in length. Only A–Z, a–z, 0–9, underscore characters, hyphens, and periods allowed.

Secondary indexes

Restore the entire table

Your restored table will include all local and global secondary indexes.

Restore the table without secondary indexes

Your restored table will exclude all local and global secondary indexes. Restoring this way can be faster and more cost efficient.

Destination AWS Region

Same Region (Oregon)

Restore the table to the same Region as the original table.

Cross-Region

Restore the table to a different Region for greater redundancy but with higher data transfer costs.

▼ Encryption at rest - optional

All user data stored in Amazon DynamoDB is fully encrypted at rest. By default, Amazon DynamoDB manages the encryption key, and you are not charged any fee for using it.

Encryption key management Info

Owned by Amazon DynamoDB

The key is owned and managed by DynamoDB. You are not charged an additional fee for using this customer master key (CMK).

AWS managed CMK

The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.

Stored in your account, and owned and managed by you

Choose a key that is owned and managed by you, and stored in AWS KMS.

i The time it takes to restore a table from a backup can vary and is based on multiple variables. After your table is restored from the backup, you might need to reapply configuration settings. [Learn more](#) ↗

The table that is being restored is shown with the status **Creating**. After the restore process is finished, the status of the Music table changes to **Active**.

Restoring a table from a backup (AWS CLI)

Follow these steps to use the AWS CLI to restore the Music table using the MusicBackup that is created in the [Backing up a DynamoDB table](#) tutorial.

To restore a table from a backup

1. Confirm the backup that you want to restore by using the `list-backups` command. This example uses MusicBackup.

```
aws dynamodb list-backups
```

To get additional details for the backup, use the `describe-backup` command. You can get the input `backup-arn` from the previous step.

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d
```

2. Restore the table from the backup. In this case, the MusicBackup restores the Music table to the same AWS Region.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d
```

3. Restore the table from the backup with custom table settings. In this case, the MusicBackup restores the Music table and specifies an encryption mode for the restored table.

Note

The `sse-specification-override` parameter takes the same values as the `sse-specification-override` parameter used in the `CreateTable` command. To learn more, see [Managing encrypted tables in DynamoDB](#).

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581080476474-e177ebe2 \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can restore the table to a different AWS Region from where the backup resides.

 **Note**

- The `sse-specification-override` parameter is mandatory for cross-Region restores but optional for restores in the same Region as the source table.
- When performing a cross-Region restore from the command line, you must set the default AWS Region to the desired destination Region. To learn more, see [Command line options](#) in the *AWS Command Line Interface User Guide*.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581080476474-e177ebe2 \
--sse-specification-override Enabled=true,SSEType=KMS
```

You can override the billing mode and the provisioned throughput for the restored table.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d \
--billing-mode-override PAY_PER_REQUEST
```

You can exclude some or all secondary indexes from being created on the restored table.

Note

Restores can be faster and more cost-efficient if you exclude some or all secondary indexes from being created on the restored table.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581081403719-db9c1f91 \
--global-secondary-index-override '[]' \
--sse-specification-override Enabled=true,SSEType=KMS
```

Note

The secondary indexes provided should match existing indexes. You cannot create new indexes at the time of restore.

You can use a combination of different overrides. For example, you can use a single global secondary index and change provisioned throughput at the same time, as follows.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:eu-west-1:123456789012:table/Music/
backup/01581082594992-303b6239 \
--billing-mode-override PROVISIONED \
--provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100 \
--global-secondary-index-override IndexName=singers-
index,KeySchema=[{"AttributeName=SingerName,KeyType=HASH"}],Projection="{ProjectionType=KEYS_"
 \
--sse-specification-override Enabled=true,SSEType=KMS
```

To verify the restore, use the `describe-table` command to describe the `Music` table.

```
aws dynamodb describe-table --table-name Music
```

The table that is being restored from the backup is shown with the status **Creating**. After the restore process is finished, the status of the Music table changes to **Active**.

Important

While a restore is in progress, don't modify or delete your IAM role policy; otherwise, unexpected behavior can result. For example, suppose that you removed write permissions for a table while that table is being restored. In this case, the underlying `RestoreTableFromBackup` operation would not be able to write any of the restored data to the table.

After the restore operation is complete, you can modify or delete your IAM role policy. IAM policies involving [source IP restrictions](#) for accessing the target restore table should have the [`aws:ViaAWSService`](#) key set to `false` to ensure that the restrictions apply only to requests made directly by a principal. Otherwise, the restore will be canceled.

If your backup is encrypted with an AWS managed key or a customer managed key, don't disable or delete the key while a restore is in progress, or the restore will fail.

After the restore operation is complete, you can change the encryption key for the restored table and disable or delete the old key.

Deleting a DynamoDB table backup

This section describes how to use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to delete an Amazon DynamoDB table backup.

Note

If you want to use the AWS CLI, you have to configure it first. For more information, see [Using the AWS CLI](#).

Topics

- [Deleting a table backup \(console\)](#)
- [Deleting a table backup \(AWS CLI\)](#)

Deleting a table backup (console)

The following procedure shows how to use the console to delete the MusicBackup that is created in the [Backing up a DynamoDB table](#) tutorial.

To delete a backup

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the list of backups, choose MusicBackup.

The screenshot shows the 'On-demand backups' section of the AWS DynamoDB console. It displays one backup entry: 'MusicBackup' for the 'Music' table, which is currently 'Available'. The table has columns for Name, Table, Status, Creation time, and ARN. Buttons for 'Info', 'Restore', 'Delete', and 'Create backup' are visible at the top right. A search bar and pagination controls are also present.

Name	Table	Status	Creation t...	ARN
MusicBackup	Music	Available	August 23, 2...	arn:aws:dynamodb:us-w...

4. Choose **Delete**. Confirm that you want to delete the backup by typing **delete** and clicking **Delete**.

Deleting a table backup (AWS CLI)

The following example deletes a backup for an existing table Music table using the AWS CLI.

```
aws dynamodb delete-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc
```

Using IAM with DynamoDB backup and restore

You can use AWS Identity and Access Management (IAM) to restrict Amazon DynamoDB backup and restore actions for some resources. The `CreateBackup` and `RestoreTableFromBackup` APIs operate on a per-table basis.

For more information about using IAM policies in DynamoDB, see [Identity-based policies for DynamoDB](#).

The following are examples of IAM policies that you can use to configure specific backup and restore functionality in DynamoDB.

Example 1: Allow the CreateBackup and RestoreTableFromBackup actions

The following IAM policy grants permissions to allow the CreateBackup and RestoreTableFromBackup DynamoDB actions on all tables:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:CreateBackup",  
                "dynamodb:RestoreTableFromBackup",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:GetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Important

DynamoDB RestoreTableFromBackup permissions are necessary on the source backup, and DynamoDB read and write permissions on the target table are necessary for restore functionality.

DynamoDB RestoreTableToPointInTime permissions are necessary on the source table, and DynamoDB read and write permissions on the target table are necessary for restore functionality.

Example 2: Allow CreateBackup and deny RestoreTableFromBackup

The following IAM policy grants permissions for the CreateBackup action and denies the RestoreTableFromBackup action:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb>CreateBackup"],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": ["dynamodb:RestoreTableFromBackup"],  
            "Resource": "*"  
        }  
    ]  
}
```

Example 3: Allow ListBackups and deny CreateBackup and RestoreTableFromBackup

The following IAM policy grants permissions for the `ListBackups` action and denies the `CreateBackup` and `RestoreTableFromBackup` actions:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb>ListBackups"],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": [  
                "dynamodb>CreateBackup",  
                "dynamodb:RestoreTableFromBackup"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example 4: Allow ListBackups and deny DeleteBackup

The following IAM policy grants permissions for the `ListBackups` action and denies the `DeleteBackup` action:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb>ListBackups"],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": ["dynamodb>DeleteBackup"],  
            "Resource": "*"  
        }  
    ]  
}
```

Example 5: Allow RestoreTableFromBackup and DescribeBackup for all resources and deny DeleteBackup for a specific backup

The following IAM policy grants permissions for the `RestoreTableFromBackup` and `DescribeBackup` actions and denies the `DeleteBackup` action for a specific backup resource:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb>DescribeBackup",  
                "dynamodb>RestoreTableFromBackup",  
            ],  
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/  
backup/01489173575360-b308cd7d"  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
        }
```

```
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb>DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:BatchWriteItem"
    ],
    "Resource": "*"
},
{
    "Effect": "Deny",
    "Action": [
        "dynamodb>DeleteBackup"
    ],
    "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d"
}
]
```

Important

DynamoDB RestoreTableFromBackup permissions are necessary on the source backup, and DynamoDB read and write permissions on the target table are necessary for restore functionality.

DynamoDB RestoreTableToPointInTime permissions are necessary on the source table, and DynamoDB read and write permissions on the target table are necessary for restore functionality.

Example 6: Allow CreateBackup for a specific table

The following IAM policy grants permissions for the CreateBackup action on the Movies table only:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb>CreateBackup",
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Movies"
        }
    ]
}
```

```
        "Action": ["dynamodb>CreateBackup"],
        "Resource": [
            "arn:aws:dynamodb:us-east-1:123456789012:table/Movies"
        ]
    }
]
```

Example 7: Allow ListBackups

The following IAM policy grants permissions for the `ListBackups` action:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb>ListBackups"],
            "Resource": "*"
        }
    ]
}
```

Important

You cannot grant permissions for the `ListBackups` action on a specific table.

Example 8: Allow access to AWS Backup features

You will need API permissions for the `StartAwsBackupJob` action for a successful backup with advanced features, and the `dynamodb:RestoreTableFromAwsBackup` action to successfully restore that backup.

The following IAM policy grants AWS Backup the permissions to trigger backups with advanced features and restores. Also note that if the tables are encrypted the policy will need access to the [AWS KMS key](#).

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
{  
    "Sid": "DescribeQueryScanBooksTable",  
    "Effect": "Allow",  
    "Action": [  
        "dynamodb:StartAwsBackupJob",  
        "dynamodb:DescribeTable",  
        "dynamodb:Query",  
        "dynamodb:Scan"  
    ],  
    "Resource": "arn:aws:dynamodb:us-west-2:account-id:table/Books"  
},  
{  
    "Sid": "AllowRestoreFromAwsBackup",  
    "Effect": "Allow",  
    "Action": ["dynamodb:RestoreTableFromAwsBackup"],  
    "Resource": "*"  
},  
]  
}
```

Example 9: Deny RestoreTableToPointInTime for a Specific Source Table

The following IAM policy denies permissions for the `RestoreTableToPointInTime` action for a specific source table:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Deny",  
            "Action": [  
                "dynamodb:RestoreTableToPointInTime"  
            ],  
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music"  
        }  
    ]  
}
```

Example 10: Deny RestoreTableFromBackup for all Backups for a Specific Source Table

The following IAM policy denies permissions for the `RestoreTableToPointInTime` action for all backups for a specific source table:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Deny",  
            "Action": [  
                "dynamodb:RestoreTableFromBackup"  
            ],  
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/*"  
        }  
    ]  
}
```

Point-in-time recovery for DynamoDB

You can create on-demand backups of your Amazon DynamoDB tables, or you can enable continuous backups using point-in-time recovery. For more information about on-demand backups, see [Using On-Demand backup and restore for DynamoDB](#).

Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point-in-time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. After you enable point-in-time recovery, you can restore to any point in time from five minutes before the current time until 35 days ago. DynamoDB maintains incremental backups of your table.

In addition, point-in-time operations don't affect performance or API latencies. For more information, see [Point-in-time recovery: How it works](#).

You can restore a DynamoDB table to a point in time using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. The point-in-time recovery process restores to a new table. For more information, see [Restoring a DynamoDB table to a point in time](#).

The following video will give you an introductory look at the backup and restore concept and talk more about point-in-time recovery.

[Backup and restore](#)

For more information about AWS Region availability and pricing, see [Amazon DynamoDB pricing](#).

Topics

- [Point-in-time recovery: How it works](#)
- [Before you begin using point-in-time recovery](#)
- [Restoring a DynamoDB table to a point in time](#)

Point-in-time recovery: How it works

Amazon DynamoDB point-in-time recovery (PITR) provides automatic backups of your DynamoDB table data. This section provides an overview of how the process works in DynamoDB.

Enabling point-in-time recovery

You can enable point-in-time recovery using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the DynamoDB API. When enabled, point-in-time recovery provides continuous backups until you explicitly turn it off. For more information, see [Restoring a DynamoDB table to a point in time](#).

After you enable point-in-time recovery, you can restore to any point in time within `EarliestRestorableDateTime` and `LatestRestorableDateTime`. `LatestRestorableDateTime` is typically five minutes before the current time.

 **Note**

The point-in-time recovery process always restores to a new table.

Restoring a table using point-in-time recovery

For `EarliestRestorableDateTime`, you can restore your table to any point in time during the last 35 days. The retention period is a fixed 35 days (5 calendar weeks) and can't be modified. Any number of users can run up to 50 concurrent restores (any type of restore) in a given account.

 **Important**

If you disable point-in-time recovery and later re-enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.

When you restore using point-in-time recovery, DynamoDB restores your table data to the state based on the selected date and time (day:hour:minute:second) to a new table.

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore using point-in-time recovery, or you can configure the destination table settings. You can change the following table settings on the restored table:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity
- Encryption settings

Important

When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units that the source table had when the backup was requested. For example, suppose that a table's provisioned throughput was recently lowered to 50 read capacity units and 50 write capacity units. You then restore the table's state to three weeks ago, at which time its provisioned throughput was set to 100 read capacity units and 100 write capacity units. In this case, DynamoDB restores your table data to that point in time with the provisioned throughput from that time (100 read capacity units and 100 write capacity units).

You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the source table resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you transfer out of the source Region and for restoring to a new table in the destination Region.

Note

Cross-Region restore is not supported if the source or destination Region is Asia Pacific (Hong Kong) or Middle East (Bahrain).

Restores can be faster and more cost-efficient if you exclude some or all indexes from being created on the restored table.

You must manually set the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Point-in-time recovery settings

The time it takes you to restore a table varies based on multiple factors. The point-in-time restore times are not always correlated directly to the size of the table. For more information, see [Restores](#).

Deleting a table with point-in-time recovery enabled

When you delete a table that has point-in-time recovery enabled, DynamoDB automatically creates a backup snapshot called a *system backup* and retains it for 35 days (at no additional cost). You can use the system backup to restore the deleted table to the state it was in just before deletion. All system backups follow a standard naming convention of *table-name\$DeletedTableBackup*.

Note

Once a table with point-in-time recovery enabled is deleted, you can use system restore to restore that table to a single point in time: the moment just before deletion. You do not have the capability to restore a deleted table to any other point in time in the past 35 days.

Before you begin using point-in-time recovery

Before you enable point-in-time recovery (PITR) on an Amazon DynamoDB table, consider the following:

- If you disable point-in-time recovery and later re-enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.

- You can enable point-in-time recovery on each local replica of a global table. When you restore the table, the backup restores to an independent table that is not part of the global table. If you are using [Global Tables version 2019.11.21 \(Current\)](#) of global tables, you can create a new global table from the restored table. For more information, see [Global tables: How it works](#).
- You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the source table resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you transfer out of the source Region and for restoring to a new table in the destination Region.
- AWS CloudTrail logs all console and API actions for point-in-time recovery to enable logging, continuous monitoring, and auditing. For more information, see [Logging DynamoDB operations by using AWS CloudTrail](#).

Restoring a DynamoDB table to a point in time

Amazon DynamoDB point-in-time recovery (PITR) provides continuous backups of your DynamoDB table data. You can restore a table to a point in time using the DynamoDB console or the AWS Command Line Interface (AWS CLI). The point-in-time recovery process restores to a new table.

If you want to use the AWS CLI, you must configure it first. For more information, see [Accessing DynamoDB](#).

Topics

- [Restoring a DynamoDB table to a point in time \(console\)](#)
- [Restoring a table to a point in time \(AWS CLI\)](#)

Restoring a DynamoDB table to a point in time (console)

The following example demonstrates how to use the DynamoDB console to restore an existing table named `Music` to a point in time.

Note

This procedure assumes that you have enabled point-in-time recovery. To enable it for the `Music` table, on the **Backups** tab, in the **Point-in-time recovery (PITR)** section, choose **Edit** and then check the box next to **Enable point-in-time-recovery**.

To restore a table to a point in time

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. In the list of tables, choose the Music table.
4. On the **Backups** tab of the Music table, in the **Point-in-time recovery (PITR)** section, choose **Restore**.
5. For the new table name, enter **MusicMinutesAgo**.

 **Note**

You can restore the table to the same AWS Region or to a different Region from where the source table resides. You can also exclude secondary indexes from being created on the restored table. In addition, you can specify a different encryption mode.

6. To confirm the restorable time, set the restore date and time to **Earliest**. Then choose **Restore** to start the restore process.

The table that is being restored is shown with the status **Restoring**. After the restore process is finished, the status of the MusicMinutesAgo table changes to **Active**.

Restoring a table to a point in time (AWS CLI)

The following procedure shows how to use the AWS CLI to restore an existing table named Music to a point in time.

 **Note**

This procedure assumes that you have enabled point-in-time recovery. To enable it for the Music table, run the following command.

```
aws dynamodb update-continuous-backups \
    --table-name Music \
    --point-in-time-recovery-specification PointInTimeRecoveryEnabled=True
```

To restore a table to a point in time

1. Confirm that point-in-time recovery is enabled for the Music table by using the describe-continuous-backups command.

```
aws dynamodb describe-continuous-backups \
--table-name Music
```

Continuous backups (automatically enabled on table creation) and point-in-time recovery are enabled.

```
{
    "ContinuousBackupsDescription": {
        "PointInTimeRecoveryDescription": {
            "PointInTimeRecoveryStatus": "ENABLED",
            "EarliestRestorableDateTime": 1519257118.0,
            "LatestRestorableDateTime": 1520018653.01
        },
        "ContinuousBackupsStatus": "ENABLED"
    }
}
```

2. Restore the table to a point in time. In this case, the Music table is restored to the LatestRestorableDateTime (~5 minutes ago) to the same AWS Region.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time
```

Note

You can also restore to a specific point in time. To do this, run the command using the --restore-date-time argument, and specify a timestamp. You can specify any point in time during the last 35 days. For example, the following command restores the table to the EarliestRestorableDateTime.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicEarliestRestorableDateTime \
```

```
--no-use-latest-restorable-time \
--restore-date-time 1519257118.0
```

Specifying the `--no-use-latest-restorable-time` argument is optional when restoring to a specific point in time.

3. Restore the table to a point in time with custom table settings. In this case, the `Music` table is restored to the `LatestRestorableDateTime` (~5 minutes ago).

You can specify a different encryption mode for the restored table, as follows.

 **Note**

The `sse-specification-override` parameter takes the same values as the `sse-specification-override` parameter used in the `CreateTable` command. To learn more, see [Managing encrypted tables in DynamoDB](#).

```
aws dynamodb restore-table-to-point-in-time \
  --source-table-name Music \
  --target-table-name MusicMinutesAgo \
  --use-latest-restorable-time \
  --sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can restore the table to a different AWS Region from where the source table resides.

 **Note**

- The `sse-specification-override` parameter is mandatory for cross-Region restores but optional for restores to the same Region as the source table.
- The `source-table-arn` parameter must be provided for cross-Region restores.
- When performing a cross-Region restore from the command line, you must set the default AWS Region to the desired destination Region. To learn more, see [Command line options](#) in the *AWS Command Line Interface User Guide*.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can override the billing mode and the provisioned throughput for the restored table.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time \
--billing-mode-override PAY_PER_REQUEST
```

You can exclude some or all secondary indexes from being created on the restored table.

 **Note**

Restores can be faster and more cost-efficient if you exclude some or all secondary indexes from being created on the new restored table.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time \
--global-secondary-index-override '[]'
```

You can use a combination of different overrides. For example, you can use a single global secondary index and change provisioned throughput at the same time, as follows.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
--billing-mode-override PROVISIONED \
--provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100 \
\
```

```
--global-secondary-index-override IndexName=singers-
index,KeySchema=[{"AttributeName=SingerName,KeyType=HASH"}],Projection="{ProjectionType=KEY
 \
--sse-specification-override Enabled=true,SSEType=KMS \
--use-latest-restorable-time
```

To verify the restore, use the `describe-table` command to describe the `MusicEarliestRestorableDateTime` table.

```
aws dynamodb describe-table --table-name MusicEarliestRestorableDateTime
```

The table that is being restored is shown with the status **Creating** and restore in progress as **true**. After the restore process is finished, the status of the `MusicEarliestRestorableDateTime` table changes to **Active**.

Important

While a restore is in progress, don't modify or delete the AWS Identity and Access Management (IAM) policies that grant the IAM entity (for example, user, group, or role) permission to perform the restore. Otherwise, unexpected behavior can result. For example, suppose that you remove write permissions for a table while that table is being restored. In this case, the underlying `RestoreTableToPointInTime` operation can't write any of the restored data to the table. IAM policies involving source IP restrictions for accessing the target restore table can similarly cause issues.

You can modify or delete permissions only after the restore operation is completed.

In-memory acceleration with DynamoDB Accelerator (DAX)

Amazon DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For these use cases, DynamoDB Accelerator (DAX) delivers fast response times for accessing eventually consistent data.

DAX is a DynamoDB-compatible caching service that enables you to benefit from fast in-memory performance for demanding applications. DAX addresses three core scenarios:

1. As an in-memory cache, DAX reduces the response times of eventually consistent read workloads by an order of magnitude from single-digit milliseconds to microseconds.
2. DAX reduces operational and application complexity by providing a managed service that is API-compatible with DynamoDB. Therefore, it requires only minimal functional changes to use with an existing application.
3. For read-heavy or bursty workloads, DAX provides increased throughput and potential operational cost savings by reducing the need to overprovision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

DAX supports server-side encryption. With encryption at rest, the data persisted by DAX on disk will be encrypted. DAX writes data to disk as part of propagating changes from the primary node to read replicas. For more information, see [DAX encryption at rest](#).

DAX also supports encryption in transit, ensuring that all requests and responses between your application and the cluster are encrypted by transport level security (TLS), and connections to the cluster can be authenticated by verification of a cluster x509 certificate. For more information, see [DAX encryption in transit](#).

Topics

- [Use cases for DAX](#)
- [DAX usage notes](#)
- [DAX: How it works](#)
- [DAX cluster components](#)
- [Creating a DAX cluster](#)

- [DAX and DynamoDB consistency models](#)
- [Developing with the DynamoDB Accelerator \(DAX\) client](#)
- [Managing DAX clusters](#)
- [Monitoring DAX](#)
- [DAX T3/T2 burstable instances](#)
- [DAX access control](#)
- [DAX encryption at rest](#)
- [DAX encryption in transit](#)
- [Using service-linked IAM roles for DAX](#)
- [Accessing DAX across AWS accounts](#)
- [DAX cluster sizing guide](#)
- [DAX API reference](#)

Use cases for DAX

DAX provides access to eventually consistent data from DynamoDB tables, with microsecond latency. A Multi-AZ DAX cluster can serve millions of requests per second.

DAX is ideal for the following types of applications:

- Applications that require the fastest possible response time for reads. Some examples include real-time bidding, social gaming, and trading applications. DAX delivers fast, in-memory read performance for these use cases.
- Applications that read a small number of items more frequently than others. For example, consider an ecommerce system that has a one-day sale on a popular product. During the sale, demand for that product (and its data in DynamoDB) would sharply increase, compared to all of the other products. To mitigate the impacts of a "hot" key and a non-uniform traffic distribution, you could offload the read activity to a DAX cache until the one-day sale is over.
- Applications that are read-intensive, but are also cost-sensitive. With DynamoDB, you provision the number of reads per second that your application requires. If read activity increases, you can increase your tables' provisioned read throughput (at an additional cost). Or, you can offload the activity from your application to a DAX cluster, and reduce the number of read capacity units that you need to purchase otherwise.

- Applications that require repeated reads against a large set of data. Such an application could potentially divert database resources from other applications. For example, a long-running analysis of regional weather data could temporarily consume all the read capacity in a DynamoDB table. This situation would negatively impact other applications that need to access the same data. With DAX, the weather analysis could be performed against cached data instead.

DAX is *not* ideal for the following types of applications:

- Applications that require strongly consistent reads (or that cannot tolerate eventually consistent reads).
- Applications that do not require microsecond response times for reads, or that do not need to offload repeated read activity from underlying tables.
- Applications that are write-intensive, or that do not perform much read activity.
- Applications that are already using a different caching solution with DynamoDB, and are using their own client-side logic for working with that caching solution.

DAX usage notes

- For a list of AWS Regions where DAX is available, see [Amazon DynamoDB pricing](#).
- DAX supports applications written in Go, Java, Node.js, Python, and .NET, using AWS-provided clients for those programming languages.
- DAX is only available for the EC2-VPC platform.
- The DAX cluster service role policy must allow the dynamodb:DescribeTable action in order to maintain metadata about the DynamoDB table.
- DAX clusters maintain metadata about the attribute names of items they store. That metadata is maintained indefinitely (even after the item has expired or been evicted from the cache). Applications that use an unbounded number of attribute names can, over time, cause memory exhaustion in the DAX cluster. This limitation applies only to top-level attribute names, not nested attribute names. Examples of problematic top-level attribute names include timestamps, UUIDs, and session IDs.

This limitation applies only to attribute names, not their values. Items like the following are not a problem.

{

```
"Id": 123,  
"Title": "Bicycle 123",  
"CreationDate": "2017-10-24T01:02:03+00:00"  
}
```

But items like the following are a problem if there are enough of them and they each have a different timestamp.

```
{  
    "Id": 123,  
    "Title": "Bicycle 123",  
    "2017-10-24T01:02:03+00:00": "created"  
}
```

DAX: How it works

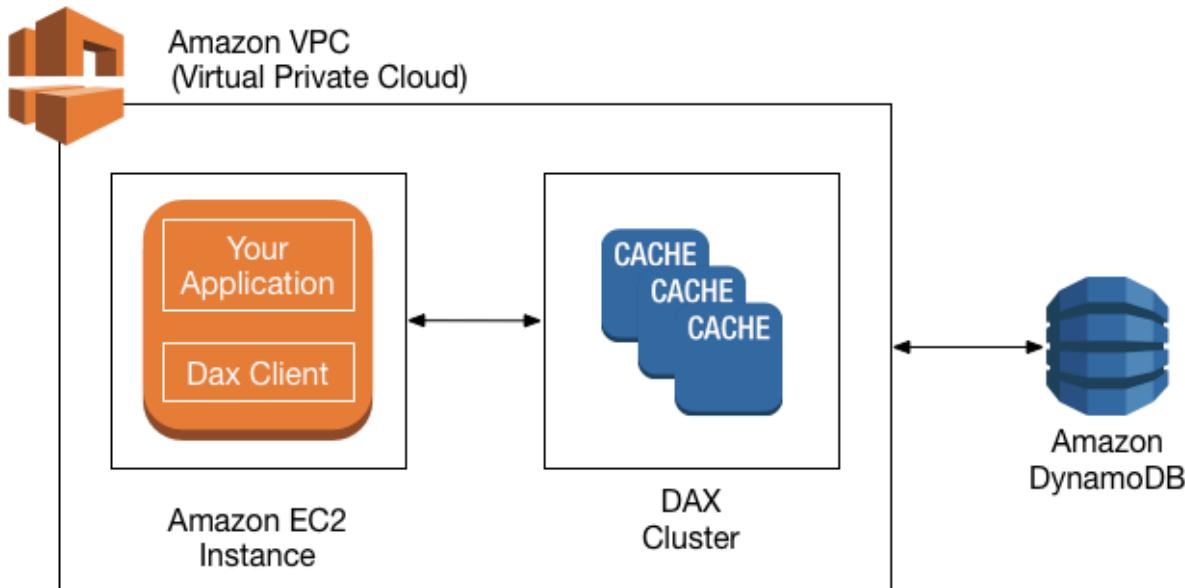
Amazon DynamoDB Accelerator (DAX) is designed to run within an Amazon Virtual Private Cloud (Amazon VPC) environment. The Amazon VPC service defines a virtual network that closely resembles a traditional data center. With a VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security settings. You can launch a DAX cluster in your virtual network and control access to the cluster by using Amazon VPC security groups.

Note

If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. The VPC is ready for you to use immediately—without having to perform any additional configuration steps.

For more information, see [Default VPC and default subnets](#) in the *Amazon VPC User Guide*.

The following diagram shows a high-level overview of DAX.



To create a DAX cluster, you use the AWS Management Console. Unless you specify otherwise, your DAX cluster runs within your default VPC. To run your application, you launch an Amazon EC2 instance into your Amazon VPC. You then deploy your application (with the DAX client) on the EC2 instance.

At runtime, the DAX client directs all of your application's DynamoDB API requests to the DAX cluster. If DAX can process one of these API requests directly, it does so. Otherwise, it passes the request through to DynamoDB.

Finally, the DAX cluster returns the results to your application.

Topics

- [How DAX processes requests](#)
- [Item cache](#)
- [Query cache](#)

How DAX processes requests

A DAX cluster consists of one or more nodes. Each node runs its own instance of the DAX caching software. One of the nodes serves as the primary node for the cluster. Additional nodes (if present) serve as read replicas. For more information, see [Nodes](#).

Your application can access DAX by specifying the endpoint for the DAX cluster. The DAX client software works with the cluster endpoint to perform intelligent load balancing and routing.

Read operations

DAX can respond to the following API calls:

- `GetItem`
- `BatchGetItem`
- `Query`
- `Scan`

If the request specifies *eventually consistent reads* (the default behavior), it tries to read the item from DAX:

- If DAX has the item available (a *cache hit*), DAX returns the item to the application without accessing DynamoDB.
- If DAX does not have the item available (a *cache miss*), DAX passes the request through to DynamoDB. When it receives the response from DynamoDB, DAX returns the results to the application. But it also writes the results to the cache on the primary node.

 **Note**

If there are any read replicas in the cluster, DAX automatically keeps the replicas in sync with the primary node. For more information, see [Clusters](#).

If the request specifies *strongly consistent reads*, DAX passes the request through to DynamoDB. The results from DynamoDB are not cached in DAX. Instead, they are simply returned to the application.

Write operations

The following DAX API operations are considered "write-through":

- `BatchWriteItem`
- `UpdateItem`
- `DeleteItem`
- `PutItem`

With these operations, data is first written to the DynamoDB table, and then to the DAX cluster. The operation is successful only if the data is successfully written to *both* the table and to DAX.

Other operations

DAX does not recognize any DynamoDB operations for managing tables (such as `CreateTable`, `UpdateTable`, and so on). If your application needs to perform these operations, it must access DynamoDB directly rather than using DAX.

For detailed information about DAX and DynamoDB consistency, see [DAX and DynamoDB consistency models](#).

For information about how transactions work in DAX, see [Using transactional APIs in DynamoDB Accelerator \(DAX\)](#).

Request rate limiting

If the number of requests sent to DAX exceeds the capacity of a node, DAX limits the rate at which it accepts additional requests by returning a [ThrottlingException](#). DAX continuously evaluates your CPU utilization to determine the volume of requests it can process while maintaining a healthy cluster state.

You can monitor the [ThrottledRequestCount metric](#) that DAX publishes to Amazon CloudWatch. If you see these exceptions regularly, you should consider [scaling up your cluster](#).

Item cache

DAX maintains an *item cache* to store the results from `GetItem` and `BatchGetItem` operations. The items in the cache represent eventually consistent data from DynamoDB, and are stored by their primary key values.

When an application sends a `GetItem` or `BatchGetItem` request, DAX tries to read the items directly from the item cache using the specified key values. If the items are found (cache hit), DAX returns them to the application immediately. If the items are not found (cache miss), DAX sends the request to DynamoDB. DynamoDB processes the requests using eventually consistent reads and returns the items to DAX. DAX stores them in the item cache and then returns them to the application.

The item cache has a Time to Live (TTL) setting, which is 5 minutes by default. DAX assigns a timestamp to every item that it writes to the item cache. An item expires if it has remained in the cache for longer than the TTL setting. If you issue a `GetItem` request on an expired item, this is considered a cache miss, and DAX sends the `GetItem` request to DynamoDB.

Note

You can specify the TTL setting for the item cache when you create a new DAX cluster. For more information, see [Managing DAX clusters](#).

DAX also maintains a least recently used (LRU) list for the item cache. The LRU list tracks when an item was first written to the cache, and when the item was last read from the cache. If the item cache becomes full, DAX evicts older items (even if they haven't expired yet) to make room for new items. The LRU algorithm is always enabled for the item cache and is not user-configurable.

If you specify zero as the *item cache* TTL setting, items in the item cache will only be refreshed due to an LRU eviction or a ["write-through"](#) operation.

For detailed information about the consistency of the item cache in DAX, see [DAX item cache behavior](#).

Query cache

DAX also maintains a *query cache* to store the results from `Query` and `Scan` operations. The items in this cache represent result sets from queries and scans on DynamoDB tables. These result sets are stored by their parameter values.

When an application sends a `Query` or `Scan` request, DAX tries to read a matching result set from the query cache using the specified parameter values. If the result set is found (cache hit), DAX returns it to the application immediately. If the result set is not found (cache miss), DAX sends

the request to DynamoDB. DynamoDB processes the requests using eventually consistent reads and returns the result set to DAX. DAX stores it in the query cache and then returns it to the application.

 **Note**

You can specify the TTL setting for the query cache when you create a new DAX cluster. For more information, see [Managing DAX clusters](#).

DAX also maintains an LRU list for the query cache. The list tracks when a result set was first written to the cache, and when the result was last read from the cache. If the query cache becomes full, DAX evicts older result sets (even if they have not expired yet) to make room for new result sets. The LRU algorithm is always enabled for the query cache, and is not user-configurable.

If you specify zero as the *query cache* TTL setting, the query response will not be cached.

For detailed information about the consistency of the query cache in DAX, see [DAX query cache behavior](#).

DAX cluster components

An Amazon DynamoDB Accelerator (DAX) cluster consists of AWS infrastructure components. This section describes these components and how they work together.

Topics

- [Nodes](#)
- [Clusters](#)
- [Regions and availability zones](#)
- [Parameter groups](#)
- [Security groups](#)
- [Cluster ARN](#)
- [Cluster endpoint](#)
- [Node endpoints](#)
- [Subnet groups](#)

- [Events](#)
- [Maintenance window](#)

Nodes

A *node* is the smallest building block of a DAX cluster. Each node runs an instance of the DAX software, and maintains a single replica of the cached data.

You can scale your DAX cluster in one of two ways:

- By adding more nodes to the cluster. This increases the overall read throughput of the cluster.
- By using a larger node type. Larger node types provide more capacity and can increase throughput. (You must create a new cluster with the new node type.)

Every node within a cluster is of the same node type and runs the same DAX caching software. For a list of available node types, see [Amazon DynamoDB pricing](#).

Clusters

A *cluster* is a logical grouping of one or more nodes that DAX manages as a unit. One of the nodes in the cluster is designated as the *primary* node, and the other nodes (if any) are *read replicas*.

The primary node is responsible for the following:

- Fulfilling application requests for cached data.
- Handling write operations to DynamoDB.
- Evicting data from the cache according to the cluster's eviction policy.

When changes are made to cached data on the primary node, DAX propagates the changes to all of the read replica nodes using replication logs. After the confirmation is received from all read replicas, DynamoDB deletes the replication logs from the primary node.

Read replicas are responsible for the following:

- Fulfilling application requests for cached data.
- Evicting data from the cache according to the cluster's eviction policy.

However, unlike the primary node, read replicas don't write to DynamoDB.

Read replicas serve two additional purposes:

- **Scalability.** If you have a large number of application clients that need to access DAX concurrently, you can add more replicas for read-scaling. DAX spreads the load evenly across all the nodes in the cluster. (Another way to increase throughput is to use larger cache node types.)
- **High availability.** In the event of a primary node failure, DAX automatically fails over to a read replica and designates it as the new primary. If a replica node fails, other nodes in the DAX cluster can still serve requests until the failed node can be recovered. For maximum fault tolerance, you should deploy read replicas in separate Availability Zones. This configuration ensures that your DAX cluster can continue to function, even if an entire Availability Zone becomes unavailable.

A DAX cluster can support up to 11 nodes per cluster (the primary node plus a maximum of 10 read replicas).

Important

For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in different Availability Zones. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

Regions and availability zones

A DAX cluster in an AWS Region can only interact with DynamoDB tables that are in the same Region. For this reason, ensure that you launch your DAX cluster in the correct Region. If you have DynamoDB tables in other Regions, you must launch DAX clusters in those Regions too.

Each Region is designed to be completely isolated from the other Regions. Within each Region are multiple Availability Zones. By launching your nodes in different Availability Zones, you can achieve the greatest possible fault tolerance.

⚠ Important

Don't place all of your cluster's nodes in a single Availability Zone. In this configuration, your DAX cluster becomes unavailable if there is an Availability Zone failure.

For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in different Availability Zones. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

Parameter groups

Parameter groups are used to manage runtime settings for DAX clusters. DAX has several parameters that you can use to optimize performance (such as defining a TTL policy for cached data). A parameter group is a named set of parameters that you can apply to a cluster. You can thereby ensure that all the nodes in that cluster are configured in exactly the same way.

Security groups

A DAX cluster runs in an Amazon Virtual Private Cloud (Amazon VPC) environment. This environment is a virtual network that is dedicated to your AWS account and is isolated from other VPCs. A *security group* acts as a virtual firewall for your VPC, allowing you to control inbound and outbound network traffic.

When you launch a cluster in your VPC, you add an *ingress* rule to your security group to allow incoming network traffic. The ingress rule specifies the protocol (TCP) and port number (8111) for your cluster. After you add this rule to your security group, the applications that are running within your VPC can access the DAX cluster.

Cluster ARN

Every DAX cluster is assigned an *Amazon Resource Name* (ARN). The ARN format is as follows.

```
arn:aws:dax:region:accountID:cache/clusterName
```

You use the cluster ARN in an IAM policy to define permissions for DAX API operations. For more information, see [DAX access control](#).

Cluster endpoint

Every DAX cluster provides a *cluster endpoint* for use by your application. By accessing the cluster using its endpoint, your application does not need to know the hostnames and port numbers of individual nodes in the cluster. Your application automatically "knows" all the nodes in the cluster, even if you add or remove read replicas.

The following is an example of a cluster endpoint in the us-east-1 region that is not configured to use encryption in transit.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

The following is an example of a cluster endpoint in the same region that is configured to use encryption in transit.

```
daxs://my-encrypted-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

Node endpoints

Each of the individual nodes in a DAX cluster has its own hostname and port number. The following is an example of a *node endpoint*.

```
myDAXcluster-a.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111
```

Your application can access a node directly by using its endpoint. However, we recommend that you treat the DAX cluster as a single unit and access it using the cluster endpoint instead. The cluster endpoint insulates your application from having to maintain a list of nodes and keep that list up to date when you add or remove nodes from the cluster.

Subnet groups

Access to DAX cluster nodes is restricted to applications running on Amazon EC2 instances within an Amazon VPC environment. You can use *subnet groups* to grant cluster access from Amazon EC2 instances running on specific subnets. A subnet group is a collection of subnets (typically private) that you can designate for your clusters running in an Amazon VPC environment.

When you create a DAX cluster, you must specify a subnet group. DAX uses that subnet group to select a subnet and IP addresses within that subnet to associate with your nodes.

Events

DAX records significant events within your clusters, such as a failure to add a node, success in adding a node, or changes to security groups. By monitoring key events, you can know the current state of your clusters and, depending upon the event, be able to take corrective action. You can access these events using the AWS Management Console or the `DescribeEvents` action in the DAX management API.

You can also request that notifications be sent to a specific Amazon Simple Notification Service (Amazon SNS) topic. Then you will know immediately when an event occurs in your DAX cluster.

Maintenance window

Every cluster has a weekly maintenance window during which any system changes are applied. If you don't specify a preferred maintenance window when you create or modify a cache cluster, DAX assigns a 60-minute maintenance window on a randomly selected day of the week.

The 60-minute maintenance window is selected at random from an 8-hour block of time per AWS Region. The following table lists the time blocks for each Region from which the default maintenance windows are assigned.

Region code	Region name	Maintenance window
ap-northeast-1	Asia Pacific (Tokyo) Region	13:00–21:00 UTC
ap-southeast-1	Asia Pacific (Singapore) Region	14:00–22:00 UTC
ap-southeast-2	Asia Pacific (Sydney) Region	12:00–20:00 UTC
ap-south-1	Asia Pacific (Mumbai) Region	17:30–1:30 UTC
cn-northwest-1	China (Ningxia) Region	23:00–07:00 UTC
cn-north-1	China (Beijing) Region	14:00–22:00 UTC
eu-central-1	Europe (Frankfurt) Region	23:00–07:00 UTC
eu-west-1	Europe (Ireland) Region	22:00–06:00 UTC

Region code	Region name	Maintenance window
eu-west-2	Europe (London) Region	23:00–07:00 UTC
eu-west-3	Europe (Paris) Region	23:00–07:00 UTC
sa-east-1	South America (São Paulo) Region	01:00–09:00 UTC
us-east-1	US East (N. Virginia) Region	03:00–11:00 UTC
us-east-2	US East (Ohio) Region	23:00–07:00 UTC
us-west-1	US West (N. California) Region	06:00–14:00 UTC
us-west-2	US West (Oregon) Region	06:00–14:00 UTC

The maintenance window should fall at the time of lowest usage and thus might need modification from time to time. You can specify a time range of up to 24 hours in duration during which any maintenance activities that you request should occur.

Creating a DAX cluster

This section walks you through the first-time setup and usage of Amazon DynamoDB Accelerator (DAX) in your default Amazon Virtual Private Cloud (Amazon VPC) environment. You can create your first DAX cluster using either the AWS Command Line Interface (AWS CLI) or the AWS Management Console.

After you create your DAX cluster, you can access it from an Amazon EC2 instance running in the same VPC. You can then use your DAX cluster with an application program. For more information, see [Developing with the DynamoDB Accelerator \(DAX\) client](#).

Topics

- [Creating an IAM service role for DAX to access DynamoDB](#)
- [Creating a DAX cluster using the AWS CLI](#)
- [Creating a DAX cluster using the AWS Management Console](#)

Creating an IAM service role for DAX to access DynamoDB

For your DAX cluster to access DynamoDB tables on your behalf, you must create a *service role*. A service role is an AWS Identity and Access Management (IAM) role that authorizes an AWS service to act on your behalf. The service role allows DAX to access your DynamoDB tables, as if you were accessing those tables yourself. You must create the service role before you can create the DAX cluster.

If you are using the console, the workflow for creating a cluster checks for the presence of a pre-existing DAX service role. If none is found, the console creates a new service role for you. For more information, see [the section called “Step 2: Create a DAX cluster”](#).

If you are using the AWS CLI, you must specify a DAX service role that you have created previously. Otherwise, you need to create a new service role beforehand. For more information, see [Step 1: Create an IAM service role for DAX to access DynamoDB using the AWS CLI](#).

Permissions required to create a service role

The AWS managed AdministratorAccess policy provides all the permissions needed for creating a DAX cluster and a service role. If your user has AdministratorAccess attached, no further action is needed.

Otherwise, you must add the following permissions to your IAM policy so that your user can create the service role:

- `iam:CreateRole`
- `iam:CreatePolicy`
- `iam:AttachRolePolicy`
- `iam:PassRole`

Attach these permissions to the user who is trying to perform the action.

Note

The `iam:CreateRole`, `iam:CreatePolicy`, `iam:AttachRolePolicy`, and `iam:PassRole` permissions are not included in the AWS managed policies for DynamoDB. This is by design because these permissions provide the possibility of privilege escalation: That is, a user could use these permissions to create a new administrator policy and then

attach that policy to an existing role. For this reason, you (the administrator of your DAX cluster) must explicitly add these permissions to your policy.

Troubleshooting

If your user policy is missing the `iam:CreateRole`, `iam:CreatePolicy`, and `iam:AttachPolicy` permissions, you will get error messages. The following table lists these messages and describes how to correct the problems.

If you see this error message...	Do the following:
User: arn:aws:iam:: <i>accountID</i> :user/ <i>userName</i> is not authorized to perform: iam:CreateRole on resource: arn:aws:iam:: <i>accountID</i> :role/service-role/ <i>roleName</i>	Add <code>iam:CreateRole</code> to your user policy.
User: arn:aws:iam:: <i>accountID</i> :user/ <i>userName</i> is not authorized to perform: iam:CreatePolicy on resource: policy <i>policyName</i>	Add <code>iam:CreatePolicy</code> to your user policy.
User: arn:aws:iam:: <i>accountID</i> :user/ <i>userName</i> is not authorized to perform: iam:AttachRolePolicy on resource: role <i>daxServiceRole</i>	Add <code>iam:AttachRolePolicy</code> to your user policy.

For more information about the IAM policies required for DAX cluster administration, see [DAX access control](#).

Creating a DAX cluster using the AWS CLI

This section describes how to create an Amazon DynamoDB Accelerator (DAX) cluster using the AWS Command Line Interface (AWS CLI). If you haven't already done so, you must install and configure the AWS CLI. To do this, see the following instructions in the *AWS Command Line Interface User Guide*:

- [Installing the AWS CLI](#)
- [Configuring the AWS CLI](#)

A **Important**

To manage DAX clusters using the AWS CLI, install or upgrade to version 1.11.110 or higher.

All of the AWS CLI examples use the us-west-2 Region and fictitious account IDs.

Topics

- [Step 1: Create an IAM service role for DAX to access DynamoDB using the AWS CLI](#)
- [Step 2: Create a subnet group](#)
- [Step 3: Create a DAX cluster using the AWS CLI](#)
- [Step 4: Configure security group inbound rules using the AWS CLI](#)

Step 1: Create an IAM service role for DAX to access DynamoDB using the AWS CLI

Before you can create an Amazon DynamoDB Accelerator (DAX) cluster, you must create a service role for it. A *service role* is an AWS Identity and Access Management (IAM) role that authorizes an AWS service to act on your behalf. The service role allows DAX to access your DynamoDB tables as if you were accessing those tables yourself.

In this step, you create an IAM policy and then attach that policy to an IAM role. This enables you to assign the role to a DAX cluster so that it can perform DynamoDB operations on your behalf.

To create an IAM service role for DAX

1. Create a file named `service-trust-relationship.json` with the following contents.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {
```

```
        "Service": "dax.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
]
}
}
```

2. Create the service role.

```
aws iam create-role \
--role-name DAXServiceRoleForDynamoDBAccess \
--assume-role-policy-document file://service-trust-relationship.json
```

3. Create a file named `service-role-policy.json` with the following contents.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-west-2:accountID:*"
            ]
        }
    ]
}
```

Replace **accountID** with your AWS account ID. To find your AWS account ID, in the upper-right corner of the console, choose your login ID. Your AWS account ID appears in the drop-down menu.

In the Amazon Resource Name (ARN) in the example, *accountID* must be a 12-digit number. Don't use hyphens or any other punctuation.

4. Create an IAM policy for the service role.

```
aws iam create-policy \
--policy-name DAXServicePolicyForDynamoDBAccess \
--policy-document file://service-role-policy.json
```

In the output, note the ARN for the policy that you created, as in the following example.

arn:aws:iam::123456789012:policy/DAXServicePolicyForDynamoDBAccess

5. Attach the policy to the service role. Replace *arn* in the following code with the actual role ARN from the previous step.

```
aws iam attach-role-policy \
--role-name DAXServiceRoleForDynamoDBAccess \
--policy-arn arn
```

Next, you specify a subnet group for your default VPC. A *subnet group* is a collection of one or more subnets within your VPC. See [Step 2: Create a subnet group](#).

Step 2: Create a subnet group

Follow this procedure to create a subnet group for your Amazon DynamoDB Accelerator (DAX) cluster using the AWS Command Line Interface (AWS CLI).

Note

If you already created a subnet group for your default VPC, you can skip this step.

DAX is designed to run within an Amazon Virtual Private Cloud environment (Amazon VPC). If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. For more information, see [Default VPC and default subnets](#) in the *Amazon VPC User Guide*.

To create a subnet group

1. To determine the identifier for your default VPC, enter the following command.

```
aws ec2 describe-vpcs
```

In the output, note the identifier for your default VPC, as in the following example.

vpc-12345678

2. Determine the subnet IDs associated with your default VPC. Replace *vpcID* with your actual VPC ID—for example, vpc-12345678.

```
aws ec2 describe-subnets \
--filters "Name=vpc-id,Values=vpcID" \
--query "Subnets[*].SubnetId"
```

In the output, note the subnet identifiers—for example, subnet-11111111.

3. Create the subnet group. Ensure that you specify at least one subnet ID in the --subnet-ids parameter.

```
aws dax create-subnet-group \
--subnet-group-name my-subnet-group \
--subnet-ids subnet-11111111 subnet-22222222 subnet-33333333 subnet-44444444
```

To create the cluster, see [Step 3: Create a DAX cluster using the AWS CLI](#).

Step 3: Create a DAX cluster using the AWS CLI

Follow this procedure to use the AWS Command Line Interface (AWS CLI) to create an Amazon DynamoDB Accelerator (DAX) cluster in your default Amazon VPC.

To create a DAX cluster

1. Get the Amazon Resource Name (ARN) for your service role.

```
aws iam get-role \
--role-name DAXServiceRoleForDynamoDBAccess \
--query "Role.Arn" --output text
```

In the output, note the service role ARN, as in the following example.

arn:aws:iam::123456789012:role/DAXServiceRoleForDynamoDBAccess

2. Create the DAX cluster. Replace *roleARN* with the ARN from the previous step.

```
aws dax create-cluster \
--cluster-name mydaxcluster \
--node-type dax.r4.large \
--replication-factor 3 \
--iam-role-arn roleARN \
--subnet-group my-subnet-group \
--sse-specification Enabled=true \
--region us-west-2
```

All of the nodes in the cluster are of type `dax.r4.large` (`--node-type`). There are three nodes (`--replication-factor`)—one primary node and two replicas.

 **Note**

Since `sudo` and `grep` are reserved keywords, you cannot create a DAX cluster with these words in the cluster name. For example, `sudo` and `sudocluster` are invalid cluster names.

To view the cluster status, enter the following command.

```
aws dax describe-clusters
```

The status is shown in the output—for example, "Status": "creating".

 **Note**

Creating the cluster takes several minutes. When the cluster is ready, its status changes to available. In the meantime, proceed to [Step 4: Configure security group inbound rules using the AWS CLI](#) and follow the instructions there.

Step 4: Configure security group inbound rules using the AWS CLI

The nodes in your Amazon DynamoDB Accelerator (DAX) cluster use the default security group for your Amazon VPC. For the default security group, you must authorize inbound traffic on TCP

port 8111 for unencrypted clusters or port 9111 for encrypted clusters. This allows Amazon EC2 instances in your Amazon VPC to access your DAX cluster.

Note

If you launched your DAX cluster with a different security group (other than default), you must perform this procedure for that group instead.

To configure security group inbound rules

1. To determine the default security group identifier, enter the following command. Replace *vpcID* with your actual VPC ID (from [Step 2: Create a subnet group](#)).

```
aws ec2 describe-security-groups \
--filters Name=vpc-id,Values=vpcID Name=group-name,Values=default \
--query "SecurityGroups[*].{GroupName:GroupName,GroupId:GroupId}"
```

In the output, note the security group identifier—for example, sg-01234567.

2. Then enter the following. Replace *sgID* with your actual security group identifier. Use port 8111 for unencrypted clusters and 9111 for encrypted clusters.

```
aws ec2 authorize-security-group-ingress \
--group-id sgID --protocol tcp --port 8111
```

Creating a DAX cluster using the AWS Management Console

This section describes how to create an Amazon DynamoDB Accelerator (DAX) cluster using the AWS Management Console.

Topics

- [Step 1: Create a subnet group using the AWS Management Console](#)
- [Step 2: Create a DAX cluster using the AWS Management Console](#)
- [Step 3: Configure security group inbound rules using the AWS Management Console](#)

Step 1: Create a subnet group using the AWS Management Console

Follow this procedure to create a subnet group for your Amazon DynamoDB Accelerator (DAX) cluster using the AWS Management Console.

 **Note**

If you already created a subnet group for your default VPC, you can skip this step.

DAX is designed to run within an Amazon Virtual Private Cloud (Amazon VPC) environment. If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. For more information, see [Default VPC and default subnets](#) in the *Amazon VPC User Guide*.

As part of the creation process for a DAX cluster, you must specify a *subnet group*. A subnet group is a collection of one or more subnets within your VPC. When you create your DAX cluster, the nodes are deployed to the subnets within the subnet group.

To create a subnet group

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, under **DAX**, choose **Subnet groups**.
3. Choose **Create subnet group**.
4. In the **Create subnet group** window, do the following:
 - a. **Name**—Enter a short name for the subnet group.
 - b. **Description**—Enter a description for the subnet group.
 - c. **VPC ID**—Choose the identifier for your Amazon VPC environment.
 - d. **Subnets**—Choose one or more subnets from the list.

 **Note**

The subnets are distributed among multiple Availability Zones. If you plan to create a multi-node DAX cluster (a primary node and one or more read replicas), we recommend that you choose multiple subnet IDs. Then DAX can deploy the cluster nodes into multiple Availability Zones. If an Availability Zone becomes

unavailable, DAX automatically fails over to a surviving Availability Zone. Your DAX cluster will continue to function without interruption.

When the settings are as you want them, choose **Create subnet group**.

To create the cluster, see [Step 2: Create a DAX cluster using the AWS Management Console](#).

Step 2: Create a DAX cluster using the AWS Management Console

Follow this procedure to create an Amazon DynamoDB Accelerator (DAX) cluster in your default Amazon VPC.

To create a DAX cluster

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose **Create cluster**.
4. In the **Create cluster** window, do the following:
 - a. **Cluster name**—Enter a short name for your DAX cluster.

 **Note**

Since sudo and grep are reserved keywords, you cannot create a DAX cluster with these words in the cluster name. For example, sudo and sudocluster are invalid cluster names.

- b. **Cluster description**—Enter a description for the cluster.
- c. **Node types**—Choose the node type for all of the nodes in the cluster.
- d. **Cluster size**—Choose the number of nodes in the cluster. A cluster consists of one primary node and up to nine read replicas.

 **Note**

If you want to create a single-node cluster, choose **1**. Your cluster will consist of one primary node.

If you want to create a multi-node cluster, choose a number between **3** (one primary and two read replicas) and **10** (one primary and nine read replicas).

 **Important**

For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in a different Availability Zone. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

- e. Choose **Next**.
- f. **Subnet group**—Select **Choose existing** and choose the subnet group that you created in [Step 1: Create a subnet group using the AWS Management Console](#).
- g. **Access control**—Choose the **default** security group.
- h. **Availability Zones (AZ)**—Choose **Automatic**.
- i. Choose next.
- j. **IAM service role for DynamoDB access**—Choose **Create new**, and enter the following information:
 - **IAM role name**—Enter a name for an IAM role, for example, `DAXServiceRole`. The console creates a new IAM role, and your DAX cluster assumes this role at runtime.
 - Select the box next to **Create policy**.
 - **IAM role policy**—Choose **Read/Write**. This allows the DAX cluster to perform read and write operations in DynamoDB.
 - **New IAM policy name**—This field will populate as you enter the IAM role name. You can also enter a name for an IAM policy, for example, `DAXServicePolicy`. The console creates a new IAM policy and attaches the policy to the IAM role.
 - **Access to DynamoDB tables**—Choose **All tables**.
- k. **Encryption**—Choose **Turn on encryption at rest** and **Turn on encryption in transit** For more information, see [DAX encryption at rest](#) and [DAX encryption in transit](#).

A separate service role for DAX to access Amazon EC2 is also required. DAX automatically creates this service role for you. For more information, see [Using service-linked roles for DAX](#).

5. When the settings are as you want them, choose **Next**.
6. **Parameter group**—Choose **Choose existing**.
7. **Maintenance window**—Choose **No preference** if you don't have a preference when software upgrades are applied, or choose **Specify time window** and provide the **Weekday**, **Time(UTC)** and **Start within (hours)** options to schedule your maintenance window.
8. **Tags**—Choose **Add new tag** to enter a key/value pair for tagging purposes.
9. Choose **Next**.

On the **Review and create** screen, you can review all of the settings. If you are ready to create the cluster, choose **Create cluster**.

On the **Clusters** screen, your DAX cluster will be listed with a status of **Creating**.

 **Note**

Creating the cluster takes several minutes. When the cluster is ready, its status changes to **Available**.

In the meantime, proceed to [Step 3: Configure security group inbound rules using the AWS Management Console](#) and follow the instructions there.

Step 3: Configure security group inbound rules using the AWS Management Console

Your Amazon DynamoDB Accelerator (DAX) cluster communicates via TCP port 8111 (for unencrypted clusters) or 9111 (for encrypted clusters), so you must authorize inbound traffic on that port. This allows Amazon EC2 instances in your Amazon VPC to access your DAX cluster.

 **Note**

If you launched your DAX cluster with a different security group (other than default), you must perform this procedure for that group instead.

To configure security group inbound rules

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. In the navigation pane, choose **Security Groups**.
3. Choose the **default** security group. On the **Actions** menu, choose **Edit inbound rules**.
4. Choose **Add Rule**, and enter the following information:
 - **Port Range**—Enter **8111** (if your cluster is unencrypted) or **9111** (if your cluster is encrypted).
 - **Source**—Leave this as **Custom**, and choose the search field to the right. A drop-down menu will be displayed. Choose the identifier for your default security group.
5. Choose **Save rules** to save your changes.
6. To update the name in the console, go to the **Name** property and choose the **Edit** option that is displayed.

DAX and DynamoDB consistency models

Amazon DynamoDB Accelerator (DAX) is a write-through caching service that is designed to simplify the process of adding a cache to DynamoDB tables. Because DAX operates separately from DynamoDB, it is important that you understand the consistency models of both DAX and DynamoDB to ensure that your applications behave as you expect.

In many use cases, the way that your application uses DAX affects the consistency of data within the DAX cluster, and the consistency of data between DAX and DynamoDB.

Topics

- [Consistency among DAX cluster nodes](#)
- [DAX item cache behavior](#)
- [DAX query cache behavior](#)
- [Strongly consistent and transactional reads](#)
- [Negative caching](#)
- [Strategies for writes](#)

Consistency among DAX cluster nodes

To achieve high availability for your application, we recommend that you provision your DAX cluster with at least three nodes. Then place those nodes in multiple Availability Zones within a Region.

When your DAX cluster is running, it replicates the data among all of the nodes in the cluster (assuming that you provisioned more than one node). Consider an application that performs a successful `UpdateItem` using DAX. This action causes the item cache in the primary node to be modified with the new value. That value is then replicated to all the other nodes in the cluster. This replication is eventually consistent and usually takes less than one second to complete.

In this scenario, it's possible for two clients to read the same key from the same DAX cluster but receive different values, depending on the node that each client accessed. The nodes are all consistent when the update has been fully replicated throughout all the nodes in the cluster. (This behavior is similar to the eventually consistent nature of DynamoDB.)

If you are building an application that uses DAX, that application should be designed so that it can tolerate eventually consistent data.

DAX item cache behavior

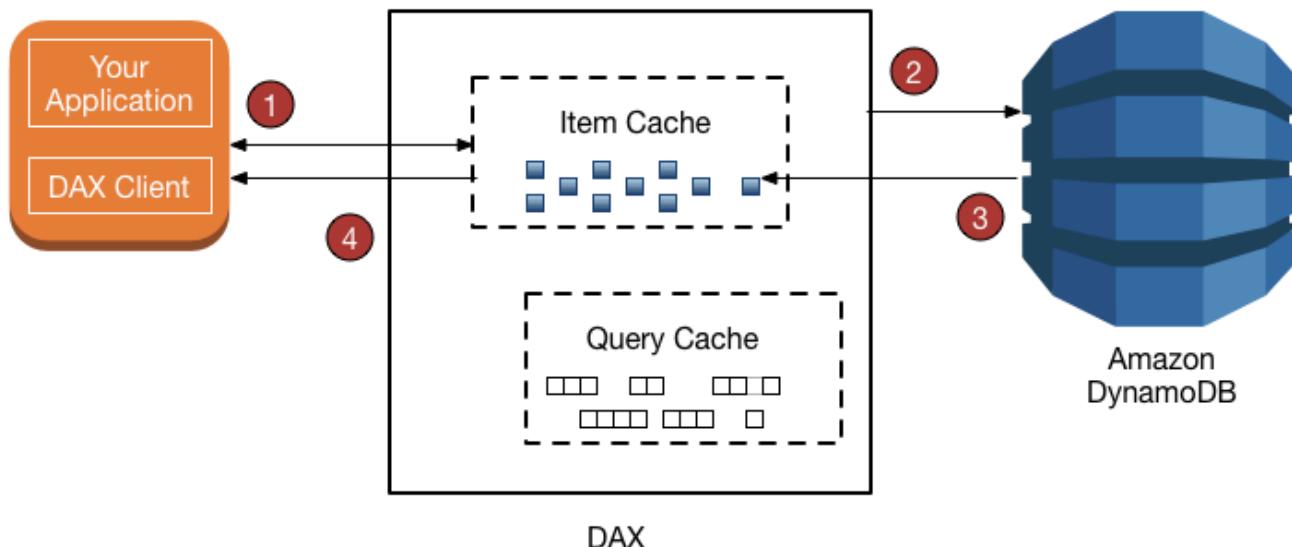
Every DAX cluster has two distinct caches—an *item* cache and a *query* cache. For more information, see [DAX: How it works](#).

This section addresses the consistency implications of reading from and writing to the DAX item cache.

Consistency of reads

With DynamoDB, the `GetItem` operation performs an eventually consistent read by default. Suppose that you use `UpdateItem` with the DynamoDB client. If you then try to read the same item immediately afterward, you might see the data as it appeared before the update. This is due to propagation delay across all the DynamoDB storage locations. Consistency is usually reached within seconds. So if you retry the read, you will likely see the updated item.

When you use `GetItem` with the DAX client, the operation (in this case, an eventually consistent read) proceeds as shown following.



1. The DAX client issues a `GetItem` request. DAX tries to read the requested item from the item cache. If the item is in the cache (*cache hit*), DAX returns it to the application.
2. If the item is not available (*cache miss*), DAX performs an eventually consistent `GetItem` operation against DynamoDB.
3. DynamoDB returns the requested item, and DAX stores it in the item cache.
4. DAX returns the item to the application.
5. (Not shown) If the DAX cluster contains more than one node, the item is replicated to all the other nodes in the cluster.

The item remains in the DAX item cache, subject to the Time to Live (TTL) setting and the least recently used (LRU) algorithm for the cache. For more information, see [DAX: How it works](#).

However, during this period, DAX doesn't re-read the item from DynamoDB. If someone else updates the item using a DynamoDB client, bypassing DAX entirely, a `GetItem` request using the DAX client yields different results from the same `GetItem` request using the DynamoDB client. In this scenario, DAX and DynamoDB hold inconsistent values for the same key until the TTL for the DAX item expires.

If an application modifies data in an underlying DynamoDB table, bypassing DAX, the application needs to anticipate and tolerate data inconsistencies that might arise.

Note

In addition to `GetItem`, the DAX client also supports `BatchGetItem` requests.

`BatchGetItem` is essentially a wrapper around one or more `GetItem` requests, so DAX treats each of these as an individual `GetItem` operation.

Consistency of writes

DAX is a write-through cache, which simplifies the process of keeping the DAX item cache consistent with the underlying DynamoDB tables.

The DAX client supports the same write API operations as DynamoDB (`PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, and `TransactWriteItems`). When you use these operations with the DAX client, the items are modified in both DAX and DynamoDB. DAX updates the items in its item cache, regardless of the TTL value for these items.

For example, suppose that you issue a `GetItem` request from the DAX client to read an item from the `ProductCatalog` table. (The partition key is `Id`, and there is no sort key.) You retrieve the item whose `Id` is `101`. The `QuantityOnHand` value for that item is `42`. DAX stores the item in its item cache with a specific TTL. For this example, assume that the TTL is `10` minutes. Then, `3` minutes later, another application uses the DAX client to update the same item so that its `QuantityOnHand` value is now `41`. Assuming that the item is not updated again, any subsequent reads of the same item during the next `10` minutes return the cached value for `QuantityOnHand` (`41`).

How DAX processes writes

DAX is intended for applications that require high-performance reads. As a write-through cache, DAX passes your writes through to DynamoDB synchronously, then automatically and asynchronously replicates resulting updates to your item cache across all nodes in the cluster. You don't need to manage cache invalidation logic because DAX handles it for you.

DAX supports the following write operations: `PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, and `TransactWriteItems`.

When you send a `PutItem`, `UpdateItem`, `DeleteItem`, or `BatchWriteItem` request to DAX, the following occurs:

- DAX sends the request to DynamoDB.
- DynamoDB replies to DAX, confirming that the write succeeded.
- DAX writes the item to its item cache.
- DAX returns success to the requester.

When you send a `TransactWriteItems` request to DAX, the following occurs:

- DAX sends the request to DynamoDB.
- DynamoDB replies to DAX, confirming that the transaction completed.
- DAX returns success to the requester.
- In the background, DAX makes a `TransactGetItems` request for each item in the `TransactWriteItems` request to store the item in the item cache. `TransactGetItems` is used to ensure [Serializable isolation](#).

If a write to DynamoDB fails for any reason, including throttling, the item is not cached in DAX. The exception for the failure is returned to the requester. This ensures that data is not written to the DAX cache unless it is first written successfully to DynamoDB.

 **Note**

Every write to DAX alters the state of the item cache. However, writes to the item cache don't affect the query cache. (The DAX item cache and query cache serve different purposes, and operate independently from one another.)

DAX query cache behavior

DAX caches the results from Query and Scan requests in its query cache. However, these results don't affect the item cache at all. When your application issues a Query or Scan request with DAX, the result set is saved in the query cache—not in the item cache. You can't "warm up" the item cache by performing a Scan operation because the item cache and query cache are separate entities.

Consistency of query-update-query

Updates to the item cache, or to the underlying DynamoDB table, do not invalidate or modify the results stored in the query cache.

To illustrate, consider the following scenario. An application is working with the DocumentRevisions table, which has DocId as its partition key and RevisionNumber as its sort key.

1. A client issues a Query for DocId 101, for all items with RevisionNumber greater than or equal to 5. DAX stores the result set in the query cache and returns the result set to the user.
2. The client issues a PutItem request for DocId 101 with a RevisionNumber value of 20.
3. The client issues the same Query as described in step 1 (DocId 101 and RevisionNumber >= 5).

In this scenario, the cached result set for the Query issued in step 3 is identical to the result set that was cached in step 1. The reason is that DAX does not invalidate Query or Scan result sets based on updates to individual items. The PutItem operation from step 2 is only reflected in the DAX query cache when the TTL for the Query expires.

Your application should consider the TTL value for the query cache and how long your application can tolerate inconsistent results between the query cache and the item cache.

Strongly consistent and transactional reads

To perform a strongly consistent GetItem, BatchGetItem, Query, or Scan request, you set the ConsistentRead parameter to true. DAX passes strongly consistent read requests to DynamoDB. When it receives a response from DynamoDB, DAX returns the results to the client, but it does not cache the results. DAX can't serve strongly consistent reads by itself because it's not tightly coupled to DynamoDB. For this reason, any subsequent reads from DAX would have to be eventually consistent reads. And any subsequent strongly consistent reads would have to be passed through to DynamoDB.

DAX handles TransactGetItems requests the same way it handles strongly consistent reads. DAX passes all TransactGetItems requests to DynamoDB. When it receives a response from DynamoDB, DAX returns the results to the client, but it doesn't cache the results.

Negative caching

DAX supports negative cache entries in both the item cache and the query cache. A *negative cache entry* occurs when DAX can't find requested items in an underlying DynamoDB table. Instead of generating an error, DAX caches an empty result and returns that result to the user.

For example, suppose that an application sends a `GetItem` request to a DAX cluster, and that there is no matching item in the DAX item cache. This causes DAX to read the corresponding item from the underlying DynamoDB table. If the item doesn't exist in DynamoDB, DAX stores an empty item in its item cache and returns the empty item to the application. Now suppose that the application sends another `GetItem` request for the same item. DAX finds the empty item in the item cache and returns it to the application immediately. It does not consult DynamoDB at all.

A negative cache entry remains in the DAX item cache until its item TTL has expired, its LRU is invoked, or the item is modified using `PutItem`, `UpdateItem`, or `DeleteItem`.

The DAX query cache handles negative cache results in a similar way. If an application performs a `Query` or `Scan`, and the DAX query cache doesn't contain a cached result, DAX sends the request to DynamoDB. If there are no matching items in the result set, DAX stores an empty result set in the query cache and returns the empty result set to the application. Subsequent `Query` or `Scan` requests yield the same (empty) result set, until the TTL for that result set has expired.

Strategies for writes

The write-through behavior of DAX is appropriate for many application patterns. However, there are some application patterns where a write-through model might not be appropriate.

For applications that are sensitive to latency, writing through DAX incurs an extra network hop. So a write to DAX is a little slower than a write directly to DynamoDB. If your application is sensitive to write latency, you can reduce the latency by writing directly to DynamoDB instead. For more information, see [Write-around](#).

For write-intensive applications (such as those that perform bulk data loading), you might not want to write all of the data through DAX because only a small percentage of that data is ever read by the application. When you write large amounts of data through DAX, it must invoke its LRU algorithm to make room in the cache for the new items to be read. This diminishes the effectiveness of DAX as a read cache.

When you write an item to DAX, the item cache state is altered to accommodate the new item. (For example, DAX might need to evict older data from the item cache to make room for the new item.)

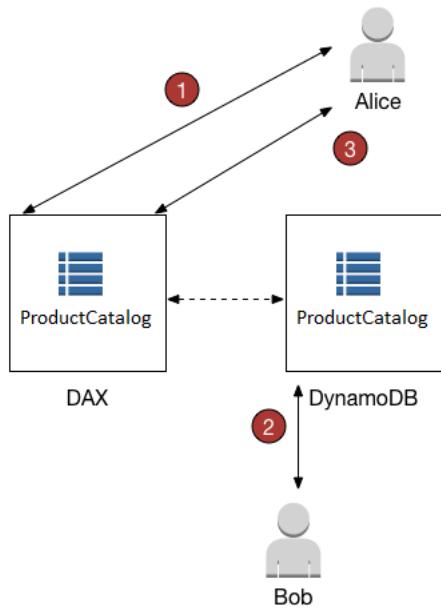
The new item remains in the item cache, subject to the cache's LRU algorithm and the TTL setting for the cache. As long as the item persists in the item cache, DAX doesn't re-read the item from DynamoDB.

Write-through

The DAX item cache implements a write-through policy. For more information, see [How DAX processes writes](#).

When you write an item, DAX ensures that the cached item is synchronized with the item as it exists in DynamoDB. This is helpful for applications that need to re-read an item immediately after writing it. However, if other applications write directly to a DynamoDB table, the item in the DAX item cache is no longer in sync with DynamoDB.

To illustrate, consider two users (Alice and Bob), who are working with the `ProductCatalog` table. Alice accesses the table using DAX, but Bob bypasses DAX and accesses the table directly in DynamoDB.



1. Alice updates an item in the `ProductCatalog` table. DAX forwards the request to DynamoDB, and the update succeeds. DAX then writes the item to its item cache and returns a successful response to Alice. From that point on, until the item is ultimately evicted from the cache, any user who reads the item from DAX sees the item with Alice's update.

2. A short time later, Bob updates the same ProductCatalog item that Alice wrote. However, Bob updates the item directly in DynamoDB. DAX does not automatically refresh its item cache in response to updates via DynamoDB. Therefore, DAX users don't see Bob's update.
3. Alice reads the item from DAX again. The item is in the item cache, so DAX returns it to Alice without accessing the DynamoDB table.

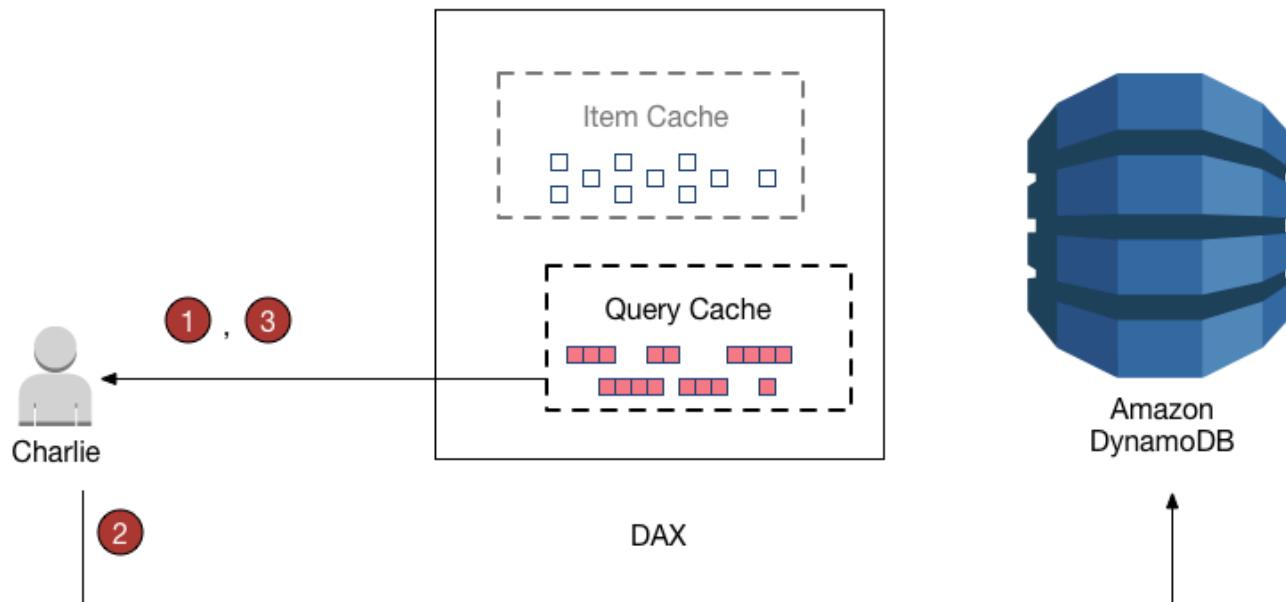
In this scenario, Alice and Bob see different representations of the same ProductCatalog item. This is the case until DAX evicts the item from the item cache, or until another user updates the same item again using DAX.

Write-around

If your application needs to write large quantities of data (such as a bulk data load), it might make sense to bypass DAX and write the data directly to DynamoDB. Such a *write-around* strategy reduces write latency. However, the item cache doesn't remain in sync with the data in DynamoDB.

If you decide to use a write-around strategy, remember that DAX populates its item cache whenever applications use the DAX client to read data. This can be advantageous in some cases because it ensures that only the most frequently read data is cached (as opposed to the most frequently written data).

For example, consider a user (Charlie) who wants to work with a different table, the GameScores table, using DAX. The partition key for GameScores is UserId, so all of Charlie's scores would have the same UserId.



1. Charlie wants to retrieve all of his scores, so he sends a Query to DAX. Assuming that this query has not been issued before, DAX forwards the query to DynamoDB for processing. It stores the results in the DAX query cache, and then returns the results to Charlie. The result set remains available in the query cache until it is evicted.
2. Now suppose that Charlie plays the Meteor Blasters game and achieves a high score. Charlie sends an UpdateItem request to DynamoDB, modifying an item in the GameScores table.
3. Finally, Charlie decides to rerun his earlier Query to retrieve all of his data from GameScores. Charlie does not see his high score for Meteor Blasters in the results. This is because the query results come from the query cache, not the item cache. The two caches are independent from one another, so a change in one cache does not affect the other cache.

DAX does not refresh result sets in the query cache with the most current data from DynamoDB. Each result set in the query cache is current as of the time that the Query or Scan operation was performed. Thus, Charlie's Query results don't reflect his PutItem operation. This is the case until DAX evicts the result set from the query cache.

Developing with the DynamoDB Accelerator (DAX) client

To use DAX from an application, you use the DAX client for your programming language. The DAX client is designed for minimal disruption to your existing Amazon DynamoDB applications—with only a few simple code modifications needed.

Note

DAX clients for various programming languages are available on the following site:

- <http://dax-sdk.s3-website-us-west-2.amazonaws.com>

This section demonstrates how to launch an Amazon EC2 instance in your default Amazon VPC, connect to the instance, and run a sample application. It also provides information about how to modify your existing application so that it can use your DAX cluster.

Topics

- [Tutorial: Running a sample application using DynamoDB Accelerator \(DAX\)](#)
- [Modifying an existing application to use DAX](#)

Tutorial: Running a sample application using DynamoDB Accelerator (DAX)

This tutorial demonstrates how to launch an Amazon EC2 instance in your default virtual private cloud (VPC), connect to the instance, and run a sample application that uses Amazon DynamoDB Accelerator (DAX).

Note

To complete this tutorial, you must have a DAX cluster running in your default VPC. If you haven't created a DAX cluster, see [Creating a DAX cluster](#) for instructions.

Topics

- [Step 1: Launch an Amazon EC2 instance](#)

- [Step 2: Create a user and policy](#)
- [Step 3: Configure an Amazon EC2 instance](#)
- [Step 4: Run a sample application](#)

Step 1: Launch an Amazon EC2 instance

When your Amazon DynamoDB Accelerator (DAX) cluster is available, you can launch an Amazon EC2 instance in your default Amazon VPC. You can then install and run DAX client software on that instance.

To launch an EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Choose **Launch Instance**, and do the following:

Step 1: Choose an Amazon Machine Image (AMI)

1. In the list of AMIs, find the **Amazon Linux AMI**, and choose **Select**.

Step 2: Choose an Instance Type

1. In the list of instance types, choose **t2.micro**.
2. Choose **Next: Configure Instance Details**.

Step 3: Configure Instance Details

1. For **Network**, choose your default VPC.
2. Choose **Next: Add Storage**.

Step 4: Add Storage

1. Skip this step by choosing **Next: Add Tags**.

Step 5: Add Tags

1. Skip this step by choosing **Next: Configure Security Group**.

Step 6: Configure Security Group

1. Choose **Select an existing security group**.
2. In the list of security groups, choose **default**. This is the default security group for your VPC.
3. Choose **Next: Review and Launch**.

Step 7: Review Instance Launch

1. Choose **Launch**.
3. In the **Select an existing key pair or create a new key pair** window, do one of the following:
 - If you don't have an Amazon EC2 key pair, choose **Create a new key pair** and follow the instructions. You are asked to download a private key file (. pem file). You need this file later when you log in to your Amazon EC2 instance.
 - If you already have an existing Amazon EC2 key pair, go to **Select a key pair** and choose your key pair from the list. You must already have the private key file (. pem file) available in order to log in to your Amazon EC2 instance.
4. After configuring your key pair, choose **Launch Instances**.
5. In the console navigation pane, choose **EC2 Dashboard**, and then choose the instance that you launched. In the lower pane, on the **Description** tab, find the **Public DNS** for your instance, for example: ec2-11-22-33-44.us-west-2.compute.amazonaws.com. Make a note of this public DNS name because you need it for [Step 3: Configure an Amazon EC2 instance](#).

 **Note**

It takes a few minutes for your Amazon EC2 instance to become available. In the meantime, proceed to [Step 2: Create a user and policy](#) and follow the instructions there.

Step 2: Create a user and policy

In this step, you create a user with a policy that grants access to your Amazon DynamoDB Accelerator (DAX) cluster and to DynamoDB using AWS Identity and Access Management. You can then run applications that interact with your DAX cluster.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create an administrative user

1. Enable IAM Identity Center.

For instructions, see [Enabling AWS IAM Identity Center](#) in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to an administrative user.

For a tutorial about using the IAM Identity Center directory as your identity source, see [Configure user access with the default IAM Identity Center directory](#) in the *AWS IAM Identity Center User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

Create a permission set. Follow the instructions in [Create a permission set](#) in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

Create a role for identity federation. Follow the instructions in [Creating a role for a third-party identity provider \(federation\)](#) in the *IAM User Guide*.

- IAM users:

- Create a role that your user can assume. Follow the instructions in [Creating a role for an IAM user](#) in the *IAM User Guide*.

- (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in [Adding permissions to a user \(console\)](#) in the *IAM User Guide*.

To use the JSON policy editor to create a policy

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane on the left, choose **Policies**.
If this is your first time choosing **Policies**, the **Welcome to Managed Policies** page appears. Choose **Get Started**.
3. At the top of the page, choose **Create policy**.
4. In the **Policy editor** section, choose the **JSON** option.
5. Enter or paste a JSON policy document. For details about the IAM policy language, see [IAM JSON policy reference](#).
6. Resolve any security warnings, errors, or general warnings generated during [policy validation](#), and then choose **Next**.

 **Note**

You can switch between the **Visual** and **JSON** editor options anytime. However, if you make changes or choose **Next** in the **Visual** editor, IAM might restructure your policy to optimize it for the visual editor. For more information, see [Policy restructuring](#) in the *IAM User Guide*.

7. (Optional) When you create or edit a policy in the AWS Management Console, you can generate a JSON or YAML policy template that you can use in AWS CloudFormation templates.

To do this, in the **Policy editor** choose **Actions**, and then choose **Generate CloudFormation template**. To learn more about AWS CloudFormation, see [AWS Identity and Access Management resource type reference](#) in the *AWS CloudFormation User Guide*.

8. When you are finished adding permissions to the policy, choose **Next**.
9. On the **Review and create** page, enter a **Policy name** and a **Description** (optional) for the policy that you are creating. Review **Permissions defined in this policy** to see the permissions that are granted by your policy.
10. (Optional) Add metadata to the policy by attaching tags as key-value pairs. For more information about using tags in IAM, see [Tagging IAM resources](#) in the *IAM User Guide*.
11. Choose **Create policy** to save your new policy.

Policy document – Copy and paste the following document to create the JSON policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dax:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        },  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        }  
    ]  
}
```

Step 3: Configure an Amazon EC2 instance

When your Amazon EC2 instance is available, you can log in to the instance and prepare it for use.

Note

The following steps assume that you are connecting to your Amazon EC2 instance from a computer running Linux. For other ways to connect, see [Connect to Your Linux Instance](#) in the *Amazon EC2 User Guide for Linux Instances*.

To configure the EC2 instance

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.

2. Use the ssh command to log in to your Amazon EC2 instance, as shown in the following example.

```
ssh -i my-keypair.pem ec2-user@public-dns-name
```

You need to specify your private key file (. pem file) and the public DNS name of your instance. (See [Step 1: Launch an Amazon EC2 instance](#).)

The login ID is ec2-user. No password is required.

3. After you log in to your EC2 instance, configure your AWS credentials as shown following. Enter your AWS access key ID and secret key (from [Step 2: Create a user and policy](#)), and set the default Region name to your current Region. (In the following example, the default Region name is us-west-2.)

```
aws configure
```

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
```

```
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRficiYEXAMPLEKEY
```

```
Default region name [None]: us-west-2
```

```
Default output format [None]:
```

After launching and configuring your Amazon EC2 instance, you can test the functionality of DAX using one of the available sample applications. For more information, see [Step 4: Run a sample application](#).

Step 4: Run a sample application

To help you test Amazon DynamoDB Accelerator (DAX) functionality, you can run one of the available sample applications on your Amazon EC2 instance.

Topics

- [DAX SDK for Go](#)
- [Java and DAX](#)
- [.NET and DAX](#)
- [Node.js and DAX](#)
- [Python and DAX](#)

DAX SDK for Go

Follow this procedure to run the Amazon DynamoDB Accelerator (DAX) SDK for Go sample application on your Amazon EC2 instance.

To run the SDK for Go sample for DAX

1. Set up the SDK for Go on your Amazon EC2 instance:

- a. Install the Go programming language (Golang).

```
sudo yum install -y golang
```

- b. Test that Golang is installed and running correctly.

```
go version
```

A message like this should appear.

```
go version go1.15.5 linux/amd64
```

The remaining instructions rely on module support, which became the default with Go version 1.13.

2. Install the sample Golang application.

```
go get github.com/aws-samples/aws-dax-go-sample
```

3. Run the following Golang programs. The first program creates a DynamoDB table named TryDaxGoTable. The second program writes data to the table.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -service dynamodb -command create-table
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -service dynamodb -command put-item
```

4. Run the following Golang programs.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -service dynamodb -command get-item
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -service dynamodb -command query
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -service dynamodb -command scan
```

Take note of the timing information—the number of milliseconds required for the GetItem, Query, and Scan tests.

5. In the previous step, you ran the programs against the DynamoDB endpoint. Now, run the programs again, but this time, the GetItem, Query, and Scan operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint is shown in the output, as in the following example.

```
{  
    "Address": "my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com",  
    "Port": 8111,  
    "URL": "dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com"  
}
```

Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go  
-service dax -command get-item -endpoint my-cluster.16fzcv.dax-clusters.us-  
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go  
-service dax -command query -endpoint my-cluster.16fzcv.dax-clusters.us-  
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go  
-service dax -command scan -endpoint my-cluster.16fzcv.dax-clusters.us-  
east-1.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for GetItem, Query, and Scan should be significantly lower with DAX than with DynamoDB.

6. Run the following Golang program to delete TryDaxGoTable.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -  
service dynamodb -command delete-table
```

Java and DAX

DAX SDK for Java 2.x is compatible with [AWS SDK for Java 2.x](#). It's built on top of Java 8+ and includes support for non-blocking I/O. For information about using DAX with AWS SDK for Java 1.x, see [Using DAX with AWS SDK for Java 1.x](#)

Using the client as a Maven dependency

Follow these steps to use the client for the DAX SDK for Java in your application as a dependency.

1. Download and install Apache Maven. For more information, see [Downloading Apache Maven](#) and [Installing Apache Maven](#).
2. Add the client Maven dependency to your application's Project Object Model (POM) file. In this example, replace `x.x.x` with the actual version number of the client.

```
<!--Dependency:-->  
<dependencies>  
    <dependency>
```

```
<groupId>software.amazon.dax</groupId>
<artifactId>amazon-dax-client</artifactId>
<version>x.x.x</version>
</dependency>
</dependencies>
```

TryDax sample code

After you've set up your workspace and added the DAX SDK as a dependency, copy [TryDax.java](#) into your project.

Run the code using this command.

```
java -cp classpath TryDax
```

You should see output similar to the following.

```
Creating a DynamoDB client
```

```
Attempting to create table; please wait...
Successfully created table. Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
...
```

```
Running GetItem and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits
```

```
GetItem test - partition key 1-100 and sort keys 1-10
Total time: 4390.240 ms - Avg time: 4.390 ms
Total time: 3097.089 ms - Avg time: 3.097 ms
Total time: 3273.463 ms - Avg time: 3.273 ms
Total time: 3353.739 ms - Avg time: 3.354 ms
Total time: 3533.314 ms - Avg time: 3.533 ms
Query test - partition key 1-100 and sort keys between 2 and 9
Total time: 475.868 ms - Avg time: 4.759 ms
Total time: 423.333 ms - Avg time: 4.233 ms
Total time: 460.271 ms - Avg time: 4.603 ms
Total time: 397.859 ms - Avg time: 3.979 ms
```

```
Total time: 466.644 ms - Avg time: 4.666 ms
```

```
Attempting to delete table; please wait...
Successfully deleted table.
```

Take note of the timing information—the number of milliseconds required for the GetItem and Query tests. In this case, you ran the program against the DynamoDB endpoint. Now you'll run the program again, this time against your DAX cluster.

To determine the endpoint of your DAX cluster, choose one of the following:

- In the DynamoDB console, select your DAX cluster. The cluster endpoint is shown in the console, as in the following example.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

- Using the AWS CLI, enter the following command:

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint address, port, and URL are shown in the output, as in the following example.

```
{
  "Address": "my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com",
  "Port": 8111,
  "URL": "dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

Now run the program again, but this time, specify the cluster endpoint URL as a command line parameter.

```
java -cp classpath TryDax dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

Look at the output and take note of the timing information. The elapsed times for GetItem and Query should be significantly lower with DAX than with DynamoDB.

SDK metrics

With DAX SDK for Java 2.x, you can collect metrics about the service clients in your application and analyze the output in Amazon CloudWatch. See [Enabling SDK metrics](#) for more information.

 **Note**

The DAX SDK for Java only collects `ApiCallSuccessful` and `ApiCallDuration` metrics.

TryDax.java

```
import java.util.Map;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.dax.ClusterDaxAsyncClient;
import software.amazon.dax.Configuration;

public class TryDax {
    public static void main(String[] args) throws Exception {
        DynamoDbAsyncClient ddbClient = DynamoDbAsyncClient.builder()
            .build();

        DynamoDbAsyncClient daxClient = null;
        if (args.length >= 1) {
            daxClient = ClusterDaxAsyncClient.builder()
                .overrideConfiguration(Configuration.builder()
                    .url(args[0]) // e.g. dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com
                .build())
            .build();
        }
    }
}
```

```
}

String tableName = "TryDaxTable";

System.out.println("Creating table...");
createTable(tableName, ddbClient);

System.out.println("Populating table...");
writeData(tableName, ddbClient, 100, 10);

DynamoDbAsyncClient testClient = null;
if (daxClient != null) {
    testClient = daxClient;
} else {
    testClient = ddbClient;
}

System.out.println("Running GetItem and Query tests...");
System.out.println("First iteration of each test will result in cache misses");
System.out.println("Next iterations are cache hits\n");

// GetItem
getItemTest(tableName, testClient, 100, 10, 5);

// Query
queryTest(tableName, testClient, 100, 2, 9, 5);

System.out.println("Deleting table...");
deleteTable(tableName, ddbClient);
}

private static void createTable(String tableName, DynamoDbAsyncClient client) {
    try {
        System.out.println("Attempting to create table; please wait...");

        client.createTable(CreateTableRequest.builder()
            .tableName(tableName)
            .keySchema(KeySchemaElement.builder()
                .keyType(KeyType.HASH)
                .attributeName("pk")
                .build(), KeySchemaElement.builder()
                .keyType(KeyType.RANGE)
                .attributeName("sk")
                .build())
            .build())
    }
}
```

```
        .attributeDefinitions(AttributeDefinition.builder()
            .attributeName("pk")
            .attributeType(ScalarAttributeType.N)
            .build(), AttributeDefinition.builder()
            .attributeName("sk")
            .attributeType(ScalarAttributeType.N)
            .build())
        .billingMode(BillingMode.PAY_PER_REQUEST)
        .build()).get();
    client.waiter().waitUntilTableExists(DescribeTableRequest.builder()
        .tableName(tableName)
        .build()).get();
    System.out.println("Successfully created table.");

} catch (Exception e) {
    System.err.println("Unable to create table: ");
    e.printStackTrace();
}
}

private static void deleteTable(String tableName, DynamoDbAsyncClient client) {
try {
    System.out.println("\nAttempting to delete table; please wait...");
    client.deleteTable(DeleteTableRequest.builder()
        .tableName(tableName)
        .build()).get();
    client.waiter().waitUntilTableNotExists(DescribeTableRequest.builder()
        .tableName(tableName)
        .build()).get();
    System.out.println("Successfully deleted table.");

} catch (Exception e) {
    System.err.println("Unable to delete table: ");
    e.printStackTrace();
}
}

private static void writeData(String tableName, DynamoDbAsyncClient client, int
pkmax, int skmax) {
    System.out.println("Writing data to the table...");

    int stringSize = 1000;
    StringBuilder sb = new StringBuilder(stringSize);
    for (int i = 0; i < stringSize; i++) {
```

```
        sb.append('X');
    }
    String someData = sb.toString();

    try {
        for (int ipk = 1; ipk <= pkmax; ipk++) {
            System.out.println("Writing " + skmax + " items for partition key: " +
ipk);
            for (int isk = 1; isk <= skmax; isk++) {
                client.putItem(PutItemRequest.builder()
                    .tableName(tableName)
                    .item(Map.of("pk", attr(ipk), "sk", attr(isk), "someData",
attr(someData)))
                    .build()).get();
            }
        }
    } catch (Exception e) {
        System.err.println("Unable to write item:");
        e.printStackTrace();
    }
}

private static AttributeValue attr(int n) {
    return AttributeValue.builder().n(String.valueOf(n)).build();
}

private static AttributeValue attr(String s) {
    return AttributeValue.builder().s(s).build();
}

private static void getItemTest(String tableName, DynamoDbAsyncClient client, int
pk, int sk, int iterations) {
    long startTime, endTime;
    System.out.println("GetItem test - partition key 1-" + pk + " and sort keys 1-"
+ sk);

    for (int i = 0; i < iterations; i++) {
        startTime = System.nanoTime();
        try {
            for (int ipk = 1; ipk <= pk; ipk++) {
                for (int isk = 1; isk <= sk; isk++) {
                    client.getItem(GetItemRequest.builder()
                        .tableName(tableName)
                        .key(Map.of("pk", attr(ipk), "sk", attr(isk))))
```

```
        .build()).get();
    }
}
} catch (Exception e) {
    System.err.println("Unable to get item:");
    e.printStackTrace();
}
endTime = System.nanoTime();
printTime(startTime, endTime, pk * sk);
}
}

private static void queryTest(String tableName, DynamoDbAsyncClient client, int pk,
int sk1, int sk2, int iterations) {
    long startTime, endTime;
    System.out.println("Query test - partition key 1-" + pk + " and sort keys
between " + sk1 + " and " + sk2);

    for (int i = 0; i < iterations; i++) {
        startTime = System.nanoTime();
        for (int ipk = 1; ipk <= pk; ipk++) {
            try {
                // Pagination API for Query.
                client.queryPaginator(QueryRequest.builder()
                    .tableName(tableName)
                    .keyConditionExpression("pk = :pkval and sk between :skval1
and :skval2")
                    .expressionAttributeValues(Map.of(":pkval", attr(ipk),
":skval1", attr(sk1), ":skval2", attr(sk2)))
                    .build()).items().subscribe((item) -> {
                }).get();
            } catch (Exception e) {
                System.err.println("Unable to query table:");
                e.printStackTrace();
            }
        }
        endTime = System.nanoTime();
        printTime(startTime, endTime, pk);
    }
}

private static void printTime(long startTime, long endTime, int iterations) {
    System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) /
(1000000.0));
```

```
        System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *  
1000000.0));  
    }  
}
```

.NET and DAX

Follow these steps to run the .NET sample on your Amazon EC2 instance.

Note

This tutorial uses the .NET 6 SDK, but will work with the .NET Core SDK also. It shows how you can run a program in your default Amazon VPC to access your Amazon DynamoDB Accelerator (DAX) cluster. If you prefer, you can use the AWS Toolkit for Visual Studio to write a .NET application and deploy it into your VPC.

For more information, see [Creating and Deploying Elastic Beanstalk Applications in .NET Using AWS Toolkit for Visual Studio](#) in the *AWS Elastic Beanstalk Developer Guide*.

To run the .NET sample for DAX

1. Go to the [Microsoft Downloads page](#) and download the latest .NET 6 (or .NET Core) SDK for Linux. The downloaded file is dotnet-sdk-**N.N.N**-linux-x64.tar.gz.
2. Extract the SDK files.

```
mkdir dotnet  
tar zxvf dotnet-sdk-N.N.N-linux-x64.tar.gz -C dotnet
```

Replace **N.N.N** with the actual version number of the .NET SDK (for example: 6.0.100).

3. Verify the installation.

```
alias dotnet=$HOME/dotnet/dotnet  
dotnet --version
```

This should print the version number of the .NET SDK.

Note

Instead of the version number, you might receive the following error:

error: libunwind.so.8: cannot open shared object file: No such file or directory
To resolve the error, install the libunwind package.

```
sudo yum install -y libunwind
```

After you do this, you should be able to run the dotnet --version command without any errors.

4. Create a new .NET project.

```
dotnet new console -o myApp
```

This requires a few minutes to perform a one-time-only setup. When it is complete, run the sample project.

```
dotnet run --project myApp
```

You should receive the following message: Hello World!

5. The myApp/myApp.csproj file contains metadata about your project. To use the DAX client in your application, modify the file so that it looks like the following.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="AWSSDK.DAX.Client" Version="*" />
  </ItemGroup>
</Project>
```

6. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

7. Now run the sample programs, one at a time. For each program, copy its contents into the `myApp/Program.cs`, and then run the `MyApp` project.

Run the following .NET programs. The first program creates a DynamoDB table named `TryDaxTable`. The second program writes data to the table.

```
cp TryDax/dotNet/01-CreateTable.cs myApp/Program.cs  
dotnet run --project MyApp
```

```
cp TryDax/dotNet/02-Write-Data.cs myApp/Program.cs  
dotnet run --project MyApp
```

8. Next, run some programs to perform `GetItem`, `Query`, and `Scan` operations on your DAX cluster. To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint is shown in the output, as in the following example.

```
{  
    "Address": "my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com",  
    "Port": 8111,  
    "URL": "dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com"  
}
```

Now run the following programs, specifying your cluster endpoint as a command line parameter. (Replace the sample endpoint with your actual DAX cluster endpoint.)

```
cp TryDax/dotNet/03-GetItem-Test.cs myApp/Program.cs
```

```
dotnet run --project myApp dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com

cp TryDax/dotNet/04-Query-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com

cp TryDax/dotNet/05-Scan-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

Take note of the timing information—the number of milliseconds required for the GetItem, Query, and Scan tests.

9. Run the following .NET program to delete TryDaxTable.

```
cp TryDax/dotNet/06-DeleteTable.cs myApp/Program.cs
dotnet run --project myApp
```

For more information about these programs, see the following sections:

- [01-CreateTable.cs](#)
- [02-Write-Data.cs](#)
- [03-GetItem-Test.cs](#)
- [04-Query-Test.cs](#)
- [05-Scan-Test.cs](#)
- [06-DeleteTable.cs](#)

01-CreateTable.cs

The `01-CreateTable.cs` program creates a table (TryDaxTable). The remaining .NET programs in this section depend on this table.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
```

```
namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            var request = new CreateTableRequest()
            {
                TableName = tableName,
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement{ AttributeName = "pk",KeyType = "HASH" },
                    new KeySchemaElement{ AttributeName = "sk",KeyType = "RANGE" }
                },
                AttributeDefinitions = new List<AttributeDefinition>() {
                    new AttributeDefinition{ AttributeName = "pk",AttributeType = "N" },
                    new AttributeDefinition{ AttributeName = "sk",AttributeType = "N" }
                },
                ProvisionedThroughput = new ProvisionedThroughput()
                {
                    ReadCapacityUnits = 10,
                    WriteCapacityUnits = 10
                }
            };
            var response = await client.CreateTableAsync(request);

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

02-Write-Data.cs

The 02-Write-Data.cs program writes test data to TryDaxTable.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            string someData = new string('X', 1000);
            var pkmax = 10;
            var skmax = 10;

            for (var ipk = 1; ipk <= pkmax; ipk++)
            {
                Console.WriteLine($"Writing {skmax} items for partition key: {ipk}");
                for (var isk = 1; isk <= skmax; isk++)
                {
                    var request = new PutItemRequest()
                    {
                        TableName = tableName,
                        Item = new Dictionary<string, AttributeValue>()
                        {
                            { "pk", new AttributeValue{N = ipk.ToString()} },
                            { "sk", new AttributeValue{N = isk.ToString()} },
                            { "someData", new AttributeValue{S = someData} }
                        }
                    };
                    var response = await client.PutItemAsync(request);
                }
            }

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

```
}
```

03-GetItem-Test.cs

The `03-GetItem-Test.cs` program performs `GetItem` operations on `TryDaxTable`.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DAX;
using Amazon.DynamoDbv2.Model;
using Amazon.Runtime;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");

            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var pk = 1;
            var sk = 10;
            var iterations = 5;

            var startTime = System.DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                for (var ipk = 1; ipk <= pk; ipk++)
                {
                    for (var isk = 1; isk <= sk; isk++)
                    {
```

```
var request = new GetItemRequest()
{
    TableName = tableName,
    Key = new Dictionary<string, AttributeValue>() {
        {"pk", new AttributeValue {N = ipk.ToString()} },
        {"sk", new AttributeValue {N = isk.ToString()} }
    }
};

var response = await client.GetItemAsync(request);
Console.WriteLine($"GetItem succeeded for pk: {ipk},sk: {isk}");
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}

var endTime = DateTime.Now;
TimeSpan timeSpan = endTime - startTime;
Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
milliseconds");

Console.WriteLine("Hit <enter> to continue...");
Console.ReadLine();
}
```

```
}
```

```
}
```

04-Query-Test.cs

The 04-Query-Test.cs program performs Query operations on TryDaxTable.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Runtime;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
```

```
{  
    string endpointUri = args[0];  
    Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");  
  
    var clientConfig = new DaxClientConfig(endpointUri)  
    {  
        AwsCredentials = FallbackCredentialsFactory.GetCredentials()  
    };  
    var client = new ClusterDaxClient(clientConfig);  
  
    var tableName = "TryDaxTable";  
  
    var pk = 5;  
    var sk1 = 2;  
    var sk2 = 9;  
    var iterations = 5;  
  
    var startTime = DateTime.Now;  
  
    for (var i = 0; i < iterations; i++)  
    {  
        var request = new QueryRequest()  
        {  
            TableName = tableName,  
            KeyConditionExpression = "pk = :pkval and sk between :skval1  
and :skval2",  
            ExpressionAttributeValues = new Dictionary<string,  
AttributeValue>() {  
                {"":pkval", new AttributeValue {N = pk.ToString()} },  
                {"":skval1", new AttributeValue {N = sk1.ToString()} },  
                {"":skval2", new AttributeValue {N = sk2.ToString()} }  
            }  
        };  
        var response = await client.QueryAsync(request);  
        Console.WriteLine($"{i}: Query succeeded");  
  
    }  
  
    var endTime = DateTime.Now;  
    TimeSpan timeSpan = endTime - startTime;  
    Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}  
milliseconds");  
}
```

```
        Console.WriteLine("Hit <enter> to continue...");  
        Console.ReadLine();  
    }  
}  
}
```

05-Scan-Test.cs

The `05-Scan-Test.cs` program performs Scan operations on TryDaxTable.

```
using System;  
using System.Threading.Tasks;  
using Amazon.Runtime;  
using Amazon.DAX;  
using Amazon.DynamoDBv2.Model;  
  
namespace ClientTest  
{  
    class Program  
    {  
        public static async Task Main(string[] args)  
        {  
            string endpointUri = args[0];  
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");  
  
            var clientConfig = new DaxClientConfig(endpointUri)  
            {  
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()  
            };  
            var client = new ClusterDaxClient(clientConfig);  
  
            var tableName = "TryDaxTable";  
  
            var iterations = 5;  
  
            var startTime = DateTime.Now;  
  
            for (var i = 0; i < iterations; i++)  
            {  
                var request = new ScanRequest()  
                {  
                    TableName = tableName  
                };  
                var response = await client.ScanAsync(request);  
                var items = response.Items;  
                if (items != null)  
                {  
                    foreach (var item in items)  
                    {  
                        var id = item["id"];  
                        var name = item["name"];  
                        var age = item["age"];  
                        var email = item["email"];  
                        Console.WriteLine($"ID: {id}, Name: {name}, Age: {age}, Email: {email}");  
                    }  
                }  
            }  
        }  
    }  
}
```

```
    };

    var response = await client.ScanAsync(request);
    Console.WriteLine($"[{i}]: Scan succeeded");
}

var endTime = DateTime.Now;
TimeSpan timeSpan = endTime - startTime;
Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
milliseconds");

Console.WriteLine("Hit <enter> to continue...");
Console.ReadLine();
}
}

}
```

06-DeleteTable.cs

The 06-DeleteTable.cs program deletes TryDaxTable. Run this program after you have finished testing.

```
using System;
using System.Threading.Tasks;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            var request = new DeleteTableRequest()
            {
                TableName = tableName
            };
        }
    }
}
```

```
        var response = await client.DeleteTableAsync(request);

        Console.WriteLine("Hit <enter> to continue...");
        Console.ReadLine();
    }
}
```

Node.js and DAX

Follow these steps to run the Node.js sample application on your Amazon EC2 instance.

To run the Node.js sample for DAX

1. Set up Node.js on your Amazon EC2 instance, as follows:

- a. Install node version manager (nvm).

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

- b. Use nvm to install Node.js.

```
nvm install 12.16.3
```

- c. Test that Node.js is installed and running correctly.

```
node -e "console.log('Running Node.js ' + process.version)"
```

This should display the following message.

Running Node.js v12.16.3

2. Install the DAX Node.js client using the node package manager (npm).

```
npm install amazon-dax-client
```

3. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

- Run the following Node.js programs. The first program creates an Amazon DynamoDB table named TryDaxTable. The second program writes data to the table.

```
node 01-create-table.js  
node 02-write-data.js
```

- Run the following Node.js programs.

```
node 03-getitem-test.js  
node 04-query-test.js  
node 05-scan-test.js
```

Take note of the timing information—the number of milliseconds required for the GetItem, Query, and Scan tests.

- In the previous step, you ran the programs against the DynamoDB endpoint. Run the programs again, but this time, the GetItem, Query and Scan operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following.

- Using the DynamoDB console**—Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

- Using the AWS CLI**—Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint is shown in the output, as in the following example.

```
{  
    "Address": "my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com",  
    "Port": 8111,  
    "URL": "dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com"
```

{}

Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
node 03-getitem-test.js dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com  
node 04-query-test.js dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com  
node 05-scan-test.js dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

Look at the rest of the output, and take note of the timing information. The elapsed times for GetItem, Query, and Scan should be significantly lower with DAX than with DynamoDB.

7. Run the following Node.js program to delete TryDaxTable.

```
node 06-delete-table
```

For more information about these programs, see the following sections:

- [01-create-table.js](#)
- [02-write-data.js](#)
- [03-getitem-test.js](#)
- [04-query-test.js](#)
- [05-scan-test.js](#)
- [06-delete-table.js](#)

01-create-table.js

The `01-create-table.js` program creates a table (TryDaxTable). The remaining Node.js programs in this section depend on this table.

```
const AmazonDaxClient = require("amazon-dax-client");  
var AWS = require("aws-sdk");  
  
var region = "us-west-2";  
  
AWS.config.update({
```

```
    region: region,
});

var dynamodb = new AWS.DynamoDB(); //low-level client

var tableName = "TryDaxTable";

var params = {
  TableName: tableName,
  KeySchema: [
    { AttributeName: "pk", KeyType: "HASH" }, //Partition key
    { AttributeName: "sk", KeyType: "RANGE" } //Sort key
  ],
  AttributeDefinitions: [
    { AttributeName: "pk", AttributeType: "N" },
    { AttributeName: "sk", AttributeType: "N" },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 10,
    WriteCapacityUnits: 10,
  },
};

dynamodb.createTable(params, function (err, data) {
  if (err) {
    console.error(
      "Unable to create table. Error JSON:",
      JSON.stringify(err, null, 2)
    );
  } else {
    console.log(
      "Created table. Table description JSON:",
      JSON.stringify(data, null, 2)
    );
  }
});
```

02-write-data.js

The `02-write-data.js` program writes test data to `TryDaxTable`.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");
```

```
var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();

var tableName = "TryDaxTable";

var someData = "X".repeat(1000);
var pkmax = 10;
var skmax = 10;

for (var ipk = 1; ipk <= pkmax; ipk++) {
  for (var isk = 1; isk <= skmax; isk++) {
    var params = {
      TableName: tableName,
      Item: {
        pk: ipk,
        sk: isk,
        someData: someData,
      },
    };
    //put item
    ddbClient.put(params, function (err, data) {
      if (err) {
        console.error("Unable to write data: ", JSON.stringify(err, null, 2));
      } else {
        console.log("PutItem succeeded");
      }
    });
  }
}
```

03-getitem-test.js

The `03-getitem-test.js` program performs GetItem operations on TryDaxTable.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 1;
var sk = 10;
var iterations = 5;

for (var i = 0; i < iterations; i++) {
  var startTime = new Date().getTime();

  for (var ipk = 1; ipk <= pk; ipk++) {
    for (var isk = 1; isk <= sk; isk++) {
      var params = {
        TableName: tableName,
        Key: {
          pk: ipk,
          sk: isk,
        },
      };

      client.get(params, function (err, data) {
        if (err) {
          console.error(
            `Error performing GET on item ${ipk}#${isk}: ${err.message}`
          );
        } else {
          console.log(`Item ${ipk}#${isk} retrieved successfully`);
        }
      });
    }
  }
}
```

```
        "Unable to read item. Error JSON:",
        JSON.stringify(err, null, 2)
    );
} else {
    // GetItem succeeded
}
});
}

var endTime = new Date().getTime();
console.log(
"\tTotal time: ",
endTime - startTime,
"ms - Avg time: ",
(endTime - startTime) / iterations,
"ms"
);
}
```

04-query-test.js

The `04-query-test.js` program performs Query operations on TryDaxTable.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}
```

```
var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 5;
var sk1 = 2;
var sk2 = 9;
var iterations = 5;

var params = {
    TableName: tableName,
    KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
    ExpressionAttributeValues: {
        ":pkval": pk,
        ":skval1": sk1,
        ":skval2": sk2,
    },
};

for (var i = 0; i < iterations; i++) {
    var startTime = new Date().getTime();

    client.query(params, function (err, data) {
        if (err) {
            console.error(
                "Unable to read item. Error JSON:",
                JSON.stringify(err, null, 2)
            );
        } else {
            // Query succeeded
        }
    });
}

var endTime = new Date().getTime();
console.log(
    "\tTotal time: ",
    endTime - startTime,
    "ms - Avg time: ",
    (endTime - startTime) / iterations,
    "ms"
);
}
```

05-scan-test.js

The `05-scan-test.js` program performs Scan operations on TryDaxTable.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var iterations = 5;

var params = {
  TableName: tableName,
};

var startTime = new Date().getTime();
for (var i = 0; i < iterations; i++) {
  client.scan(params, function (err, data) {
    if (err) {
      console.error(
        "Unable to read item. Error JSON:",
        JSON.stringify(err, null, 2)
      );
    } else {
      // Scan succeeded
    }
  });
}
```

```
}

var endTime = new Date().getTime();
console.log(
  "\tTotal time: ",
  endTime - startTime,
  "ms - Avg time: ",
  (endTime - startTime) / iterations,
  "ms"
);
}
```

06-delete-table.js

The `06-delete-table.js` program deletes `TryDaxTable`. Run this program after you have finished testing.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var dynamodb = new AWS.DynamoDB(); //low-level client

var tableName = "TryDaxTable";

var params = {
  TableName: tableName,
};

dynamodb.deleteTable(params, function (err, data) {
  if (err) {
    console.error(
      "Unable to delete table. Error JSON:",
      JSON.stringify(err, null, 2)
    );
  } else {
    console.log(
      "Deleted table. Table description JSON:",
      JSON.stringify(data, null, 2)
    );
  }
});
```

```
    JSON.stringify(data, null, 2)
);
}
});
```

Python and DAX

Follow this procedure to run the Python sample application on your Amazon EC2 instance.

To run the Python sample for DAX

1. Install the DAX Python client using the pip utility.

```
pip install amazon-dax-client
```

2. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

3. Run the following Python programs. The first program creates an Amazon DynamoDB table named TryDaxTable. The second program writes data to the table.

```
python 01-create-table.py
python 02-write-data.py
```

4. Run the following Python programs.

```
python 03-getitem-test.py
python 04-query-test.py
python 05-scan-test.py
```

Take note of the timing information—the number of milliseconds required for the GetItem, Query, and Scan tests.

5. In the previous step, you ran the programs against the DynamoDB endpoint. Now run the programs again, but this time, the GetItem, Query, and Scan operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint is shown in the output, as in this example.

```
{  
    "Address": "my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com",  
    "Port": 8111,  
    "URL": "dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com"  
}
```

Run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
python 03-getitem-test.py dax://my-cluster.16fzcv.dax-clusters.us-  
east-1.amazonaws.com  
python 04-query-test.py dax://my-cluster.16fzcv.dax-clusters.us-  
east-1.amazonaws.com  
python 05-scan-test.py dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

Look at the rest of the output, and take note of the timing information. The elapsed times for GetItem, Query, and Scan should be significantly lower with DAX than with DynamoDB.

6. Run the following Python program to delete TryDaxTable.

```
python 06-delete-table.py
```

For more information about these programs, see the following sections:

- [01-create-table.py](#)
- [02-write-data.py](#)
- [03-getitem-test.py](#)
- [04-query-test.py](#)
- [05-scan-test.py](#)
- [06-delete-table.py](#)

01-create-table.py

The `01-create-table.py` program creates a table (`TryDaxTable`). The remaining Python programs in this section depend on this table.

```
import boto3

def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table_name = "TryDaxTable"
    params = {
        "TableName": table_name,
        "KeySchema": [
            {"AttributeName": "partition_key", "KeyType": "HASH"},
            {"AttributeName": "sort_key", "KeyType": "RANGE"},
        ],
        "AttributeDefinitions": [
            {"AttributeName": "partition_key", "AttributeType": "N"},
            {"AttributeName": "sort_key", "AttributeType": "N"},
        ],
        "ProvisionedThroughput": {"ReadCapacityUnits": 10, "WriteCapacityUnits": 10},
    }
    table = dyn_resource.create_table(**params)
```

```
print(f"Creating {table_name}...")
table.wait_until_exists()
return table

if __name__ == "__main__":
    dax_table = create_dax_table()
    print(f"Created table.")
```

02-write-data.py

The `02-write-data.py` program writes test data to TryDaxTable.

```
import boto3

def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate the
                      table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    some_data = "X" * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(
                Item={
                    "partition_key": partition_key,
                    "sort_key": sort_key,
                    "some_data": some_data,
                }
            )
            print(f"Put item ({partition_key}, {sort_key}) succeeded.")

    if __name__ == "__main__":
```

```
write_key_count = 10
write_item_size = 1000
print(
    f"Writing {write_key_count}*{write_key_count} items to the table. "
    f"Each item is {write_item_size} characters."
)
write_data_to_dax_table(write_key_count, write_item_size)
```

03-getitem-test.py

The `03-getitem-test.py` program performs GetItem operations on TryDaxTable. This example is given for the Region eu-west-1.

```
import argparse
import sys
import time
import amazondax
import boto3

def get_item_test(key_count, iterations, dyn_resource=None):
    """
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each iteration.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource('dynamodb')

    table = dyn_resource.Table('TryDaxTable')
    start = time.perf_counter()
    for _ in range(iterations):
        for partition_key in range(1, key_count + 1):
            for sort_key in range(1, key_count + 1):
                table.get_item(Key={
                    'partition_key': partition_key,
                    'sort_key': sort_key
                })
```

```
        })
        print('..', end='')
        sys.stdout.flush()

print()
end = time.perf_counter()
return start, end

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'endpoint_url', nargs='?',
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.")
args = parser.parse_args()

test_key_count = 10
test_iterations = 50
if args.endpoint_url:
    print(f"Getting each item from the table {test_iterations} times, "
          f"using the DAX client.")
    # Use a with statement so the DAX client closes the cluster after completion.
    with amazonadax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url,
region_name='eu-west-1') as dax:
        test_start, test_end = get_item_test(
            test_key_count, test_iterations, dyn_resource=dax)
else:
    print(f"Getting each item from the table {test_iterations} times, "
          f"using the Boto3 client.")
    test_start, test_end = get_item_test(
        test_key_count, test_iterations)
print(f"Total time: {test_end - test_start:.4f} sec. Average time: "
      f"{{(test_end - test_start) / test_iterations}}")
```

04-query-test.py

The `04-query-test.py` program performs Query operations on TryDaxTable.

```
import argparse
import time
import sys
import amazonadax
import boto3
from boto3.dynamodb.conditions import Key
```

```
def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param partition_key: The partition key value to use in the query. The query
                          returns items that have partition keys equal to this value.
    :param sort_keys: The range of sort key values for the query. The query returns
                      items that have sort key values between these two values.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """

    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    key_condition_expression = Key("partition_key").eq(partition_key) & Key(
        "sort_key"
    ).between(*sort_keys)

    start = time.perf_counter()
    for _ in range(iterations):
        table.query(KeyConditionExpression=key_condition_expression)
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end

if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()
```

```
test_partition_key = 5
test_sort_keys = (2, 9)
test_iterations = 100
if args.endpoint_url:
    print(f"Querying the table {test_iterations} times, using the DAX client.")
    # Use a with statement so the DAX client closes the cluster after completion.
    with amazonadax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations, dyn_resource=dax
        )
else:
    print(f"Querying the table {test_iterations} times, using the Boto3 client.")
    test_start, test_end = query_test(
        test_partition_key, test_sort_keys, test_iterations
    )

print(
    f"Total time: {test_end - test_start:.4f} sec. Average time: "
    f"{(test_end - test_start)/test_iterations}."
)
```

05-scan-test.py

The `05-scan-test.py` program performs Scan operations on TryDaxTable.

```
import argparse
import time
import sys
import amazonadax
import boto3

def scan_test(iterations, dyn_resource=None):
    """
    Scans the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")
```

```
table = dyn_resource.Table("TryDaxTable")
start = time.perf_counter()
for _ in range(iterations):
    table.scan()
    print(".", end="")
    sys.stdout.flush()
print()
end = time.perf_counter()
return start, end

if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_iterations = 100
    if args.endpoint_url:
        print(f"Scanning the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = scan_test(test_iterations, dyn_resource=dax)
    else:
        print(f"Scanning the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = scan_test(test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{{(test_end - test_start)/test_iterations}}."
    )
```

06-delete-table.py

The `06-delete-table.py` program deletes `TryDaxTable`. Run this program after you have finished testing Amazon DynamoDB Accelerator (DAX) functionality.

```
import boto3
```

```
def delete_dax_table(dyn_resource=None):
    """
    Deletes the demonstration table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    table.delete()

    print(f"Deleting {table.name}...")
    table.wait_until_not_exists()

if __name__ == "__main__":
    delete_dax_table()
    print("Table deleted!")
```

Modifying an existing application to use DAX

If you already have a Java application that uses Amazon DynamoDB, you can modify it so that it can access your DynamoDB Accelerator (DAX) cluster. You don't have to rewrite the entire application because the DAX Java client is similar to the DynamoDB low-level client included in the AWS SDK for Java 2.x. See [Working with items in DynamoDB](#) for details.

 **Note**

This example uses AWS SDK for Java 2.x. For the legacy SDK for Java 1.x version, see [Modifying an existing SDK for Java 1.x application to use DAX](#).

To modify your program, replace the DynamoDB client with a DAX client.

```
Region region = Region.US_EAST_1;

// Create an asynchronous DynamoDB client
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .region(region)
    .build();
```

```
// Create an asynchronous DAX client
DynamoDbAsyncClient client = ClusterDaxAsyncClient.builder()
    .overrideConfiguration(Configuration.builder()
        .url(<cluster url>) // for example, "dax://my-cluster.16fzcv.dax-
clusters.us-east-1.amazonaws.com"
        .region(region)
        .addMetricPublisher(cloudWatchMetricsPub) // optionally enable SDK
metric collection
    .build())
    .build();
```

You can also use the high-level library that is part of the AWS SDK for Java 2.x, replacing the DynamoDB client with a DAX client.

```
Region region = Region.US_EAST_1;
DynamoDbAsyncClient dax = ClusterDaxAsyncClient.builder()
    .overrideConfiguration(Configuration.builder()
        .url(<cluster url>) // for example, "dax://my-cluster.16fzcv.dax-
clusters.us-east-1.amazonaws.com"
        .region(region)
    .build())
    .build();

DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
    .dynamoDbClient(dax)
    .build();
```

For more information, see [Mapping items in DynamoDB tables](#).

Managing DAX clusters

This section addresses some of the common management tasks for Amazon DynamoDB Accelerator (DAX) clusters.

Topics

- [IAM permissions for managing a DAX cluster](#)
- [Scaling a DAX cluster](#)
- [Customizing DAX cluster settings](#)
- [Configuring TTL settings](#)

- [Tagging support for DAX](#)
- [AWS CloudTrail integration](#)
- [Deleting a DAX cluster](#)

IAM permissions for managing a DAX cluster

When you administer a DAX cluster using the AWS Management Console or the AWS Command Line Interface (AWS CLI), we strongly recommend that you narrow the scope of actions that users can perform. By doing so, you help mitigate risk while following the principle of least privilege.

The following discussion focuses on access control for the DAX management APIs. For more information, see [Amazon DynamoDB accelerator](#) in the *Amazon DynamoDB API Reference*.

Note

For more detailed information about managing AWS Identity and Access Management (IAM) permissions, see the following:

- IAM and creating DAX clusters: [Creating a DAX cluster](#).
- IAM and DAX data plane operations: [DAX access control](#).

For the DAX management APIs, you can't scope API actions to a specific resource. The `Resource` element must be set to `"*"`. This is different from DAX data plane API operations, such as `GetItem`, `Query`, and `Scan`. Data plane operations are exposed through the DAX client, and those operations *can* be scoped to specific resources.

To illustrate, consider the following IAM policy document.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dax:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"  
            ]  
        }  
    ]  
}
```

```
        ]
    }
]
}
```

Suppose that the intent of this policy is to allow DAX management API calls for the cluster `DAXCluster01`— and only that cluster.

Now suppose that a user issues the following AWS CLI command.

```
aws dax describe-clusters
```

This command fails with a Not Authorized exception because the underlying `DescribeClusters` API call can't be scoped to a specific cluster. Even though the policy is syntactically valid, the command fails because the `Resource` element must be set to `"*"`. However, if the user runs a program that sends DAX data plane calls (such as `GetItem` or `Query`) to `DAXCluster01`, those calls *do* succeed. This is because DAX data plane APIs can be scoped to specific resources (in this case, `DAXCluster01`).

If you want to write a single comprehensive IAM policy to encompass both DAX management APIs and DAX data plane APIs, we suggest that you include two distinct statements in the policy document. One of these statements should address the DAX data plane APIs, while the other statement addresses the DAX management APIs.

The following example policy shows this approach. Note how the `DAXDataAPIs` statement is scoped to the `DAXCluster01` resource, but the resource for `DAXManagementAPIs` must be `"*"`. The actions shown in each statement are for illustration only. You can customize them as needed for your application.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXDataAPIs",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
            ],
            "Resource": "arn:aws:dax:DAXCluster01"
        },
        {
            "Sid": "DAXManagementAPIs",
            "Action": [
                "dax:DescribeClusters"
            ],
            "Resource": "*"
        }
    ]
}
```

```
        "dax:DeleteItem",
        "dax:BatchWriteItem"
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
    ],
{
    "Sid": "DAXManagementAPIs",
    "Action": [
        "dax>CreateParameterGroup",
        "dax>CreateSubnetGroup",
        "dax>DecreaseReplicationFactor",
        "dax>DeleteCluster",
        "dax>DeleteParameterGroup",
        "dax>DeleteSubnetGroup",
        "dax>DescribeClusters",
        "dax>DescribeDefaultParameters",
        "dax>DescribeEvents",
        "dax>DescribeParameterGroups",
        "dax>DescribeParameters",
        "dax>DescribeSubnetGroups",
        "dax>IncreaseReplicationFactor",
        "dax>ListTags",
        "dax>RebootNode",
        "dax>TagResource",
        "dax>UntagResource",
        "dax>UpdateCluster",
        "dax>UpdateParameterGroup",
        "dax>UpdateSubnetGroup"
    ],
    "Effect": "Allow",
    "Resource": [
        "*"
    ]
}
]
```

Scaling a DAX cluster

There are two options available for scaling a DAX cluster. The first option is *horizontal scaling*, where you add read replicas to the cluster. The second option is *vertical scaling*, where you select

different node types. For advice on how to approach choosing an appropriate cluster size and node type for your application, see [DAX cluster sizing guide](#).

Horizontal scaling

With horizontal scaling, you can improve throughput for read operations by adding more read replicas to the cluster. A single DAX cluster supports up to 10 read replicas, and you can add or remove replicas while the cluster is running.

The following AWS CLI examples show how to increase or decrease the number of nodes. The `--new-replication-factor` argument specifies the total number of nodes in the cluster. One of the nodes is the primary node, and the other nodes are read replicas.

```
aws dax increase-replication-factor \
--cluster-name MyNewCluster \
--new-replication-factor 5
```

```
aws dax decrease-replication-factor \
--cluster-name MyNewCluster \
--new-replication-factor 3
```

Note

The cluster status changes to `modifying` when you modify the replication factor. The status changes to `available` when the modification is complete.

Vertical scaling

If you have a large working set of data, your application might benefit from using larger node types. Larger nodes can enable the cluster to store more data in memory, reducing cache misses and improving overall application performance of the application. (All of the nodes in a DAX cluster must be of the same type.)

If your DAX cluster has a high rate of write operations or cache misses, your application might also benefit from using larger node types. Write operations and cache misses consume resources on the cluster's primary node. Therefore, using larger node types might increase the performance of the primary node and thereby allow a higher throughput for these types of operations.

You can't modify the node types on a running DAX cluster. Instead, you must create a new cluster with the desired node type. For a list of supported node types, see [Nodes](#).

You can create a new DAX cluster using the AWS Management Console, [AWS CloudFormation](#), the AWS CLI, or the [AWS SDK](#). (For the AWS CLI, use the `--node-type` parameter to specify the node type.)

Customizing DAX cluster settings

When you create a DAX cluster, the following default settings are used:

- Automatic cache eviction enabled with Time to Live (TTL) of 5 minutes
- No preference for Availability Zones
- No preference for maintenance windows
- Notifications disabled

For new clusters, you can customize the settings at creation time. To do this in the AWS Management Console, clear **Use default settings** to modify the following settings:

- **Network and Security**—Allows you to run individual DAX cluster nodes in different Availability Zones within the current AWS Region. If you choose **No Preference**, the nodes are distributed among Availability Zones automatically.
- **Parameter Group**—A named set of parameters that are applied to every node in the cluster. You can use a parameter group to specify cache TTL behavior. You can change the value of any given parameter within a parameter group (except default parameter group `default.dax.1.0`) at any time.
- **Maintenance Window**—A weekly time period during which software upgrades and patches are applied to the nodes in the cluster. You can choose the start day, start time, and duration of the maintenance window. If you choose **No Preference**, the maintenance window is selected at random from an 8-hour block of time per Region. For more information, see [Maintenance window](#).

 **Note**

Parameter Group and **Maintenance Window** can also be changed at any time on a running cluster.

When a maintenance event occurs, DAX can notify you using Amazon Simple Notification Service (Amazon SNS). To configure notifications, choose an option from the **Topic for SNS notification** selector. You can create a new Amazon SNS topic, or use an existing topic.

For more information about setting up and subscribing to an Amazon SNS topic, see [Getting started with Amazon SNS](#) in the *Amazon Simple Notification Service Developer Guide*.

Configuring TTL settings

DAX maintains two caches for data that it reads from DynamoDB:

- **Item cache**—For items retrieved using GetItem or BatchGetItem.
- **Query cache**—For result sets retrieved using Query or Scan.

For more information, see [Item cache](#) and [Query cache](#).

The default TTL for each of these caches is 5 minutes. If you want to use different TTL settings, you can launch a DAX cluster using a custom parameter group. To do this on the console, choose **DAX | Parameter groups** in the navigation pane.

You can also perform these tasks using the AWS CLI. The following example shows how to launch a new DAX cluster using a custom parameter group. In this example, the item cache TTL is set to 10 minutes, and the query cache TTL is set to 3 minutes.

1. Create a new parameter group.

```
aws dax create-parameter-group \
--parameter-group-name custom-ttl
```

2. Set the item cache TTL to 10 minutes (600000 milliseconds).

```
aws dax update-parameter-group \
--parameter-group-name custom-ttl \
--parameter-name-values "ParameterName=record-ttl-millis,ParameterValue=600000"
```

3. Set the query cache TTL to 3 minutes (180000 milliseconds).

```
aws dax update-parameter-group \
--parameter-group-name custom-ttl \
--parameter-name-values "ParameterName=query-ttl-millis,ParameterValue=180000"
```

4. Verify that the parameters have been set correctly.

```
aws dax describe-parameters --parameter-group-name custom-ttl \
    --query "Parameters[*].[ParameterName,Description,ParameterValue]"
```

You can now launch a new DAX cluster with this parameter group.

```
aws dax create-cluster \
    --cluster-name MyNewCluster \
    --node-type dax.r3.large \
    --replication-factor 3 \
    --iam-role-arn arn:aws:iam::123456789012:role/DAXServiceRole \
    --parameter-group custom-ttl
```

 **Note**

You can't modify a parameter group that is being used by a running DAX instance.

Tagging support for DAX

Many AWS services, including DynamoDB, support *tagging*—the ability to label resources with user-defined names. You can assign tags to DAX clusters, allowing you to quickly identify all of your AWS resources that have the same tag, or to categorize your AWS bills by the tags you assign.

For more information, see [Adding tags and labels to resources](#).

Using the AWS Management Console

To manage DAX cluster tags

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose the cluster that you want to work with.
4. Choose the **Tags** tab. You can add, list, edit, or delete your tags here.

When the settings are as you want them, choose **Apply Changes**.

Using the AWS CLI

When you use the AWS CLI to manage DAX cluster tags, you must first determine the Amazon Resource Name (ARN) for the cluster. The following example shows how to determine the ARN for a cluster named MyDAXCluster.

```
aws dax describe-clusters \
--cluster-name MyDAXCluster \
--query "Clusters[*].ClusterArn"
```

In the output, the ARN will look similar to this: arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster

The following example shows how to tag the cluster.

```
aws dax tag-resource \
--resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
--tags="Key=ClusterUsage,Value=prod"
```

List all the tags for a cluster.

```
aws dax list-tags \
--resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster
```

To remove a tag, specify its key.

```
aws dax untag-resource \
--resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
--tag-keys ClusterUsage
```

AWS CloudTrail integration

DAX is integrated with AWS CloudTrail, allowing you to audit DAX cluster activities. You can use CloudTrail logs to view all the changes that have been made at the cluster level. You can also see changes to cluster components such as nodes, subnet groups, and parameter groups. For more information, see [Logging DynamoDB operations by using AWS CloudTrail](#).

Deleting a DAX cluster

If you are no longer using a DAX cluster, you should delete it to avoid being charged for unused resources.

You can delete a DAX cluster using the console or the AWS CLI. The following is an example.

```
aws dax delete-cluster --cluster-name mydaxcluster
```

Monitoring DAX

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon DynamoDB Accelerator (DAX) and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs.

Before you start monitoring DAX, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

Topics

- [Monitoring tools](#)
- [Monitoring with Amazon CloudWatch](#)
- [Logging DAX operations using AWS CloudTrail](#)

Monitoring tools

AWS provides tools that you can use to monitor Amazon DynamoDB Accelerator (DAX). You can configure some of these tools to do the monitoring for you, and some require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Topics

- [Automated monitoring tools](#)
- [Manual monitoring tools](#)

Automated monitoring tools

You can use the following automated monitoring tools to watch DAX and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see [Monitoring with Amazon CloudWatch](#).
- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see [Monitoring Log Files](#) in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see [What Is Amazon CloudWatch Events](#) in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see [Working with CloudTrail Log Files](#) in the *AWS CloudTrail User Guide*.

Manual monitoring tools

Another important part of monitoring DAX involves manually monitoring those items that the CloudWatch alarms don't cover. The DAX, CloudWatch, Trusted Advisor, and other AWS Management Console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on DAX.

- The DAX dashboard shows the following:
 - Service health

- The CloudWatch home page shows the following:
 - Current alarms and status
 - Graphs of alarms and resources
 - Service health status

In addition, you can use CloudWatch to do the following:

- Create [customized dashboards](#) to monitor the services that you care about.
- Graph metric data to troubleshoot issues and discover trends.
- Search and browse all of your AWS resource metrics.
- Create and edit alarms to be notified of problems.

Monitoring with Amazon CloudWatch

You can monitor Amazon DynamoDB Accelerator (DAX) using Amazon CloudWatch, which collects and processes raw data from DAX into readable, near real-time metrics. These statistics are recorded for a period of two weeks. You can then access historical information for a better perspective on how your web application or service is performing. By default, DAX metric data is sent to CloudWatch automatically. For more information, see [What Is Amazon CloudWatch?](#) in the *Amazon CloudWatch User Guide*.

Topics

- [How do I use DAX metrics?](#)
- [Viewing DAX metrics and dimensions](#)
- [Creating CloudWatch alarms to monitor DAX](#)
- [Production monitoring](#)

How do I use DAX metrics?

The metrics reported by DAX provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, and not a comprehensive list.

How Can I?	Relevant Metrics
Determine if any system errors occurred	Monitor <code>FaultRequestCount</code> to determine if any requests resulted in an HTTP 500 (server error) code. This can indicate a DAX internal service error or an HTTP 500 in the underlying table's SystemErrors metric .
Determine if any user errors occurred	Monitor <code>ErrorRequestCount</code> to determine if any requests resulted in an HTTP 400 (client error) code. If you see the error count growing, you might want to investigate and make sure you are sending correct client requests.
Determine if any cache misses occurred	Monitor <code>ItemCacheMisses</code> to determine the number of times an item was not found in the cache, and <code>QueryCacheMisses</code> and <code>ScanCacheMisses</code> to determine the number of times a query or scan result was not found in the cache.
Monitor cache hit rates	<p>Use CloudWatch Metric Math to define a cache hit rate metric using math expressions.</p> <p>For example, for the item cache, you can use the expression $m1/\text{SUM}([m1, m2]) * 100$, where $m1$ is the <code>ItemCacheHits</code> metric and $m2$ is the <code>ItemCacheMisses</code> metric for your cluster. For the query and scan caches, you can follow the same pattern using the corresponding query and scan cache metric.</p>

Viewing DAX metrics and dimensions

When you interact with Amazon DynamoDB, it sends metrics and dimensions to Amazon CloudWatch. You can use the following procedures to view the metrics for DynamoDB Accelerator (DAX).

To view metrics (console)

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.

2. In the navigation pane, choose **Metrics**.
3. Select the **DAX** namespace.

To view metrics (AWS CLI)

- At a command prompt, use the following command.

```
aws cloudwatch list-metrics --namespace "AWS/DAX"
```

DAX metrics and dimensions

The following sections contain the metrics and dimensions that DAX sends to CloudWatch.

DAX Metrics

The following metrics are available from DAX. DAX sends metrics to CloudWatch only when they have a non-zero value.

Note

CloudWatch aggregates the following DAX metrics at one-minute intervals:

- CPUUtilization
- CacheMemoryUtilization
- NetworkBytesIn
- NetworkBytesOut
- NetworkPacketsIn
- NetworkPacketsOut
- GetItemRequestCount
- BatchGetItemRequestCount
- BatchWriteItemRequestCount
- DeleteItemRequestCount
- PutItemRequestCount
- UpdateItemRequestCount
- TransactWriteItemsCount

- TransactItemCount
- ItemCacheHits
- ItemCacheMisses
- QueryCacheHits
- QueryCacheMisses
- ScanCacheHits
- ScanCacheMisses
- TotalRequestCount
- ErrorRequestCount
- FaultRequestCount
- FailedRequestCount
- QueryRequestCount
- ScanRequestCount
- ClientConnections
- EstimatedDbSize
- EvictedSize
- CPUCreditUsage
- CPUCreditBalance
- CPUSurplusCreditBalance
- CPUSurplusCreditsCharged

Not all statistics, such as Average or Sum, are applicable for every metric. However, all of these values are available through the DAX console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics. In the following table, each metric has a list of valid statistics that are applicable to that metric.

Metric	Description
CPUUtilization	The percentage of CPU utilization of the node or cluster.

Metric	Description
	<p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average
CacheMemoryUtilization	<p>The percentage of available cache memory that is in use by the item cache and query cache on the node or cluster. Cached data starts to be evicted prior to memory utilization reaching 100% (see EvictedSize metric). If CacheMemoryUtilization reaches 100% on any node, write requests will be throttled and you should consider switching to a cluster with a larger node type.</p> <p>Units: Percent</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average
NetworkBytesIn	<p>The number of bytes received on all network interfaces by the node or cluster.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average

Metric	Description
NetworkBytesOut	<p>The number of bytes sent out on all network interfaces by the node or cluster. This metric identifies the volume of outgoing traffic in terms of the number of bytes on a single node or cluster.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average
NetworkPacketsIn	<p>The number of packets received on all network interfaces by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average
NetworkPacketsOut	<p>The number of packets sent out on all network interfaces by the node or cluster. This metric identifies the volume of outgoing traffic in terms of the number of packets on a single node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average

Metric	Description
GetItemRequestCount	<p>The number of <code>.GetItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
BatchGetItemRequestCount	<p>The number of <code>BatchGetItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
BatchWriteItemRequestCount	<p>The number of <code>BatchWriteItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
DeleteItemRequestCount	<p>The number of <code>DeleteItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
PutItemRequestCount	<p>The number of PutItem requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
UpdateItemRequestCount	<p>The number of UpdateItem requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
TransactWriteItemCount	<p>The number of <code>TransactWriteItems</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
TransactGetItemCount	<p>The number of <code>TransactGetItems</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
ItemCacheHits	<p>The number of times an item was returned from the cache by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
ItemCacheMisses	<p>The number of times an item was not in the node or cluster cache, and had to be retrieved from DynamoDB.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
QueryCacheHits	<p>The number of times a query result was returned from the node or cluster cache.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
QueryCacheMisses	<p>The number of times a query result was not in the node or cluster cache, and had to be retrieved from DynamoDB.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
ScanCacheHits	<p>The number of times a scan result was returned from the node or cluster cache.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
ScanCacheMisses	<p>The number of times a scan result was not in the node or cluster cache, and had to be retrieved from DynamoDB.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
TotalRequestCount	<p>Total number of requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
ErrorRequestCount	<p>Total number of requests that resulted in a user error reported by the node or cluster. Requests that were throttled by the node or cluster are included.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
ThrottledRequestCount	<p>Total number of requests throttled by the node or cluster. Requests that were throttled by DynamoDB are not included, and can be monitored using DynamoDB Metrics.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
FaultRequestCount	<p>Total number of requests that resulted in an internal error reported by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
FailedRequestCount	<p>Total number of requests that resulted in an error reported by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
QueryRequestCount	<p>The number of query requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
ScanRequestCount	<p>The number of scan requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
ClientConnections	<p>The number of simultaneous connections made by clients to the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum
EstimatedDbSize	<p>An approximation of the amount of data cached in the item cache and the query cache by the node or cluster.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average

Metric	Description
EvictedSize	<p>The amount of data that was evicted by the node or cluster to make room for newly requested data. If the miss rate goes up, and you see this metric also growing, it probably means that your working set has increased. You should consider switching to a cluster with a larger node type.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• Sum
CPUCreditUsage	<p>The number of CPU credits spent by the node for CPU utilization. One CPU credit equals one vCPU running at 100% utilization for one minute or an equivalent combination of vCPUs, utilization, and time (for example, one vCPU running at 50% utilization for two minutes or two vCPUs running at 25% utilization for two minutes).</p> <p>CPU credit metrics are available at a five-minute frequency only. If you specify a period greater than five minutes, use the Sum statistic instead of the Average.</p> <p>Units: Credits (vCPU-minutes)</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
CPUCreditBalance	<p>The number of earned CPU credits that a node has accrued since it was launched or started.</p> <p>Credits are accrued in the credit balance after they are earned, and removed from the credit balance when they are spent. The credit balance has a maximum limit, determined by the DAX node size. After the limit is reached, any new credits that are earned are discarded.</p> <p>The credits in the CPUCreditBalance are available for the node to spend to burst beyond its baseline CPU utilization.</p> <p>Units: Credits (vCPU-minutes)</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
CPUSurplusCreditBalance	<p>The number of surplus credits that have been spent by a DAX node when its CPUCreditBalance value is zero.</p> <p>The CPUSurplusCreditBalance value is paid down by earned CPU credits. If the number of surplus credits exceeds the maximum number of credits that the node can earn in a 24-hour period, the spent surplus credits above the maximum incur an additional charge.</p> <p>Units: Credits (vCPU-minutes)</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

Metric	Description
CPUSurplusCreditsCharged	<p>The number of spent surplus credits that are not paid down by earned CPU credits, and which thus incur an additional charge.</p> <p>Spent surplus credits are charged when the spent surplus credits exceed the maximum number of credits that the node can earn in a 24-hour period. Spent surplus credits above the maximum are charged at the end of the hour or when the node is terminated.</p> <p>Units: Credits (vCPU-minutes)</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Minimum• Maximum• Average• SampleCount• Sum

 **Note**

The CPUCreditUsage, CPUCreditBalance, CPUSurplusCreditBalance, and CPUSurplusCreditsCharged metrics are available only for T3 nodes.

Dimensions for DAX Metrics

The metrics for DAX are qualified by the values for the account, cluster ID, or cluster ID and node ID combination. You can use the CloudWatch console to retrieve DAX data along any of the dimensions in the following table.

Dimension	Description
Account	Provides aggregated statistics across all nodes in an account.
ClusterId	Limits the data to a cluster.
ClusterId, NodeId	Limits the data to a node within a cluster.

Creating CloudWatch alarms to monitor DAX

You can create an Amazon CloudWatch alarm that sends an Amazon Simple Notification Service (Amazon SNS) message when the alarm changes state. An alarm watches a single metric over a time period that you specify. It performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification that is sent to an Amazon SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state. The state must have changed and been maintained for a specified number of periods.

How can I be notified of query cache misses?

1. Create an Amazon SNS topic, `arn:aws:sns:us-west-2:522194210714:QueryMissAlarm`.

For more information, see [Set Up Amazon Simple Notification Service](#) in the *Amazon CloudWatch User Guide*.

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name QueryCacheMissesAlarm \
--alarm-description "Alarm over query cache misses" \
--namespace AWS/DAX \
```

```
--metric-name QueryCacheMisses \
--dimensions Name=ClusterID,Value=myCluster \
--statistic Sum \
--threshold 8 \
--comparison-operator GreaterThanOrEqualToThreshold \
--period 60 \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-west-2:522194210714:QueryMissAlarm
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
"initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
"initializing" --state-value ALARM
```

Note

You can increase or decrease the threshold to one that makes sense for your application. You can also use [CloudWatch Metric Math](#) to define a cache miss rate metric and set an alarm over that metric.

How can I be notified if requests cause any internal error in the cluster?

1. Create an Amazon SNS topic, arn:aws:sns:us-west-2:123456789012:notify-on-system-errors.

For more information, see [Set Up Amazon Simple Notification Service](#) in the *Amazon CloudWatch User Guide*.

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name FaultRequestCountAlarm \
--alarm-description "Alarm when a request causes an internal error" \
--namespace AWS/DAX \
--metric-name FaultRequestCount \
--dimensions Name=ClusterID,Value=myCluster \
```

```
--statistic Sum \
--threshold 0 \
--comparison-operator GreaterThanThreshold \
--period 60 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-reason
"initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-reason
"initializing" --state-value ALARM
```

Production monitoring

You should establish a baseline for normal DAX performance in your environment, by measuring performance at various times and under different load conditions. As you monitor DAX, you should consider storing historical monitoring data. This stored data gives you a baseline from which to compare current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline, you should, at minimum, monitor the following items both during load testing and in production.

- CPU utilization and throttled requests, so that you can determine whether you might need to use a larger node type in your cluster. The CPU utilization of your cluster is available through the `CPUUtilization` CloudWatch metric.
- Operation latency (as measure on the client side) should remain consistent within your application's latency requirements.
- Error rates should remain low, as seen from the `ErrorRequestCount`, `FaultRequestCount`, and `FailedRequestCount` CloudWatch metrics.
- Network bytes consumption, so that you can determine whether you will need to use more nodes or a larger node type in your cluster. `NetworkBytesIn` and `NetworkBytesOut` metrics are available in CloudWatch, and you should compare them against your instances available baseline bandwidth as documented [here](#).

Note

The Amazon EC2 documented available baseline bandwidth is in Gigabits per second (Gbps), whereas the NetworkBytesIn and NetworkBytesOut metrics are in Gigabytes per minute (GBpm). To convert Gbps to GBpm and measure utilization, please multiply the baseline bandwidth by 7.5.

- Cache memory utilization and evicted size, so that you can determine whether the cluster's node type has sufficient memory to hold your working set, and if not, switch to a larger node type.

Note

In case of a large number of cache misses and writes, cache memory utilization can increase up to 100% and may cause availability downtime.

- Client connections, so that you can monitor for any unexplained spikes in connections to the cluster.

Logging DAX operations using AWS CloudTrail

Amazon DynamoDB Accelerator (DAX) is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in DAX.

To learn more about DAX and CloudTrail, see the DynamoDB Accelerator (DAX) section in [Logging DynamoDB operations by using AWS CloudTrail](#).

DAX T3/T2 burstable instances

DAX allows you to choose between fixed performance instances (such as R4 and R5) and burstable performance instances (such as T2 and T3). Burstable performance instances provide a baseline level of CPU performance with the ability to burst above the baseline when needed.

Baseline performance and the ability to burst above it are governed by CPU credits. Burstable performance instances accumulate CPU credits continuously, at a rate determined by the instance size, when the workload is below the baseline threshold. These credits may then be consumed when the workload increases. A CPU credit provides the performance of a full CPU core for one minute.

Many workloads don't need consistently high levels of CPU, but benefit significantly from having full access to very fast CPUs when they need them. Burstable performance instances are engineered specifically for these use cases. If you need consistently high CPU performance for your database, we recommend you use fixed performance instances.

DAX T2 instance family

DAX T2 instances are burstable general-purpose performance instances that provide a baseline level of CPU performance with the ability to burst above the baseline. T2 instances are a good choice for test and development workloads which need price predictability. DAX T2 instances are configured for standard mode, which means that if the instance is running low on accrued credits, CPU utilization is gradually lowered to the baseline level. For more information on standard mode, refer to [Standard mode for burstable performance instances](#) in the Amazon EC2 User Guide for Linux Instances.

DAX T3 instance family

DAX T3 instances are the next generation burstable general-purpose instance type, providing a baseline level of CPU performance with the ability to burst CPU usage at any time for as long as required. T3 instances offer a balance of compute, memory, and network resources and are ideal for workloads with moderate CPU usage that experience temporary spikes in use. DAX T3 instances are configured for unlimited mode, which means they can burst beyond the baseline over a 24-hour window for an additional charge. For more information on unlimited mode, refer to [Unlimited mode for burstable performance instances](#) in the Amazon EC2 User Guide for Linux Instances.

DAX T3 instances can sustain high CPU performance for as long as a workload requires it. For most general-purpose workloads, T3 instances will provide ample performance without any additional charges. The hourly T3 instance price automatically covers all interim spikes in usage when the average CPU utilization of a T3 instance is at or less than the baseline over a 24-hour window.

For example, a `dax.t3.small` instance receives credits continuously at a rate of 24 CPU credits per hour. This capability provides baseline performance equivalent to 20% of a CPU core ($20\% \times 60 \text{ minutes} = 12 \text{ minutes}$). If the instance does not use the credits it receives, they are stored in its CPU credit balance up to a maximum of 576 CPU credits. When the `t3.small` instance needs to burst to more than 20% of a core, it draws from its CPU credit balance to handle this surge automatically.

While DAX T2 instances are restricted to baseline performance once the CPU credit balance is drawn down to zero, DAX T3 instances can burst above the baseline even when their CPU credit

balance is zero. For the vast majority of workloads, where the average CPU utilization is at or below the baseline performance, the basic hourly price for t3.small covers all CPU bursts. If the instance happens to run at an average 25% CPU utilization (5% above baseline) over a period of 24 hours after its CPU credit balance is drawn to zero, it will be charged an additional 11.52 cents (9.6 cents/vCPU-hour × 1 vCPU × 5% × 24 hours). See [Amazon DynamoDB Pricing](#) for pricing details.

DAX access control

DynamoDB Accelerator (DAX) is designed to work together with DynamoDB, to seamlessly add a caching layer to your applications. However, DAX and DynamoDB have separate access control mechanisms. Both services use AWS Identity and Access Management (IAM) to implement their respective security policies, but the security models for DAX and DynamoDB are different.

We highly recommend that you understand both security models, so that you can implement proper security measures for your applications that use DAX.

This section describes the access control mechanisms provided by DAX and provides sample IAM policies that you can tailor to your needs.

With DynamoDB, you can create IAM policies that limit the actions a user can perform on individual DynamoDB resources. For example, you can create a user role that only allows the user to perform read-only actions on a particular DynamoDB table. (For more information, see [Identity and Access Management for Amazon DynamoDB](#).) By comparison, the DAX security model focuses on cluster security, and the ability of the cluster to perform DynamoDB API actions on your behalf.

Warning

If you are currently using IAM roles and policies to restrict access to DynamoDB tables data, then the use of DAX can **subvert** those policies. For example, a user could have access to a DynamoDB table via DAX but not have explicit access to the same table accessing DynamoDB directly. For more information, see [Identity and Access Management for Amazon DynamoDB](#).

DAX does not enforce user-level separation on data in DynamoDB. Instead, users inherit the permissions of the DAX cluster's IAM policy when they access that cluster. Thus, when accessing DynamoDB tables via DAX, the only access controls that are in effect are the permissions in the DAX cluster's IAM policy. No other permissions are recognized.

If you require isolation, we recommend that you create additional DAX clusters and scope the IAM policy for each cluster accordingly. For example, you could create multiple DAX clusters and allow each cluster to access only a single table.

IAM service role for DAX

When you create a DAX cluster, you must associate the cluster with an IAM role. This is known as the *service role* for the cluster.

Suppose that you wanted to create a new DAX cluster named *DAXCluster01*. You could create a service role named *DAXServiceRole*, and associate the role with *DAXCluster01*. The policy for *DAXServiceRole* would define the DynamoDB actions that *DAXCluster01* could perform, on behalf of the users who interact with *DAXCluster01*.

When you create a service role, you must specify a trust relationship between *DAXServiceRole* and the DAX service itself. A trust relationship determines which entities can assume a role and make use of its permissions. The following is an example trust relationship document for *DAXServiceRole*:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "dax.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

This trust relationship allows a DAX cluster to assume *DAXServiceRole* and perform DynamoDB API calls on your behalf.

The DynamoDB API actions that are allowed are described in an IAM policy document, which you attach to *DAXServiceRole*. The following is an example policy document.

```
{  
    "Version": "2012-10-17",
```

```
"Statement": [  
    {  
        "Sid": "DaxAccessPolicy",  
        "Effect": "Allow",  
        "Action": [  
            "dynamodb:DescribeTable",  
            "dynamodb:PutItem",  
            "dynamodb:GetItem",  
            "dynamodb:UpdateItem",  
            "dynamodb:DeleteItem",  
            "dynamodb:Query",  
            "dynamodb:Scan",  
            "dynamodb:BatchGetItem",  
            "dynamodb:BatchWriteItem",  
            "dynamodb:ConditionCheckItem"  
        ],  
        "Resource": [  
            "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        ]  
    }  
]
```

This policy allows DAX to perform necessary DynamoDB API actions on a DynamoDB table. The dynamodb:DescribeTable action is required for DAX to maintain metadata about the table, and the others are read and write actions performed on items in the table. The table, named Books, is in the us-west-2 Region and is owned by AWS account ID 123456789012.

 **Note**

DAX supports mechanisms to prevent the confused deputy problem during cross-Service access. For more information, see [The confused deputy problem](#) in the *IAM User Guide*.

IAM policy to allow DAX cluster access

After you create a DAX cluster, you need to grant permissions to a user so that the user can access the DAX cluster.

For example, suppose that you want to grant access to *DAXCluster01* to a user named Alice. You would first create an IAM policy (*AliceAccessPolicy*) that defines the DAX clusters and DAX API

actions that the recipient can access. You would then confer access by attaching this policy to user Alice.

The following policy document gives the recipient full access on *DAXCluster01*.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dax:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"  
            ]  
        }  
    ]  
}
```

The policy document allows access to the DAX cluster, but it does not grant any DynamoDB permissions. (The DynamoDB permissions are conferred by the DAX service role.)

For user Alice, you would first create `AliceAccessPolicy` with the policy document shown previously. You would then attach the policy to Alice.

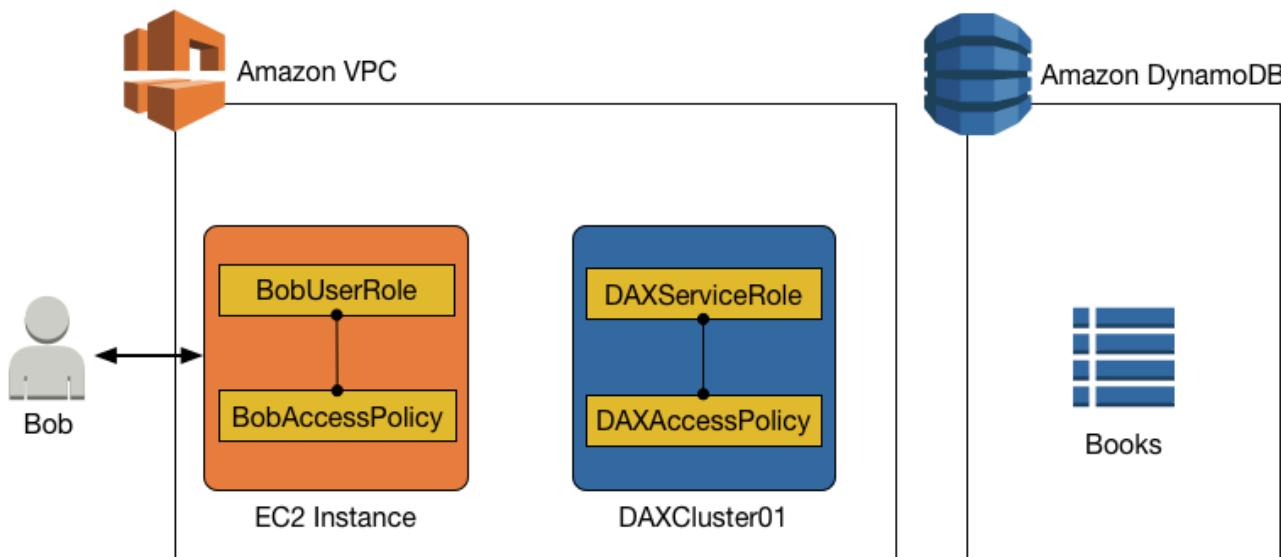
 **Note**

Instead of attaching the policy to a user, you could attach it to an IAM role. That way, all of the users who assume that role would have the permissions that you defined in the policy.

The user policy, together with the DAX service role, determine the DynamoDB resources and API actions that the recipient can access via DAX.

Case study: Accessing DynamoDB and DAX

The following scenario can help further your understanding of IAM policies for use with DAX. (This scenario is referred to throughout the rest of this section.) The following diagram shows a high-level overview of the scenario.

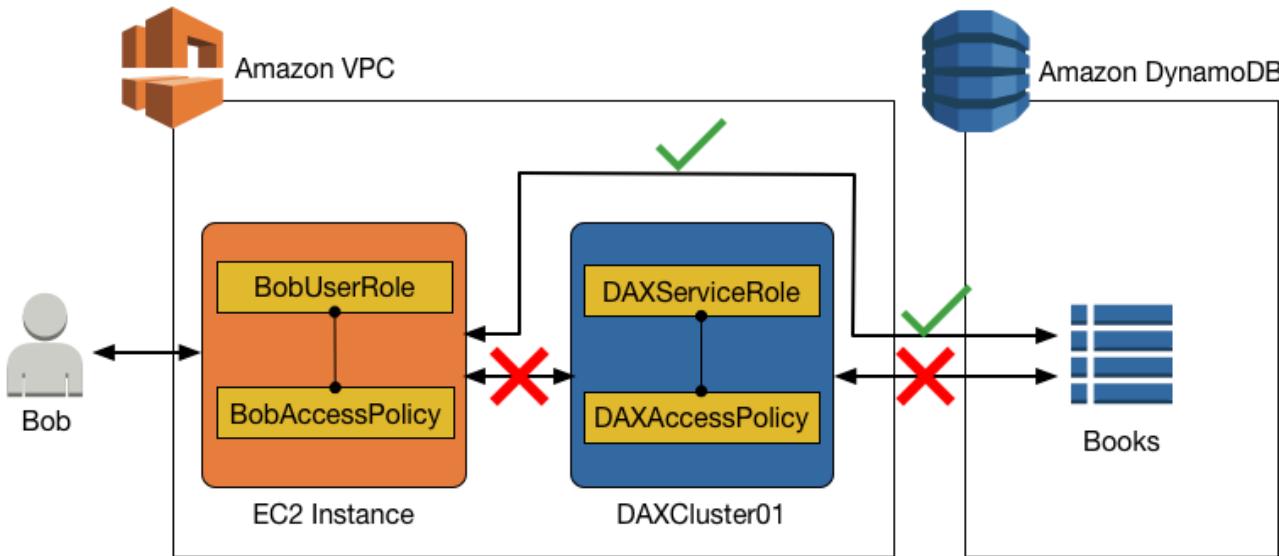


In this scenario, there are the following entities:

- A user (Bob).
- An IAM role (BobUserRole). Bob assumes this role at runtime.
- An IAM policy (BobAccessPolicy). This policy is attached to BobUserRole. BobAccessPolicy defines the DynamoDB and DAX resources that BobUserRole is allowed to access.
- A DAX cluster (DAXCluster01).
- An IAM service role (DAXServiceRole). This role allows DAXCluster01 to access DynamoDB.
- An IAM policy (DAXAccessPolicy). This policy is attached to DAXServiceRole. DAXAccessPolicy defines the DynamoDB APIs and resources that DAXCluster01 is allowed to access.
- A DynamoDB table (Books).

The combination of policy statements in BobAccessPolicy and DAXAccessPolicy determine what Bob can do with the Books table. For example, Bob might be able to access Books directly (using the DynamoDB endpoint), indirectly (using the DAX cluster), or both. Bob might also be able to read data from Books, write data to Books, or both.

Access to DynamoDB, but no access with DAX



It is possible to allow direct access to a DynamoDB table, while preventing indirect access using a DAX cluster. For direct access to DynamoDB, the permissions for BobUserRole are determined by BobAccessPolicy (which is attached to the role).

Read-only access to DynamoDB (only)

Bob can access DynamoDB with BobUserRole. The IAM policy attached to this role (BobAccessPolicy) determines the DynamoDB tables that BobUserRole can access, and what APIs that BobUserRole can invoke.

Consider the following policy document for BobAccessPolicy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DynamoDBAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan"  
            ]  
        }  
    ]  
}
```

```
        "dynamodb:Scan"
    ],
    "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
}
]
```

When this document is attached to BobAccessPolicy, it allows BobUserRole to access the DynamoDB endpoint and perform read-only operations on the Books table.

DAX does not appear in this policy, so access via DAX is denied.

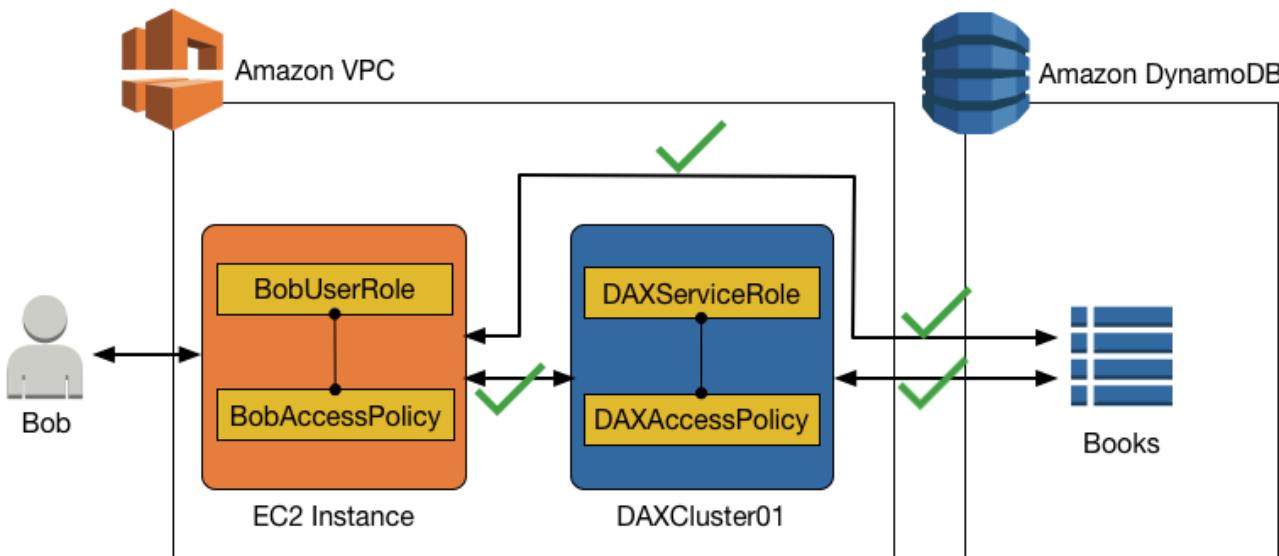
Read/write access to DynamoDB (only)

If BobUserRole requires read/write access to DynamoDB, the following policy would work.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Again, DAX does not appear in this policy, so access via DAX is denied.

Access to DynamoDB and to DAX



To allow access to a DAX cluster, you must include DAX-specific actions in an IAM policy.

The following DAX-specific actions correspond to their similarly named counterparts in the DynamoDB API:

- `dax:GetItem`
- `dax:BatchGetItem`
- `dax:Query`
- `dax:Scan`
- `dax:PutItem`
- `dax:UpdateItem`
- `dax:DeleteItem`
- `dax:BatchWriteItem`
- `dax:ConditionCheckItem`

The same is true for the `dax:EnclosingOperation` condition key.

Read-only access to DynamoDB and read-only access to DAX

Suppose that Bob requires read-only access to the Books table, from DynamoDB and from DAX. The following policy (attached to BobUserRole) confers this access.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DAXAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dax:GetItem",  
                "dax:BatchGetItem",  
                "dax:Query",  
                "dax:Scan"  
            ],  
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"  
        },  
        {  
            "Sid": "DynamoDBAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

The policy has a statement for DAX access (DAXAccessStmt) and another statement for DynamoDBaccess (DynamoDBAccessStmt). These statements allow Bob to send GetItem, BatchGetItem, Query, and Scan requests to DAXCluster01.

However, the service role for DAXCluster01 would also require read-only access to the Books table in DynamoDB. The following IAM policy, attached to DAXServiceRole, would fulfill this requirement.

```
{
```

```
"Version": "2012-10-17",
"Statement": [
    {
        "Sid": "DynamoDBAccessStmt",
        "Effect": "Allow",
        "Action": [
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
    }
]
```

Read/write access to DynamoDB and read-only with DAX

For a given user role, you can provide read/write access to a DynamoDB table, while also allowing read-only access via DAX.

For Bob, the IAM policy for BobUserRole would need to allow DynamoDB read and write actions on the Books table, while also supporting read-only actions via DAXCluster01.

The following example policy document for BobUserRole confers this access.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",

```

```
        "Action": [
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:Scan",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem",
            "dynamodb:DescribeTable",
            "dynamodb:ConditionCheckItem"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
    }
]
}
```

In addition, DAXServiceRole would require an IAM policy that allows DAXCluster01 to perform read-only actions on the Books table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:DescribeTable"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Read/write access to DynamoDB and read/write access to DAX

Now suppose that Bob required read/write access to the Books table, directly from DynamoDB or indirectly from DAXCluster01. The following policy document, attached to BobAccessPolicy, confers this access.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DAXAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dax:GetItem",  
                "dax:BatchGetItem",  
                "dax:Query",  
                "dax:Scan",  
                "dax:PutItem",  
                "dax:UpdateItem",  
                "dax:DeleteItem",  
                "dax:BatchWriteItem",  
                "dax:ConditionCheckItem"  
            ],  
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"  
        },  
        {  
            "Sid": "DynamoDBAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem",  
                "dynamodb:DescribeTable",  
                "dynamodb:ConditionCheckItem"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

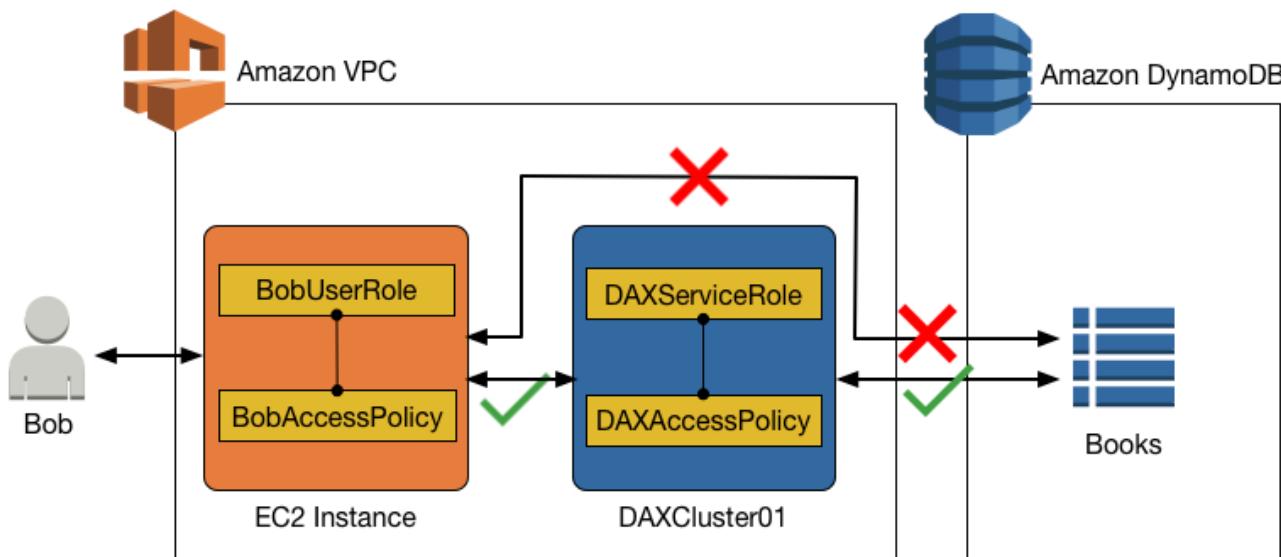
}

In addition, DAXServiceRole would require an IAM policy that allows DAXCluster01 to perform read/write actions on the Books table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DynamoDBAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem",  
                "dynamodb:DescribeTable"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

Access to DynamoDB via DAX, but no direct access to DynamoDB

In this scenario, Bob can access the Books table via DAX, but he does not have direct access to the Books table in DynamoDB. Thus, when Bob gains access to DAX, he also gains access to a DynamoDB table that he otherwise might not be able to access. When you configure an IAM policy for the DAX service role, remember that any user that is given access to the DAX cluster via the user access policy gains access to the tables specified in that policy. In this case, BobAccessPolicy gains access to the tables specified in DAXAccessPolicy.



If you are currently using IAM roles and policies to restrict access to DynamoDB tables and data, using DAX can subvert those policies. In the following policy, Bob has access to a DynamoDB table via DAX but does not have explicit direct access to the same table in DynamoDB.

The following policy document (BobAccessPolicy), attached to BobUserRole, would confer this access.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DAXAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dax:GetItem",  
                "dax:BatchGetItem",  
                "dax:Query",  
                "dax:Scan",  
                "dax:PutItem",  
                "dax:UpdateItem",  
                "dax:DeleteItem",  
                "dax:BatchWriteItem",  
                "dax:ConditionCheckItem"  
            ],  
            "Resource": "arn:aws:dynamodb:region:account-id:table/Books"  
        }  
    ]}
```

```
        "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
    }
]
}
```

In this access policy, there are no permissions to access DynamoDB directly.

Together with BobAccessPolicy, the following DAXAccessPolicy gives BobUserRole access to the DynamoDB table Books even though BobUserRole cannot directly access the Books table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

As this example shows, when you configure access control for the user access policy and the DAX cluster access policy, you must fully understand the end-to-end access to ensure that the principle of least privilege is observed. Also ensure that giving a user access to a DAX cluster does not subvert previously established access control policies.

DAX encryption at rest

Amazon DynamoDB Accelerator (DAX) encryption at rest provides an additional layer of data protection by helping secure your data from unauthorized access to the underlying storage.

Organizational policies, industry or government regulations, and compliance requirements might require the use of encryption at rest to protect your data. You can use encryption to increase the data security of your applications that are deployed in the cloud.

With encryption at rest, the data persisted by DAX on disk is encrypted using 256-bit Advanced Encryption Standard, also known as AES-256 encryption. DAX writes data to disk as part of propagating changes from the primary node to read replicas.

DAX encryption at rest automatically integrates with AWS Key Management Service (AWS KMS) for managing the single service default key that is used to encrypt your clusters. If a service default key doesn't exist when you create your encrypted DAX cluster, AWS KMS automatically creates a new AWS managed key for you. This key is used with encrypted clusters that are created in the future. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

After your data is encrypted, DAX handles the decryption of your data transparently with minimal impact on performance. You don't need to modify your applications to use encryption.

 **Note**

DAX does not call AWS KMS for every single DAX operation. DAX only uses the key at the cluster launch. Even if access is revoked, DAX can still access the data until the cluster is shut down. Customer-specified AWS KMS keys are not supported.

DAX encryption at rest is available for the following cluster node types.

Family	Node type
Memory-optimized (R4 and R5)	dax.r4.large
	dax.r4.xlarge
	dax.r4.2xlarge
	dax.r4.4xlarge
	dax.r4.8xlarge
	dax.r4.16xlarge

Family	Node type
	dax.r5.large
	dax.r5.xlarge
	dax.r5.2xlarge
	dax.r5.4xlarge
	dax.r5.8xlarge
	dax.r5.12xlarge
	dax.r5.16xlarge
	dax.r5.24xlarge
General purpose (T2)	dax.t2.small
	dax.t2.medium
General purpose (T3)	dax.t3.small
	dax.t3.medium

Important

DAX encryption at rest is not supported for dax.r3.* node types.

You cannot enable or disable encryption at rest after a cluster has been created. You must re-create the cluster to enable encryption at rest if it was not enabled at creation.

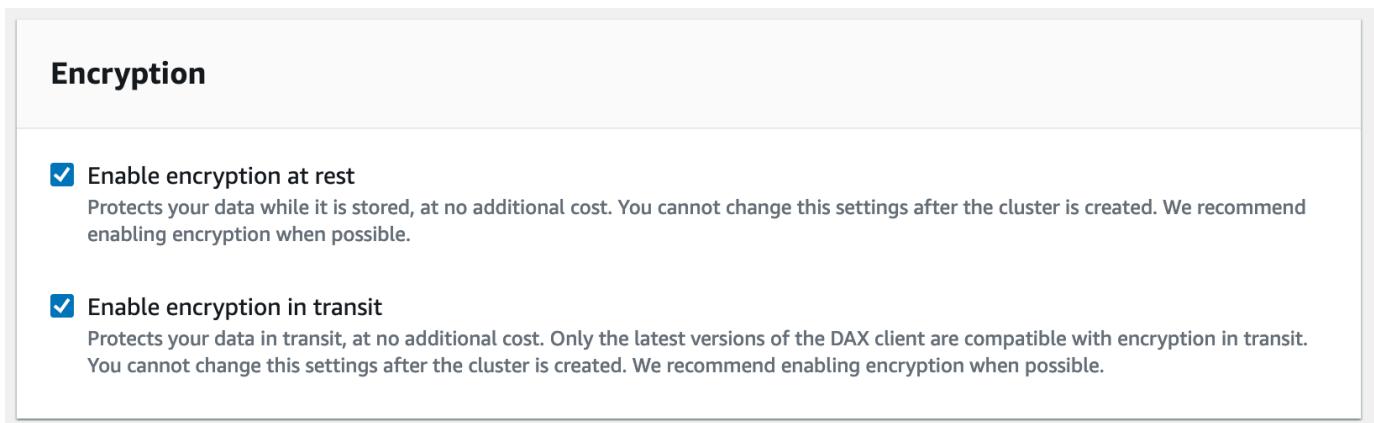
DAX encryption at rest is offered at no additional cost (AWS KMS encryption key usage charges apply). For information about pricing, see [Amazon DynamoDB pricing](#).

Enabling encryption at rest using the AWS Management Console

Follow these steps to enable DAX encryption at rest on a table using the console.

To enable DAX encryption at rest

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, under **DAX**, choose **Clusters**.
3. Choose **Create cluster**.
4. For **Cluster name**, enter a short name for your cluster. Choose the **node type** for all of the nodes in the cluster, and for the cluster size, use **3 nodes**.
5. In **Encryption**, make sure that **Enable encryption** is selected.



6. After choosing the IAM role, subnet group, security groups, and cluster settings, choose **Launch cluster**.

To confirm that the cluster is encrypted, check the cluster details under the **Clusters** pane. Encryption should be **ENABLED**.

DAX encryption in transit

Amazon DynamoDB Accelerator (DAX) supports encryption in transit of data between your application and your DAX cluster, enabling you to use DAX in applications with stringent encryption requirements.

Regardless of whether or not you choose encryption in transit, traffic between your application and your DAX cluster remains in your Amazon VPC. This traffic is routed to Elastic Network Interfaces with private IPs in your VPC that are attached to the nodes of your cluster. With your VPC as the trust boundary, you have significant control over the security of your data through the use of standard tools like security groups, subnet segmentation with Network ACLs, and VPC flow

tracing. DAX encryption in transit adds to this baseline level of confidentiality, ensuring that all requests and responses between the application and the cluster are encrypted by transport level security (TLS), and connections to the cluster can be authenticated by verification of a cluster x509 certificate. Data written to disk by DAX can also be encrypted if you choose [encryption at rest](#) when creating your DAX cluster.

Using encryption in transit with DAX is easy. Simply select this option when creating a new cluster, and use a recent version of any of the [DAX clients](#) in your application. Clusters that use encryption in transit do not support unencrypted traffic, so there is no chance to misconfigure your application and bypass encryption. The DAX client will use the cluster's x509 certificate to authenticate the identity of the cluster when it establishes connections, ensuring that your DAX requests go where intended. All methods of creating DAX clusters support encryption in transit: the AWS Management Console, AWS CLI, all SDKs, and AWS CloudFormation.

Encryption in transit cannot be enabled on an existing DAX cluster. To use encryption in transit in an existing DAX application, create a new cluster with encryption in transit enabled, shift your application's traffic to it, then delete the old cluster.

Using service-linked IAM roles for DAX

Amazon DynamoDB Accelerator (DAX) uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to DAX. Service-linked roles are predefined by DAX and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up DAX easier because you don't have to manually add the necessary permissions. DAX defines the permissions of its service-linked roles, and unless defined otherwise, only DAX can assume its roles. The defined permissions include the trust policy and the permissions policy. That permissions policy can't be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your DAX resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) in the *IAM User Guide*. Look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** link to view the service-linked role documentation for that service.

Topics

- [Service-linked role permissions for DAX](#)
- [Creating a service-linked role for DAX](#)
- [Editing a service-linked role for DAX](#)
- [Deleting a service-linked role for DAX](#)

Service-linked role permissions for DAX

DAX uses the service-linked role named `AWSServiceRoleForDAX`. This role allows DAX to call services on behalf of your DAX cluster.

Important

The `AWSServiceRoleForDAX` service-linked role makes it easier for you to set up and maintain a DAX cluster. However, you must still grant each cluster access to DynamoDB before you can use it. For more information, see [DAX access control](#).

The `AWSServiceRoleForDAX` service-linked role trusts the following services to assume the role:

- `dax.amazonaws.com`

The role permissions policy allows DAX to complete the following actions on the specified resources:

- Actions on `ec2`:
 - `AuthorizeSecurityGroupIngress`
 - `CreateNetworkInterface`
 - `CreateSecurityGroup`
 - `DeleteNetworkInterface`
 - `DeleteSecurityGroup`
 - `DescribeAvailabilityZones`
 - `DescribeNetworkInterfaces`
 - `DescribeSecurityGroups`
 - `DescribeSubnets`

- `DescribeVpcs`
- `ModifyNetworkInterfaceAttribute`
- `RevokeSecurityGroupIngress`

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

To allow an IAM entity to create `AWSServiceRoleForDAX` service-linked roles

Add the following policy statement to the permissions for that IAM entity.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iam:CreateServiceLinkedRole"  
    ],  
    "Resource": "*",  
    "Condition": {"StringLike": {"iam:AWSServiceName": "dax.amazonaws.com"}}  
}
```

Creating a service-linked role for DAX

You don't need to manually create a service-linked role. When you create a cluster, DAX creates the service-linked role for you.

Important

If you were using the DAX service before February 28, 2018, when it began supporting service-linked roles, DAX created the `AWSServiceRoleForDAX` role in your account. For more information, see [A New Role Appeared in My AWS Account](#) in the *IAM User Guide*.

If you delete this service-linked role and then need to create it again, you can use the same process to re-create the role in your account. When you create an instance or a cluster, DAX creates the service-linked role for you again.

Editing a service-linked role for DAX

DAX does not allow you to edit the `AWSServiceRoleForDAX` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

Deleting a service-linked role for DAX

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all of your DAX clusters before you can delete the service-linked role.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the `AWSServiceRoleForDAX` role.
3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review recent activity for the service-linked role.

Note

If you are unsure whether DAX is using the `AWSServiceRoleForDAX` role, you can try to delete the role. If the service is using the role, the deletion fails, and you can view the Regions where the role is being used. If the role is being used, you must delete your DAX clusters before you can delete the role. You can't revoke the session for a service-linked role.

If you want to remove the `AWSServiceRoleForDAX` role, you must first delete all of your DAX clusters.

Deleting all of your DAX clusters

Use one of these procedures to delete each of your DAX clusters.

To delete a DAX cluster (console)

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose **Actions**, and then choose **Delete**.
4. In the **Delete cluster confirmation** box, choose **Delete**.

To delete a DAX cluster (AWS CLI)

See [delete-cluster](#) in the *AWS CLI Command Reference*.

To delete a DAX cluster (API)

See [DeleteCluster](#) in the *Amazon DynamoDB API Reference*.

Deleting the service-linked role

To manually delete the service-linked role using IAM

Use the IAM console, the IAM CLI, or the IAM API to delete the `AWSServiceRoleForDAX` service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

Accessing DAX across AWS accounts

Imagine that you have a DynamoDB Accelerator (DAX) cluster running in one AWS account (account A), and the DAX cluster needs to be accessible from an Amazon Elastic Compute Cloud (Amazon EC2) instance in another AWS account (account B). In this tutorial, you accomplish this by launching an EC2 instance in account B with an IAM role from account B. You then use temporary security credentials from the EC2 instance to assume an IAM role from account A. Finally, you use the temporary security credentials from assuming the IAM role in account A to make application calls over an Amazon VPC peering connection to the DAX cluster in account A. In order to perform these tasks you will need administrative access in both AWS accounts.

⚠️ Important

It is not possible to have a DAX cluster access a DynamoDB table from a different account.

Topics

- [Set up IAM](#)
- [Set up a VPC](#)
- [Modify the DAX client to allow cross-account access](#)

Set up IAM

1. Create a text file named `AssumeDaxRoleTrust.json` with the following content, which allows Amazon EC2 to work on your behalf.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "ec2.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

2. In account B, create a role that Amazon EC2 can use when launching instances.

```
aws iam create-role \  
  --role-name AssumeDaxRole \  
  --assume-role-policy-document file://AssumeDaxRoleTrust.json
```

3. Create a text file named `AssumeDaxRolePolicy.json` with the following content, which allows code running on the EC2 instance in account B to assume an IAM role in account A. Replace `accountA` with the actual ID of account A.

```
{
```

```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "arn:aws:iam::accountA:role/DaxCrossAccountRole"
    }
]
```

4. Add that policy to the role you just created.

```
aws iam put-role-policy \
--role-name AssumeDaxRole \
--policy-name AssumeDaxRolePolicy \
--policy-document file://AssumeDaxRolePolicy.json
```

5. Create an instance profile to allow instances to use the role.

```
aws iam create-instance-profile \
--instance-profile-name AssumeDaxInstanceProfile
```

6. Associate the role with the instance profile.

```
aws iam add-role-to-instance-profile \
--instance-profile-name AssumeDaxInstanceProfile \
--role-name AssumeDaxRole
```

7. Create a text file named DaxCrossAccountRoleTrust.json with the following content, which allows account B to assume an account A role. Replace *accountB* with the actual ID of account B.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::accountB:role/AssumeDaxRole"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

```
}
```

8. In account A, create the role that account B can assume.

```
aws iam create-role \
--role-name DaxCrossAccountRole \
--assume-role-policy-document file://DaxCrossAccountRoleTrust.json
```

9. Create a text file named `DaxCrossAccountPolicy.json` that allows access to the DAX cluster. Replace `dax-cluster-arn` with the correct Amazon Resource Name (ARN) of your DAX cluster.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "dax-cluster-arn"
        }
    ]
}
```

10. In account A, add the policy to the role.

```
aws iam put-role-policy \
--role-name DaxCrossAccountRole \
--policy-name DaxCrossAccountPolicy \
--policy-document file://DaxCrossAccountPolicy.json
```

Set up a VPC

1. Find the subnet group of account A's DAX cluster. Replace *cluster-name* with the name of the DAX cluster that account B must access.

```
aws dax describe-clusters \
--cluster-name cluster-name
--query 'Clusters[0].SubnetGroup'
```

2. Using that *subnet-group*, find the cluster's VPC.

```
aws dax describe-subnet-groups \
--subnet-group-name subnet-group \
--query 'SubnetGroups[0].VpcId'
```

3. Using that *vpc-id*, find the VPC's CIDR.

```
aws ec2 describe-vpcs \
--vpc vpc-id \
--query 'Vpcs[0].CidrBlock'
```

4. From account B, create a VPC using a different, non-overlapping CIDR than the one found in the previous step. Then, create at least one subnet. You can use either the [VPC creation wizard](#) in the AWS Management Console or the [AWS CLI](#).
5. From account B, request a peering connection to the account A VPC as described in [Creating and accepting a VPC peering connection](#). From account A, accept the connection.
6. From account B, find the new VPC's routing table. Replace *vpc-id* with the ID of the VPC you created in account B.

```
aws ec2 describe-route-tables \
--filters 'Name=vpc-id,Values=vpc-id' \
--query 'RouteTables[0].RouteTableId'
```

7. Add a route to send traffic destined for account A's CIDR to the VPC peering connection. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
aws ec2 create-route \
--route-table-id accountB-route-table-id \
--destination-cidr accountA-vpc-cidr \
```

```
--vpc-peering-connection-id peering-connection-id
```

8. From account A, find the DAX cluster's route table using the *vpc-id* you found previously.

```
aws ec2 describe-route-tables \  
--filters 'Name=vpc-id, Values=accountA-vpc-id' \  
--query 'RouteTables[0].RouteTableId'
```

9. From account A, add a route to send traffic destined for account B's CIDR to the VPC peering connection. Replace each *user input placeholder* with the correct values for your accounts.

```
aws ec2 create-route \  
--route-table-id accountA-route-table-id \  
--destination-cidr accountB-vpc-cidr \  
--vpc-peering-connection-id peering-connection-id
```

10. From account B, launch an EC2 instance in the VPC that you created earlier. Give it the `AssumeDaxInstanceProfile`. You can use either the [launch wizard](#) in the AWS Management Console or the [AWS CLI](#). Take note of the instance's security group.
11. From account A, find the security group used by the DAX cluster. Remember to replace *cluster-name* with the name of your DAX cluster.

```
aws dax describe-clusters \  
--cluster-name cluster-name \  
--query 'Clusters[0].SecurityGroups[0].SecurityGroupIdentifier'
```

12. Update the DAX cluster's security group to allow inbound traffic from the security group of the EC2 instance you created in account B. Remember to replace the *user input placeholders* with the correct values for your accounts.

```
aws ec2 authorize-security-group-ingress \  
--group-id accountA-security-group-id \  
--protocol tcp \  
--port 8111 \  
--source-group accountB-security-group-id \  
--group-owner accountB-id
```

At this point, an application on account B's EC2 instance is able to use the instance profile to assume the `arn:aws:iam::accountA-id:role/DaxCrossAccountRole` role and use the DAX cluster.

Modify the DAX client to allow cross-account access

Note

AWS Security Token Service (AWS STS) credentials are temporary credentials. Some clients handle refreshing automatically, while others require additional logic to refresh the credentials. We recommend that you follow the guidance of the appropriate documentation.

Java

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [Java and DAX tutorial](#).

1. Add the following imports.

```
import com.amazonaws.auth.STSShareRoleSessionCredentialsProvider;
import com.amazonaws.services.securitytoken.AWSSecurityTokenService;
import
com.amazonaws.services.securitytoken.AWSSecurityTokenServiceClientBuilder;
```

2. Get a credentials provider from AWS STS and create a DAX client object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
AWSSecurityTokenService awsSecurityTokenService =
AWSSecurityTokenServiceClientBuilder
    .standard()
    .withRegion(region)
    .build();

STSShareRoleSessionCredentialsProvider credentials = new
STSShareRoleSessionCredentialsProvider.Builder("arn:aws:iam::accountA:role/
RoleName", "TryDax")
    .withStsClient(awsSecurityTokenService)
    .build();
```

```
DynamoDB client = AmazonDaxClientBuilder.standard()
    .withRegion(region)
    .withEndpointConfiguration(dax_endpoint)
    .withCredentials(credentials)
    .build();
```

.NET

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [.NET and DAX tutorial](#).

1. Add the [AWSSDK.SecurityToken](#) NuGet package to the solution.

```
<PackageReference Include="AWSSDK.SecurityToken" Version="latest version" />
```

2. Use the `SecurityToken` and `SecurityToken.Model` packages.

```
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;
```

3. Get temporary credentials from `AmazonSimpleTokenService` and create a `ClusterDaxClient` object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
IAmazonSecurityTokenService sts = new AmazonSecurityTokenServiceClient();

var assumeRoleResponse = sts.AssumeRole(new AssumeRoleRequest
{
    RoleArn = "arn:aws:iam::accountA:role/RoleName",
    RoleSessionName = "TryDax"
});

Credentials credentials = assumeRoleResponse.Credentials;

var clientConfig = new DaxClientConfig(dax_endpoint, port)
{
    AwsCredentials = assumeRoleResponse.Credentials
};
```

```
var client = new ClusterDaxClient(clientConfig);
```

Go

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find [working code examples on GitHub](#).

1. Import the AWS STS and session packages.

```
import (
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sts"
    "github.com/aws/aws-sdk-go/aws/credentials/stscreds"
)
```

2. Get temporary credentials from AmazonSimpleTokenService and create a DAX client object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String(region)},
)
if err != nil {
    return nil, err
}

stsClient := sts.New(sess)
arp := &stscreds.AssumeRoleProvider{
    Duration:      900 * time.Second,
    ExpiryWindow: 10 * time.Second,
    RoleARN:       "arn:aws:iam::accountA:role/role_name",
    Client:        stsClient,
    RoleSessionName: "session_name",
}
cfg := dax.DefaultConfig()

cfg.HostPorts = []string{dax_endpoint}
cfg.Region = region
cfg.Credentials = credentials.NewCredentials(arp)
daxClient := dax.New(cfg)
```

Python

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [Python and DAX](#) tutorial.

1. Import boto3.

```
import boto3
```

2. Get temporary credentials from sts and create an AmazonDaxClient object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
sts = boto3.client('sts')
stsresponse =
    sts.assume_role(RoleArn='arn:aws:iam::accountA:role/
RoleName',RoleSessionName='tryDax')
credentials = botocore.session.get_session()['Credentials']

dax = amazondax.AmazonDaxClient(session, region_name=region,
endpoints=[dax_endpoint], aws_access_key_id=credentials['AccessKeyId'],
aws_secret_access_key=credentials['SecretAccessKey'],
aws_session_token=credentials['SessionToken'])
client = dax
```

Node.js

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [Node.js and DAX](#) tutorial. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
const AmazonDaxClient = require('amazon-dax-client');
const AWS = require('aws-sdk');
const region = 'region';
const endpoints = [daxEndpoint1, ...];

const getCredentials = async() => {
  return new Promise((resolve, reject) => {
    const sts = new AWS.STS();
    const roleParams = {
```

```
        RoleArn: 'arn:aws:iam::accountA:role/RoleName',
        RoleSessionName: 'tryDax',
    );
    sts.assumeRole(roleParams, (err, session) => {
        if(err) {
            reject(err);
        } else {
            resolve({
                accessKeyId: session.Credentials.AccessKeyId,
                secretAccessKey: session.Credentials.SecretAccessKey,
                sessionToken: session.Credentials.SessionToken,
            });
        }
    });
});
};

const createDaxClient = async() => {
    const credentials = await getCredentials();
    const daxClient = new AmazonDaxClient({endpoints: endpoints, region: region,
accessKeyId: credentials.accessKeyId, secretAccessKey: credentials.secretAccessKey,
sessionToken: credentials.sessionToken});
    return new AWS.DynamoDB.DocumentClient({service: daxClient});
};

createDaxClient().then((client) => {
    client.get(...);
    ...
}).catch((error) => {
    console.log('Caught an error: ' + error);
});
```

DAX cluster sizing guide

This guide provides advice for choosing an appropriate Amazon DynamoDB Accelerator (DAX) cluster size and node type for your application. These instructions guide you through the steps of estimating your application's DAX traffic, selecting a cluster configuration, and testing it.

If you have an existing DAX cluster and want to evaluate whether it has the appropriate number and size of nodes, please refer to [Scaling a DAX cluster](#).

Topics

- [Overview](#)
- [Estimating traffic](#)
- [Load testing](#)

Overview

It's important to scale your DAX cluster appropriately for your workload, whether you're creating a new cluster or maintaining an existing cluster. As time goes on and your application's workload changes, you should periodically revisit your scaling decisions to make sure that they are still appropriate.

The process typically follows these steps:

1. **Estimating traffic.** In this step, you make predictions about the volume of traffic that your application will send to DAX, the nature of the traffic (read vs. write operations), and the expected cache hit rate.
2. **Load testing.** In this step, you create a cluster and send traffic to it mirroring your estimates from the previous step. Repeat this step until you find a suitable cluster configuration.
3. **Production monitoring.** While your application is using DAX in production, you should [monitor the cluster](#) to continuously validate that it is still scaled correctly as your workload changes over time.

Estimating traffic

There are three main factors that characterize a typical DAX workload:

- Cache hit rate
- [Read capacity units](#) (RCUs) per second
- [Write capacity units](#) (WCUs) per second

Estimating cache hit rate

If you already have a DAX cluster, you can use the `ItemCacheHits` and `ItemCacheMisses` [Amazon CloudWatch metrics](#) to determine the cache hit rate. The cache hit rate is equal to $\text{ItemCacheHits} / (\text{ItemCacheHits} + \text{ItemCacheMisses})$. If your workload includes `Query` or `Scan` operations, you should also look at the `QueryCacheHits`, `QueryCacheMisses`,

ScanCacheHits, and ScanCacheMisses metrics. Cache hit rates vary from one application to another and are heavily influenced by the cluster's Time to Live (TTL) setting. Typical hit rates for applications using DAX are 85–95 percent.

Estimating read and write capacity units

If you already have DynamoDB tables for your application, look at the ConsumedReadCapacityUnits and ConsumedWriteCapacityUnits [CloudWatch metrics](#). Use the Sum statistic and divide by the number of seconds in the period.

If you also already have a DAX cluster, remember that the DynamoDB ConsumedReadCapacityUnits metric only accounts for cache misses. So, to get an idea of the read capacity units per second handled by your DAX cluster, divide the number by your cache miss rate (that is, 1 - cache hit rate).

If you don't already have a DynamoDB table, see the documentation about [read capacity units](#) and [write capacity units](#) to estimate your traffic based on your application's estimated request rate, items accessed per request, and item size.

When making traffic estimates, plan for future growth and for expected and unexpected peaks to ensure that your cluster has enough headroom for traffic increases.

Load testing

The next step after estimating traffic is to test the cluster configuration under load.

1. For your initial load test, we recommend that you start with the dax.r4.large node type, the lowest-cost fixed performance, memory-optimized node type.
2. A fault-tolerant cluster requires at least three nodes, spread across three Availability Zones. In this case, if an Availability Zone becomes unavailable, the effective number of Availability Zones is reduced by one-third. For your initial load test, we recommend that you start with a two-node cluster, which simulates the failure of one Availability Zone in a three-node cluster.
3. Drive sustained traffic (as estimated in the previous step) to your test cluster for the duration of the load test.
4. Monitor the performance of the cluster during the load test.

Ideally, the traffic profile that you drive during the load test should be as similar as possible to your application's real traffic. This includes the distribution of operations (for example, 70 percent

GetItem, 25 percent Query, and 5 percent PutItem), the request rate for each operation, the number of items accessed per request, and the distribution of item sizes. To achieve a cache hit rate similar to your application's expected cache hit rate, pay close attention to the distribution of keys in your test traffic.

Note

Be careful when load testing T2 node types (dax.t2.small and dax.t2.medium). T2 node types provide [burstable CPU performance](#) that varies over time depending on the node's CPU credit balance. A DAX cluster running on T2 nodes might appear to be operating normally, but if any node is bursting above the [baseline performance](#) of its instance, the node is spending its accrued CPU credit balance. When the credit balance runs low, [performance is gradually lowered](#) to the baseline performance level.

[Monitor your DAX cluster](#) during the load test to determine whether the node type that you're using for the load test is the right node type for you. In addition, during a load test, you should monitor your request rate and cache hit rate to ensure that your test infrastructure is actually driving the amount of traffic you intend.

You should pay attention to network bytes consumption of your selected cluster instance type. Exceeding the available baseline bandwidth for an Amazon EC2 instance indicates that your cluster may not sustain your application's workload, and needs to be scaled.

If load testing indicates that the selected cluster configuration can't sustain your application's workload, you should [switch to a larger node type](#), especially if you see high CPU utilization on the primary node in the cluster, high eviction rates, or high cache memory utilization. If hit rates are consistently high, and the ratio of read to write traffic is high, you may want to consider [adding more nodes to your cluster](#). Refer to [Scaling a DAX cluster](#) for additional guidance on when to use a larger node type (vertical scaling) or add more nodes (horizontal scaling).

You should repeat your load test after making changes to your cluster configuration.

DAX API reference

For more information about Amazon DynamoDB Accelerator (DAX) APIs, see [Amazon DynamoDB accelerator](#) in the Amazon DynamoDB API Reference.

Data modeling for DynamoDB tables

Before we dive into data modeling, it's important to understand some DynamoDB fundamentals. DynamoDB is a key-value NoSQL database which allows flexible schema. The set of data attributes apart from the key attributes for each item can be either uniform or discrete. The DynamoDB key schema is in the form of either a simple primary key where a partition key uniquely identifies an item, or in the form of a composite primary key where a combination of a partition key and sort key uniquely defines an item. The partition key is hashed to determine the physical location of data and retrieve it. Therefore, it is important to choose a high cardinality and horizontally scalable attribute as a partition key to ensure even distribution of data. The sort key attribute is optional in the key schema and having a sort key enables modelling one-to-many relationships and creating item collections in DynamoDB. Sort keys are also referred to as range keys—they are used to sort items in an item collection and also allow flexible range-based operations.

For more details and best practices on DynamoDB key schema, you can refer to the following:

- [the section called “Partitions and data distribution”](#)
- [the section called “Partition key design”](#)
- [the section called “Sort key design”](#)
- [Choosing the right DynamoDB partition key](#)

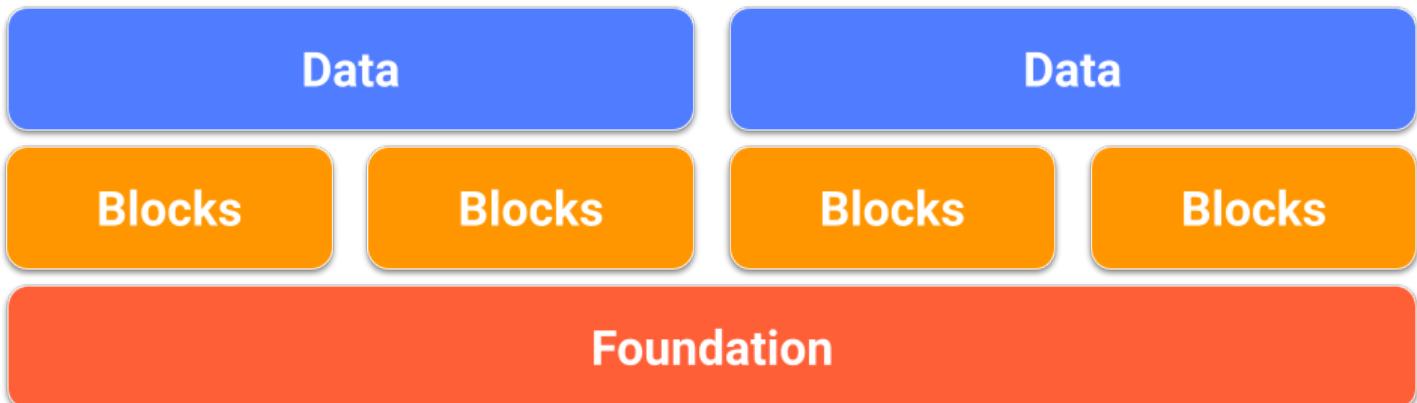
Secondary indexes are often needed to support additional query patterns in DynamoDB. Secondary indexes are shadow tables where the same data is organised via a different key schema compared to the base table. A local secondary index (LSI) shares the same partition key as the base table and allows having an alternate sort key allowing it to share the base table's capacity. A global secondary index (GSI) can have a different partition key as well as a different sort key attribute than the base table which means throughput management for a GSI is independent of the base table.

For more details on secondary indexes and best practices, you can refer to the following:

- [the section called “Working with indexes”](#)
- [the section called “Secondary indexes”](#)

Let's now look at data modeling a little closer. The process of designing a flexible and highly-optimized schema on DynamoDB, or any NoSQL database for that matter, can be a challenging

skill to learn. The goal of this module is to help you develop a mental flowchart for designing a schema that will take you from use case into production. We will start with an introduction to the foundational choice of any design, single table versus multiple table design. Then we will review the multitude of design patterns (building blocks) that can be used to achieve various organizational or performance results for your application. Finally, we are including a variety of complete schema design packages for different use cases and industries.

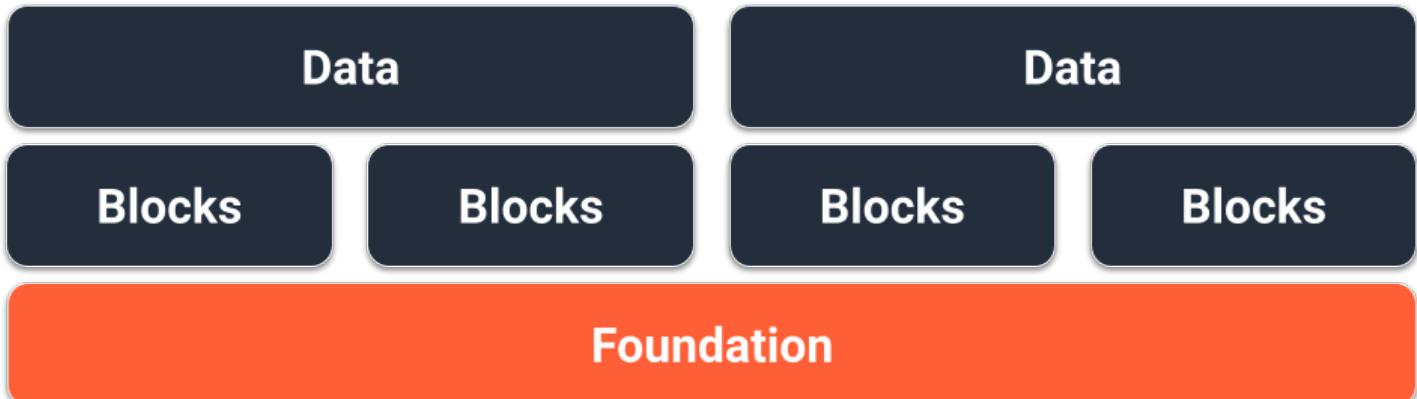


Topics

- [Data Modeling foundations in DynamoDB](#)
- [Data modeling building blocks in DynamoDB](#)
- [Data modeling schema design packages in DynamoDB](#)

Data Modeling foundations in DynamoDB

This section covers the foundation layer by examining the two types of table design: single table and multiple table.



Single table design foundation

One choice for the foundation of our DynamoDB schema is **single table design**. Single table design is a pattern that allows you to store multiple types (entities) of data in a single DynamoDB table. It aims to optimize data access patterns, improve performance, and reduce costs by eliminating the need for maintaining multiple tables and complex relationships between them. This is possible because DynamoDB stores items with the same partition key (known as an item collection) on the same partition(s) as each other. In this design, different types of data are stored as items in the same table, and each item is identified by a unique sort key.

Primary key		Attributes	
Partition key: PK	Sort key: SK		
UserID	Address#USA#CA#LA#90029	data	GSI-SK
		"Street Address"	Default
	Cart#ACTIVE#Coffee	data	GSI-SK
		CoffeeSKU	2019-11-27T103324
	Cart#ACTIVE#Spice	data	GSI-SK
		SpiceSKU	2019-11-28T091245
	Cart#SAVED#Cocoa	data	GSI-SK
		CocoaSKU	2019-11-28T125642
	OrderHistory#OrderUID	data	GSI-SK
		{Order:DataMap}	2019-10-08T132612
	ProfileName	data	
		"Paul Atreides"	
	Store#StoreUID	data	GSI-SK
		Los Angeles	Active

Advantages

- Data locality to support queries for multiple entity types in a single database call
- Reduces overall financial and latency costs of reads:
 - A single query for two items totalling less than 4KB is 0.5 RCU eventually consistent
 - Two queries for two items totalling less than 4KB is 1 RCU eventually consistent (0.5 RCU each)
 - The time to return two separate database calls will average higher than a single call

- Reduces the number of tables to manage:
 - Permissions do not need to be maintained across multiple IAM roles or policies
 - Capacity management for the table is averaged across all entities, usually resulting in a more predictable consumption pattern
 - Monitoring requires fewer alarms
 - Customer Managed Encryption Keys only need to be rotated on one table
- Smooths traffic to the table:
 - By aggregating multiple usage patterns to the same table, the overall usage tends to be smoother (the way a stock index's performance tends to be smoother than any individual stock) which works better for achieving higher utilization with provisioned mode tables

Disadvantages

- Learning curve can be steep due to paradoxical design compared to relational databases
- Data requirements must be consistent across all entity types
 - Backups are all or nothing so if some data is not mission critical, consider keeping it in a separate table
 - Table encryption is shared across all items. For multi-tenant applications with individual tenant encryption requirements, client side encryption would be required
 - Tables with a mix of historical data and operational data will not see as much of a benefit from enabling the Infrequent Access Storage Class. For more information, see [Table classes](#)
- All changed data will be propagated to DynamoDB Streams even if only a subset of entities need to be processed.
 - Thanks to Lambda event filters, this will not affect your bill when using Lambda, but will be an added cost when using the Kinesis Consumer Library
- When using GraphQL, single table design will be more difficult to implement
- When using higher-level SDK clients like Java's [DynamoDBMapper](#) or [Enhanced Client](#), it can be more difficult to process results because items in the same response may be associated with different classes

When to use

Single table design is the recommended design pattern for DynamoDB unless your use case would be impacted heavily by one of the above disadvantages. For most customers, the long term benefits outweigh the short term challenges of designing their tables this way.

Multiple table design foundation

The second choice for the foundation of our DynamoDB schema is multiple table design. Multiple table design is a pattern that is more like a traditional database design where you store a single type(entity) of data in each DynamoDB table. Data within each table will still be organized by partition key so performance within a single entity type will be optimized for scalability and performance, but queries across multiple tables must be done independently.

The screenshot shows the AWS DynamoDB Visualizer interface with two tables displayed:

Forum Table:

Primary key		Attributes			
Partition key: ForumName					
Amazon DynamoDB	Category	Threads	Messages	VIEWS	
	Amazon Web Services	2	4	1000	
Amazon Simple Notification Service	Category	Threads	Messages	VIEWS	
	Amazon Web Services	5	5	1200	
Amazon Simple Queue Service	Category	Threads	Messages	VIEWS	
	Amazon Web Services	9	6	1300	

Thread Table:

Primary key		Attributes			
Partition key: ForumName	Sort key: Subject				
Amazon DynamoDB	On-demand and transactions	Message	LastPostedBy	Replies	VIEWS
		DynamoDB on-demand and transactions now available in the AWS GovCloud (US) Regions	john@example.com	3	99
	Tagging tables	Message	LastPostedBy	Replies	VIEWS
		DynamoDB now supports tagging tables when you create them in the AWS GovCloud (US) Regions	carlos@example.com	5	30

Advantages

- Simpler to design for those who aren't used to working with single table design
- Easier implementation of GraphQL resolvers due to each resolver mapping to a single entity(table)

- Allows for unique data requirements across different entity types:
 - Backups can be made for the individual tables that are mission critical
 - Table encryption can be managed for each table. For multi-tenant applications with individual tenant encryption requirements, separate tenant tables make it possible for each customer to have their own encryption key
 - Infrequent Access Storage Class can be enabled on just the tables with historical data to realize the full cost savings benefit. For more information, see [Table classes](#)
- Each table will have its own change data stream allowing for a dedicated Lambda function to be designed for each type of item rather than a single monolithic processor

Disadvantages

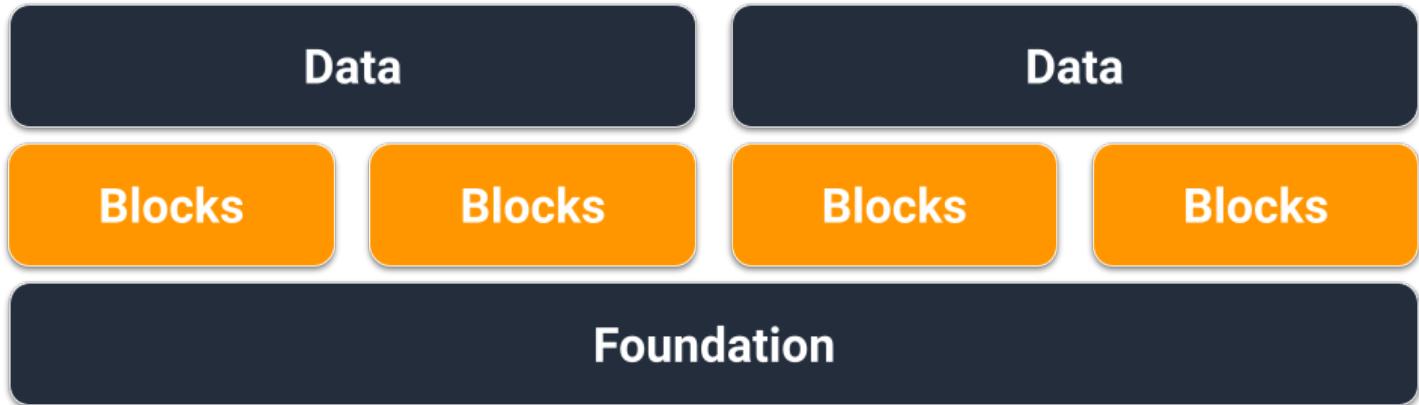
- For access patterns that require data across multiple tables, multiple reads from DynamoDB will be required and data may need to be processed/joined on the client code.
- Operations and monitoring of multiple tables requires more CloudWatch alarms and each table must be scaled independently
- Each table's permissions will need to be managed separately. The addition of tables in the future will require a change to any necessary IAM roles or policies

When to use

If your application's access patterns do not have the need to query multiple entities or tables together, then multiple table design is a good and sufficient approach.

Data modeling building blocks in DynamoDB

This section covers the building block layer to give you design patterns you can use in your application.



Topics

- [Composite sort key building block](#)
- [Multi-tenancy building block](#)
- [Sparse index building block](#)
- [Time to live building block](#)
- [Time to live for archival building block](#)
- [Vertical partitioning building block](#)
- [Write sharding building block](#)

Composite sort key building block

When people think of NoSQL, they may also think of it as non-relational. Ultimately, there is no reason relationships cannot be placed into a DynamoDB schema, they just look different than relational databases and their foreign keys. One of the most critical patterns we can use to develop a logical hierarchy of our data in DynamoDB is a composite sort key. The most common style for designing one is with each layer of the hierarchy (parent layer > child layer > grandchild layer) separated by a hashtag. For example, PARENT#CHILD#GRANDCHILD#ETC.

Primary key	
Partition key: PK	Sort key: SK
UserID	CART#ACTIVE#Apples
	CART#ACTIVE#Bananas
	CART#SAVED#Oranges
	CART#SAVED#Pears
	WISH#VEGGIES#Carrots

While a partition key in DynamoDB always requires the exact value to query for data, we can apply a partial condition to the sort key from left to right similar to traversing a binary tree.

In the example above, we have an e-Commerce store with a Shopping Cart that needs to be maintained across user sessions. Whenever the user logs in, they may want to see the entire Shopping Cart including items saved for later. But when they enter the checkout, only items in the active cart should be loaded for purchase. Since both of these KeyConditions explicitly ask for CART sort keys, the additional wishlist data is simply ignored by DynamoDB at read time. While both saved and active items are a part of the same cart, we need to treat them differently in different parts of the application, so applying a KeyCondition to the prefix of the sort key is the most optimized way of retrieving only the data needed for each part of the application.

Key features of this building block

- Related items are stored locally to each other for effective data access
- Using KeyCondition expressions, subsets of the hierarchy can be selectively retrieved meaning there are no wasted RCUs
- Different parts of the application can store their items under a specific prefix preventing overwritten items or conflicting writes

Multi-tenancy building block

Many customers use DynamoDB to host data for their multi-tenant applications. For these scenarios, we want to design the schema in a way that keeps all data from a single tenant in its own logical partition of the table. This leverages the concept of an Item Collection, which is a

term for all items in a DynamoDB table with the same partition key. For more information on how DynamoDB approaches multitenancy, see [Multitenancy on DynamoDB](#).

Primary key		Attributes
Partition key: PK	Sort key: SK	
UserOne	PhotoID1	ImageURL https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
	PhotoID2	ImageURL https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserTwo	PhotoID3	ImageURL https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
	PhotoID4	ImageURL https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserThree	PhotoID5	ImageURL https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]

For this example, we are running a photo hosting site with potentially thousands of users. Each user will only upload photos to their own profile initially, but by default we will not allow a user to see the photos of any other user. An additional level of isolation would ideally be added to the authorization of each user's call to your API to ensure they are only requesting data from their own partition, but at the schema level, unique partition keys is adequate.

Key features of this building block

- The amount of data read by any one user or tenant can only be as much as the total amount of items in their partition
- Removal of a tenant's data due to an account closure or compliance request can be done tactfully and cheaply. Simply run a query where the partition key equals their tenant ID, then execute a `DeleteItem` operation for each primary key returned

Note

Designed with multi-tenancy in mind, you can use different encryption key providers across a single table to safely isolate data. [AWS Database Encryption SDK](#) for Amazon DynamoDB enables you to include client-side encryption in your DynamoDB workloads. You can perform attribute-level encryption, enabling you to encrypt specific attribute values

before storing them in your DynamoDB table and search on encrypted attributes without decrypting the entire database beforehand.

Sparse index building block

Sometimes an access pattern requires looking for items that match a rare item or an item that receives a status (which requires an escalated response). Rather than regularly query across the entire dataset for these items, we can leverage the fact that **global secondary indexes (GSI)** are sparsely loaded with data. This means that only items in the base table that have the attributes defined in the index will be replicated to the index.

Primary key		Attributes	
Partition key: DeviceID	Sort key: State#Date		
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date
		Liz	2020-04-24
	WARNING1#2020-04-24T14:45:00	Operator	Date
d#54321		Liz	2020-04-24
	WARNING1#2020-04-24T14:50:00	Operator	Date
		Liz	2020-04-24
d#11223	NORMAL#2020-04-11T06:00:00	Operator	Date
		Liz	2020-04-11
	NORMAL#2020-04-11T09:30:00	Operator	Date
		Sue	2020-04-11
	WARNING2#2020-04-11T09:25:00	Operator	Date
d#11223		Sue	2020-04-11
	WARNING3#2020-04-11T05:55:00	Operator	Date
		Liz	2020-04-11
	WARNING4#2020-04-27T16:10:00	Operator	Date
d#11223		Sue	2020-04-27
	WARNING4#2020-04-27T16:15:00	Operator	Date
		Sue	2020-04-27
			EscalatedTo
			Sara

Primary key		Attributes	
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue

In this example, we see an IOT use case where each device in the field is reporting back a status on a regular basis. For the majority of the reports we expect the device to report everything is okay, but on occasion there can be a fault and it must be escalated to a repair technician. For reports with an escalation, the attribute EscalatedTo is added the item, but is not present otherwise. The GSI in this example is partitioned by EscalatedTo and since the GSI brings over keys from the base table we can still see which DeviceID reported the fault and at what time.

While reads are cheaper than writes in DynamoDB, sparse indexes are a very powerful tool for use cases where instances of a specific type of item is rare but reads to find them are common.

Key features of this building block

- Write and storage costs for the sparse GSI only apply to items that match the key pattern, so the cost of the GSI can be substantially less than other GSIs that have all items replicated to them
- A composite sort key can still be used to further narrow down the items that match the desired query, for instance, a timestamp could be used for the sort key to only view faults reported in the last X minutes (SK > 5 minutes ago, ScanIndexForward: False)

Time to live building block

Most data have some duration of time for which it can be considered worth keeping in a primary datastore. To facilitate data aging out from DynamoDB, it has a feature called **time to live (TTL)**. The [TTL](#) feature allows you to define a specific attribute at the table level that needs monitoring for items with an epoch timestamp (that's in the past). This allows you to delete expired records from the table for free.

 **Note**

If you are using [Global Tables version 2019.11.21 \(Current\)](#) of global tables and you also use the [Time to Live](#) feature, DynamoDB replicates TTL deletes to all replica tables. The initial TTL delete does not consume write capacity in the Region in which the TTL expiry occurs. However, the replicated TTL delete to the replica table(s) consumes replicated write capacity in each of the replica Regions and applicable charges will apply.

Primary key		Attributes	
Partition key: PK	Sort key: MessageTimestamp		
UserID	2030-06-30T12:12:12	TTL	Message
		1909570332	Hello
	2030-06-30T12:17:22	TTL	Message
		1909570647	DynamoDB
	2030-06-30T12:22:27	TTL	Message
		1909570947	TTL

In this example, we have an application designed to let a user create messages that are short-lived. When a message is created in DynamoDB, the TTL attribute is set to a date seven days in the future by the application code. In roughly seven days, DynamoDB will see that the epoch timestamp of these items is in the past and delete them.

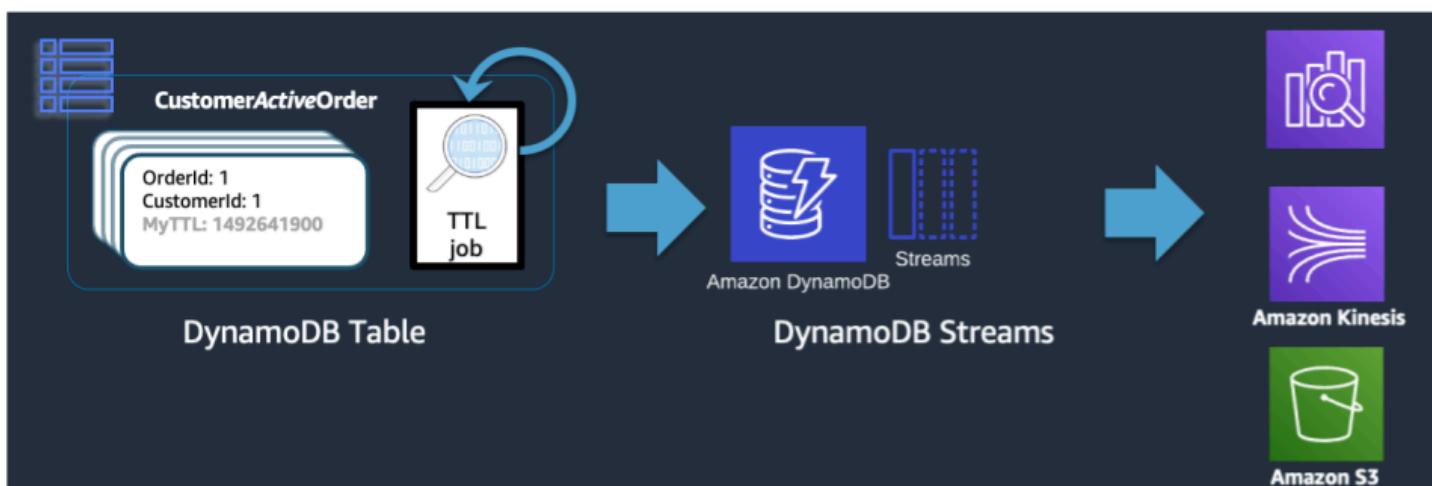
Since the deletes done by TTL are free, it is strongly recommended to use this feature to remove historical data from the table. This will reduce the overall storage bill each month and will likely reduce the costs of user reads since there will be less data to be retrieved by their queries. While TTL is enabled at the table level, it is up to you which items or entities to create a TTL attribute for and how far into the future to set the epoch timestamp to.

Key features of this building block

- TTL deletes are run behind the scenes with no impact to your table performance
- TTL is an asynchronous process that runs roughly every six hours, but can take over 48 hours for an expired record to be deleted
 - Do not rely on TTL deletes for use cases like lock records or state management if stale data must be cleaned up in less than 48 hours
- You can name the TTL attribute a valid attribute name, but the value must be a number type

Time to live for archival building block

While TTL is an effective tool for deleting older data from DynamoDB, many use cases require an archive of the data be kept for a longer period of time than the primary datastore. In this instance, we can leverage TTL's timed deletion of records to push expired records into a long-term datastore.



When a TTL delete is done by DynamoDB, it is still pushed into the DynamoDB Stream as a Delete event. When DynamoDB TTL is the one who performs the delete though, there is an attribute on the stream record of `principal:dynamodb`. Using a Lambda subscriber to the DynamoDB Stream, we can apply an event-filter for only the DynamoDB principal attribute and know that any records that match that filter are to be pushed to an archival store like S3 Glacier.

Key features of this building block

- Once the low-latency reads of DynamoDB are no longer needed for the historical items, migrating them to a colder storage service like S3 Glacier can reduce storage costs significantly while meeting the data compliance needs of your use case
- If the data is persisted into Amazon S3, cost-efficient analytics tools like Amazon Athena or Redshift Spectrum can be used to perform historical analysis of the data

Vertical partitioning building block

Users familiar with a document model database will be familiar with the idea of storing all related data within a single JSON document. While DynamoDB supports JSON data types, it does not support executing KeyConditions on nested JSON. Since KeyConditions are what dictate how much data is read from disk and effectively how many RCUs a query consumes, this can result in inefficiencies at scale. To better optimize the writes and reads of DynamoDB, we recommend breaking apart the document's individual entities into individual DynamoDB items, also referred to as **vertical partitioning**.

```
{  
    "UserProfile" : {  
        "FirstName": "Paul",  
        "LastName": "Atreides",  
        "DateJoined": "1965-08-01"},  
    "Store" : {  
        "store_id": "STOREUID",  
        "city": "Los Angeles",  
        "zip_code": "90029"}  
    "ShoppingCart" : [  
        {"Spice":  
            { "SKU": "SpicesSKU",  
                "CategoryID": "FictionalSpice",  
                "DateAdded": "2019-06-11"}},  
        {"EspressoBeans":  
            { "SKU": "CaffeineSKU",  
                "CategoryID": "FOODANDDRINK",  
                "DateAdded": "2019-06-10"}}],  
    "ShippingAddress" : {  
        "street_address": "1234 Arrakis Dr",  
        "city": "Los Angeles",  
        "zip_code": "90029",  
        "status": "default"}  
    "OrderHistory#OrderUID" : {  
        "ProductA": "SKU_A",  
        "ProductB": "SKU_B",  
        "DateOrdered": "2018-09-28"}  
}
```

Primary key		Attributes	
Partition key: PK	Sort key: SK		
UserID	Address#USA#CA#LA#90029	data	GSI-SK
		"Street Address"	Default
	Cart#ACTIVE#Coffee	data	GSI-SK
		CoffeeSKU	2019-11-27T103324
	Cart#ACTIVE#Spice	data	GSI-SK
		SpiceSKU	2019-11-28T091245
	Cart#SAVED#Cocoa	data	GSI-SK
		CocoaSKU	2019-11-28T125642
ProfileName	OrderHistory#OrderUID	data	GSI-SK
		{Order:DataMap}	2019-10-08T132612
	Store#StoreUID	data	GSI-SK
		Los Angeles	Active

Vertical partitioning, as shown above, is a key example of single table design in action but can also be implemented across multiple tables if desired. Since DynamoDB bills writes in 1KB increments, you should ideally partition the document in a way that results in items under 1KB.

Key features of this building block

- A hierarchy of data relationships is maintained via sort key prefixes so the singular document structure could be rebuilt client-side if needed
- Singular components of the data structure can be updated independently resulting in small item updates being only 1 WCU
- By using the sort key `BeginsWith`, the application can retrieve similar data in a single query, aggregating read costs for reduced total cost/latency
- Large documents can easily be larger than the 400 KB individual item size limit in DynamoDB and vertical partitioning helps work around this limit

Write sharding building block

One of the very few hard limits DynamoDB has in place is the restriction of how much throughput a single physical partition can maintain per second (not necessarily a single partition key). These limits are presently:

- 1000 WCU (or 1000 <=1KB items written per second) and 3000 RCU (or 3000 <=4KB reads per second) *strongly consistent* or
- 6000 <=4KB reads per second *eventually consistent*

In the event requests against the table exceed either of these limits, an error is sent back to the client SDK of `ThroughputExceededException`, more commonly referred to as throttling. Use cases that require read operations beyond that limit will mostly be served best by placing a read cache in front of DynamoDB, but write operations require a schema level design known as **write sharding**.



Primary Key	Attributes	
Partition Key: Candidate		
CandidateA#1	Vote-Counter	Last-Update
	10238	2019-09-30T11:35:53
CandidateA#2	Vote-Counter	Last-Update
	8452	2019-09-30T11:35:53
CandidateA#3	Vote-Counter	Last-Update
	9148	2019-09-30T11:35:53
CandidateA#4	Vote-Counter	Last-Update
	11092	2019-09-30T11:35:53

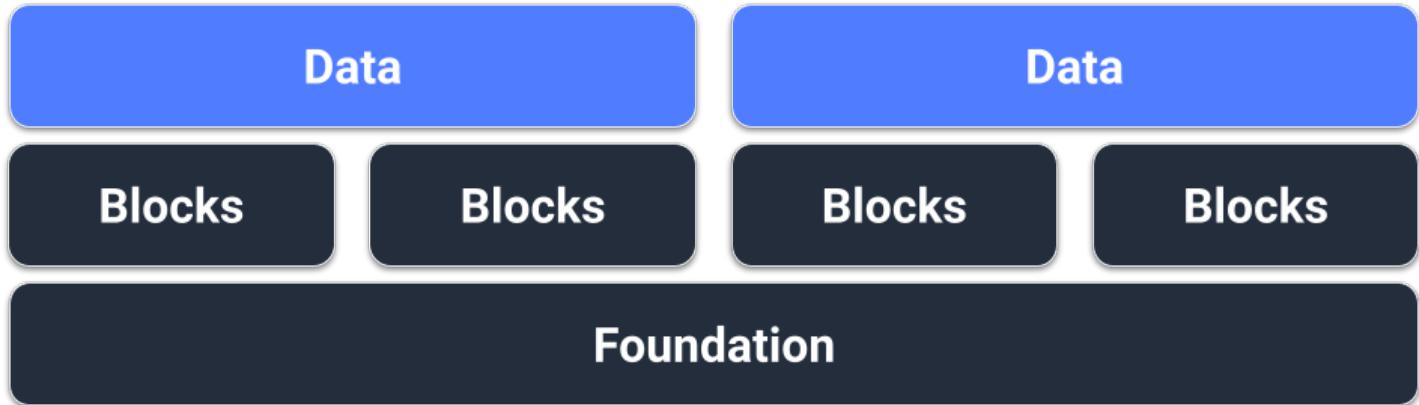
To solve this problem, we'll append a random integer onto the end of the partition key for each contestant in the application's `UpdateItem` code. The range of the random integer generator will need to have an upper bound matching or exceeding the expected amount of writes per second for a given contestant divided by 1000. To support 20,000 votes per second, it would look like `rand(0,19)`. Now that the data is stored under separate logical partitions, it must be combined back together at read time. Since vote totals doesn't need to be real time, a Lambda function scheduled to read all vote partitions every X minutes could perform occasional aggregation for each contestant and write it back to a single vote total record for live reads.

Key features of this building block

- For use cases with extremely high write throughput for a given partition key that cannot be avoided, write operations can be artificially spread across multiple DynamoDB partitions
- GSI with a low cardinality partition key should also utilize this pattern since throttling on a GSI will apply backpressure to write operations on the base table

Data modeling schema design packages in DynamoDB

This section covers the data layer to go over different examples you can use in your DynamoDB table design. Each example will go over their use case, access patterns, how to achieve the access patterns, and then what the final schema will look like.



Prerequisites

Before we attempt to design our schema for DynamoDB, we must first gather some prerequisite data on the use case the schema needs to support. Unlike relational databases, DynamoDB is sharded by default, meaning that the data will live on multiple servers behind the scenes so designing for data locality is important. We'll need to put together the following list for each schema design:

- List of entities (ER Diagram)
- Estimated volumes and throughput for each entity
- Access patterns that need to be supported (queries and writes)
- Data retention requirements

Topics

- [Social network schema design in DynamoDB](#)
- [Gaming profile schema design in DynamoDB](#)
- [Complaint management system schema design in DynamoDB](#)
- [Recurring payments schema design in DynamoDB](#)
- [Monitoring device status updates in DynamoDB](#)
- [Using DynamoDB as a data store for an online shop](#)

Social network schema design in DynamoDB

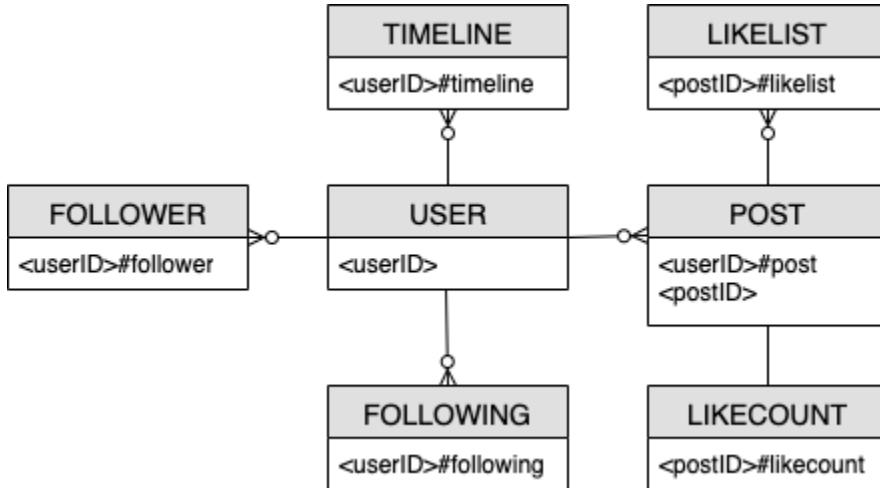
Social network business use case

This use case talks about using DynamoDB as a social network. A social network is an online service that lets different users interact with each other. The social network we'll design will let the user see a timeline consisting of their posts, their followers, who they are following, and the posts written by who they are following. The access patterns for this schema design are:

- Get user information for a given userID
- Get follower list for a given userID
- Get following list for a given userID
- Get post list for a given userID
- Get user list who likes the post for a given postID
- Get the like count for a given postID
- Get the timeline for a given userID

Social network entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the social network schema design.



Social network access patterns

These are the access patterns we'll be considering for the social network schema design.

- `getUserInfoByUserID`

- `getFollowerListByUserID`
- `getFollowingListByUserID`
- `getPostListByUserID`
- `getUserLikesByPostID`
- `getLikeCountByPostID`
- `getTimelineByUserID`

Social network schema design evolution

DynamoDB is a NoSQL database, so it does not allow you to perform a join - an operation that combines data from multiple databases. Customers unfamiliar with DynamoDB might apply relational database management system (RDBMS) design philosophies (such as creating a table for each entity) to DynamoDB when they do not need to. The purpose of DynamoDB's single-table design is to write data in a pre-joined form according to the application's access pattern, and then immediately use the data without additional computation. For more information, see [Single-table vs. multi-table design in DynamoDB](#).

Now, let's step through how we'll evolve our schema design to address all the access patterns.

Step 1: Address access pattern 1 (`getUserInfoByUserID`)

To get a given user's information, we'll need to [Query](#) the base table with a key condition of `PK=<userID>`. The query operation lets you paginate the results, which can be useful when a user has many followers. For more information on Query, see [Query operations in DynamoDB](#).

In our example, we track two types of data for our user: their "count" and their "info." A user's "count" reflects how many followers they have, how many users they are following, and how many posts they've created. A user's "info" reflects their personal information such as their name.

We see these two kinds of data represented by the two items below. The item that has "count" in its sort key (SK) is more likely to change than the item with "info." DynamoDB considers the size of the item as it appears before and after the update and the provisioned throughput consumed will reflect the larger of these item sizes. So even if you update just a subset of the item's attributes, [UpdateItem](#) will still consume the full amount of provisioned throughput (the larger of the before and after item sizes). You can get the items via a single Query operation and use `UpdateItem` to add or subtract from existing numeric attributes.

Primary key		Attributes			
Partition key: PK	Sort key: SK	follower#	following#	post#	
u#12345	"count"	3000000000	971	4945	
		hyuklee	My name is Hyuk Lee	s3://....	...
	"info"	name	content	imageUrl	etc
		hyuklee			

Step 2: Address access pattern 2 (getFollowerListByUserID)

To get a list of users who are following a given user, we'll need to Query the base table with a key condition of `PK=<userID>#follower`.

Primary key		Attributes			
Partition key: PK	Sort key: SK	follower#	following#	post#	
u#12345	"count"	3000000000	971	4945	
		hyuklee	My name is Hyuk Lee	s3://....	...
	"info"	name	content	imageUrl	etc
		hyuklee			
u#12345#follower	u#23456 u#34567 u#45678				

Step 3: Address access pattern 3 (getFollowingListByUserID)

To get a list of users a given user is following, we'll need to Query the base table with a key condition of `PK=<userID>#following`. You can then use a [TransactWriteItems](#) operation group up several requests together and do the following:

- Add User A to User B's follower list, and then increment User B's follower count by one
- Add User B to User A's follower list, and then increment User A's follower count by one

Primary key		Attributes			
Partition key: PK	Sort key: SK	follower#	following#	post#	
u#12345	"count"	3000000000	971	4945	
		hyuklee	My name is Hyuk Lee	s3://....	etc
	"info"	u#23456			
		u#34567			
u#12345#follower	u#45678				
	u#56789				
	u#67890				
	u#78912				
u#12345#following	u#23456				
	u#34567				
	u#45678				
	u#56789				
u#12345#post	u#67890				
	u#78912				

Step 4: Address access pattern 4 (getPostListByUserID)

To get a list of posts created by a given user, we'll need to Query the base table with a key condition of `PK=<userID>#post`. One important thing to note here is that a user's postIDs must be incremental: the second postID value must be greater than the first postID value (since users want to see their posts in a sorted manner). You can do this by generating postIDs based on a time value like a Universally Unique Lexicographically Sortable Identifier (ULID).

Primary key		Attributes			
Partition key: PK	Sort key: SK	follower#	following#	post#	
u#12345	"count"	3000000000	971	4945	
		hyuklee	My name is Hyuk Lee	s3://....	etc
	"info"	u#23456			
		u#34567			
u#12345#follower	u#45678				
	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
	p#12345	content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	

Step 5: Address access pattern 5 (getUserLikesByPostID)

To get a list of users who liked a given user's post, we'll need to Query the base table with a key condition of `PK=<postID>#likelist`. This approach is the same pattern that we used for retrieving the follower and following lists in access pattern 2 (`getFollowerListByUserID`) and access pattern 3 (`getFollowingListByUserID`).

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				

Step 6: Address access pattern 6 (`getLikeCountByPostID`)

To get a count of likes for a given post, we'll need to perform a [GetItem](#) operation on the base table with a key condition of `PK=<postID>#likecount`. This access pattern can cause throttling issues whenever a user with many followers (such as a celebrity) creates a post since throttling occurs when a partition's throughput exceeds 1000 WCU per second. This problem is not a result of DynamoDB, it just appears in DynamoDB since it's at the end of the software stack.

You should evaluate whether it's really essential for all users to view the like count simultaneously or if it can happen gradually over time. In general, a post's like count doesn't need to be immediately 100% accurate. You can implement this strategy by putting a queue between your application and DynamoDB to have the updates happen periodically.

Primary key		Attributes			
Partition key: PK	Sort key: SK	follower#	following#	post#	
u#12345	"count"	3000000000	971	4945	
		hyuklee	My name is Hyuk Lee	s3://....	etc
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			

Step 7: Address access pattern 7 (getTimelineByUserID)

To get the timeline for a given user, we'll need to perform a Query operation on the base table with a key condition of `PK=<userID>#timeline`. Let's consider a scenario where a user's followers need to view their post synchronously. Every time a user writes a post, their follower list is read and their userID and postID are slowly entered into the timeline key of all its followers. Then, when your application starts, you can read the timeline key with the Query operation and fill the timeline screen with a combination of userID and postID using the [BatchGetItem](#) operation for any new items. You cannot read the timeline with an API call, but this is a more cost effective solution if the posts could be edited frequently.

The timeline is a place that shows recent posts, so we'll need a way to clean up the old ones. Instead of using WCU to delete them, you can use DynamoDB's [TTL](#) feature to do it for free.

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
u#12345#follower	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#following	u#23456				
	u#34567				
	u#45678				
u#12345#post	u#56789				
	u#67890				
	u#78912				
p#12345#likelist	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likecount	u#23456				
	u#34567				
	u#45678				
p#12345#timeline	"count"	etc			
		100			
	p#34567#u#56789	ttl			
		1571827560			
	p#45678#u#67890	ttl			
		1571827560			
	p#56789#u#78901	ttl			
		1571827560			

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getUserInfoByUserID	Base table	Query	PK=<userID>		

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getFollow erListByU serID	Base table	Query	PK=<userl D>#follower		
getFollow ingListBy UserID	Base table	Query	PK=<userl D>#following		
getPostLi stByUserID	Base table	Query	PK=<userl D>#post		
getUserLi kesByPostID	Base table	Query	PK=<postl D>#likelist		
getLikeCo untByPostID	Base table	GetItem	PK=<postl D>#likecount		
getTimeli neByUserID	Base table	Query	PK=<userl D>#timeline		

Social network final schema

Here is the final schema design. To download this schema design as a JSON file, see [DynamoDB Examples](#) on GitHub.

Base table:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			
u#12345#timeline	p#34567#u#56789	ttl			
		1571827560			
	p#45678#u#67890	ttl			
		1571827560			
	p#56789#u#78901	ttl			
		1571827560			

Using NoSQL Workbench with this schema design

You can import this final schema into [NoSQL Workbench](#), a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see [the section called “Download”](#).
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see [the section called “Importing an existing model”](#).

4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see [the section called "Editing an existing model"](#).
5. To visualize your data model, add sample data, or import sample data from a CSV file, use the [Data Visualizer](#) feature of NoSQL Workbench.

Gaming profile schema design in DynamoDB

Gaming profile business use case

This use case talks about using DynamoDB to store player profiles for a gaming system. Users (in this case, players) need to create profiles before they can interact with many modern games, especially online ones. Gaming profiles typically include the following:

- Basic information such as their user name
- Game data such as items and equipment
- Game records such as tasks and activities
- Social information such as friend lists

To meet the fine-grained data query access requirements for this application, the primary keys (partition key and sort key) will use generic names (PK and SK) so they can be overloaded with various types of values as we will see below.

The access patterns for this schema design are:

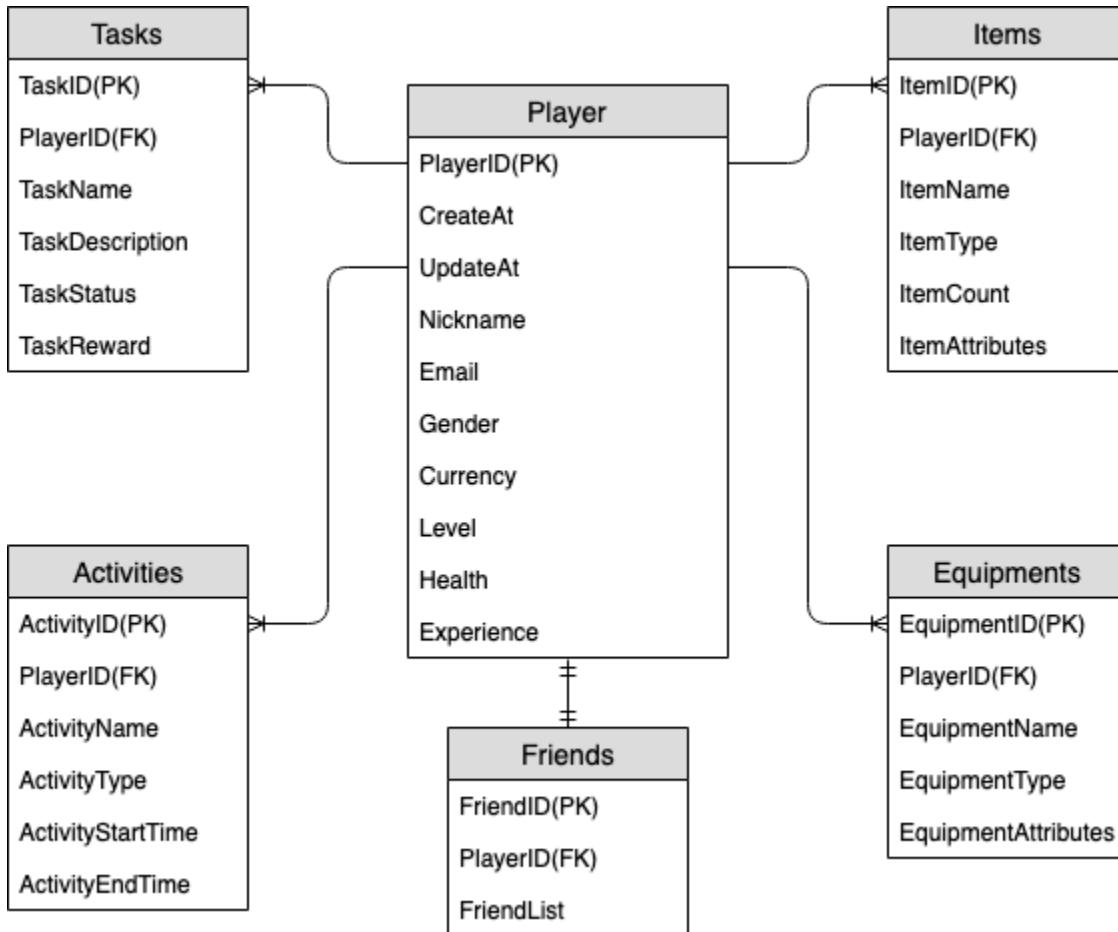
- Get a user's friend list
- Get all of a player's information
- Get a user's item list
- Get a specific item from the user's item list
- Update a user's character
- Update the item count for a user

The size of the gaming profile will vary in different games. [Compressing large attribute values](#) can let them fit within item limits in DynamoDB and reduce costs. The throughput management strategy would depend various on factors such as: number of players, number of games played

per second, and seasonality of the workload. Typically for a newly launched game, the number of players and the level of popularity are unknown so we will start with the [on-demand throughput mode](#).

Gaming profile entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the gaming profile schema design.



Gaming profile access patterns

These are the access patterns we'll be considering for the social network schema design.

- getPlayerFriends
- getPlayerAllProfile
- getPlayerAllItems
- getPlayerSpecificItem
- updateCharacterAttributes

- updateItemCount

Gaming profile schema design evolution

From the above ERD, we can see that this is a one-to-many relationship type of data modeling. In DynamoDB, one-to-many data models can be organized into item collections, which is different from traditional relational databases where multiple tables are created and linked through foreign keys. An [item collections](#) is a group of items that share the same partition key value but have different sort key values. Within an item collection, each item has a unique sort key value that distinguishes it from other items. With this in mind, let's use the following pattern for HASH and RANGE values for each entity type.

To begin, we use generic names like PK and SK to store different types of entities in the same table to make the model future-proof. For better readability, we can include prefixes to denote the type of data or include an arbitrary attribute called Entity_type or Type. In the current example, we use a string starting with player to store player_ID as the PK; use entity_name# as the prefix of SK, and add a Type attribute to indicate which entity type this piece of data is. This allows us to support storing more entity types in the future, and use advanced technologies such as GSI Overloading and Sparse GSI to meet more access patterns.

Let's start implementing the access patterns. Access patterns such as adding players and adding equipment can be realized through the [PutItem](#) operation, so we can ignore them. In this document, we'll focus on the typical access patterns listed above.

Step 1: Address access pattern 1 (getPlayerFriends)

We address access pattern 1 (getPlayerFriends) with this step. In our current design, friendship is simple and the number of friends in the game is small. For simplicity's sake, we use a list data type to store friend lists (1:1 modeling). In this design, we use [GetItem](#) to satisfy this access pattern. In the GetItem operation, we explicitly provide the partition key and sort key value to get a specific item.

However, if a game has a large number of friends, and the relationships between them are complex (such as friendships being bi-directional with both an invite and accept component) it would be necessary to use a many-to-many relationship to store each friend individually, in order to scale to an unlimited friend list size. And if the friendship change involves operating on multiple items at the same time, DynamoDB transactions can be used to group multiple actions together and submit them as a single all-or-nothing [TransactWriteItems](#) or [TransactGetItems](#) operation.

Primary key		Attributes	
Partition key: PK	Sort key: SK	Type	
player001	FRIENDS#player001	Type	FriendList
		Friends	[{"M": {"FriendId": {"S": "player002"}, "FriendName": {"S": "Alice"}}, {"M": {"FriendId": {"S": "player003"}, "FriendName": {"S": "Bob"}}}]

Step 2: Address access patterns 2 (getPlayerAllProfile), 3 (getPlayerAllItems), and 4 (getPlayerSpecificItem)

We address access patterns 2 (getPlayerAllProfile), 3 (getPlayerAllItems), and 4 (getPlayerSpecificItem) using this step. What these three access patterns have in common is a range query, which uses the [Query](#) operation. Depending on the scope of the query, [Key Condition](#) and [Filter Expressions](#) are used, which are commonly used in practical development.

In the Query operation, we provide a single value for Partition Key and get all items with that Partition Key value. Access pattern 2 (getPlayerAllProfile) is implemented in this way. Optionally, we can add a sort key condition expression — a string that determines the items to be read from the table. Access pattern 3 (getPlayerAllItems) is implemented by adding the key condition of sort key begins_with ITEMS#. Further, in order to simplify the development of the application side, we can use filter expressions to implement access pattern 4 (getPlayerSpecificItem).

Here's a pseudocode example using filter expression that filters items of the Weapon category:

```
filterExpression: "ItemType = :itemType"
expressionAttributeValues: {":itemType": "Weapon"}
```

Primary key		Attributes				
Partition key: PK	Sort key: SK	Type	ItemName	ItemType	ItemCount	ItemAttributes
player001	ITEMS#001	Type	Health Potion	Consumable	5	{"M":{"HP": {"N":"50"}}}
		Item	Armor of the Knight	Armor	1	{"M":{"DEF": {"N":"100"}}}
	ITEMS#002	Type	Sword of the Dragon	Weapon	1	{"M":{"ATK": {"N":"100"}, "DEF": {"N":"50"}}}
		Item	Armor of the Knight	Armor	1	{"M":{"DEF": {"N":"100"}}}

Note

A filter expression is applied after a Query finishes, but before the results are returned to the client. Therefore, a Query consumes the same amount of read capacity regardless of whether a filter expression is present.

If the access pattern is to query a large dataset and filter out a large amount of data to keep only a small subset of data, the appropriate approach is to design DynamoDB Partition Key and Sort Key more effectively. For example, in the above example for obtaining a certain ItemType, if there are many items for each player and querying for a certain ItemType is a typical access pattern, it would be more efficient to bring ItemType into the SK as a composite key. The data model would look like this: ITEMS#ItemType#ItemId.

Step 3: Address access patterns 5 (updateCharacterAttributes) and 6 (updateItemCount)

We address access patterns 5 (updateCharacterAttributes) and 6 (updateItemCount) using this step. When the player needs to modify the character, such as reducing the currency, or modifying the quantity of a certain weapon in their items, use [UpdateItem](#) to implement these access patterns. To update a player's currency but ensure it never goes below a minimum amount, we can add a [the section called “Condition expressions”](#) to reduce the balance only if it's greater than or equal to the minimum amount. Here is a pseudocode example:

```
UpdateExpression: "SET currency = currency - :amount"
ConditionExpression: "currency >= :minAmount"
```

Primary key		Attributes										
Partition key: PK	Sort key: SK	Type	CreatedAt	UpdatedAt	Nickname	Email	Gender	Avatar	Currency	PlayerLevel	PlayerHealth	PlayerExperience
player001	#METADATA #player001	Metadata	1618500000	1620000000	John	john@example.com	male	s3://gaming-beki65wn3bgc-lb-avatar/player001.png	500 Updated to 500-Amount	10	80	1000

When developing with DynamoDB and using [Atomic Counters](#) to decrement inventory, we can ensure idempotency by using optimistic locking. Here is a pseudocode example for Atomic Counters:

```
UpdateExpression: "SET ItemCount = ItemCount - :incr"
expression-attribute-values: '{":incr":{"N":"1"}}'
```

Primary key		Attributes						
Partition key: PK	Sort key: SK	Type	ItemName	ItemType	ItemCount	ItemAttributes		
player001	ITEMS#001	Item	Health Potion	Consumable	5 Updated to 4	{"M":{"HP": {"N":"50"}}}		

In addition, in a scenario where the player purchases an item with currency, the entire process needs to deduct currency and add an item at the same time. We can use DynamoDB Transactions to group multiple actions together and submit them as a single all-or-nothing TransactWriteItems or TransactGetItems operation. TransactWriteItems is a synchronous and idempotent write operation that groups up to 100 write actions in a single all-or-nothing operation. The actions are completed atomically so that either all of them succeed or none of them succeeds. Transactions help eliminate the risk of duplication or vanishing currency. For more information on transactions, see [DynamoDB transactions example](#).

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getPlayerFriends	Base table	GetItem	PK=PlayerID	SK="FRIENDS#playerID"	

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getPlayerAllProfile	Base table	Query	PK=PlayerID		
getPlayerAllItems	Base table	Query	PK=PlayerID	SK begins_with "ITEMS#"	
getPlayerSpecificItem	Base table	Query	PK=PlayerID	SK begins_with "ITEMS#"	filterExpression: "ItemType" = :itemType expressionAttributeValues: { ":itemType": "Weapon" }
updateCharacterAttributes	Base table	UpdateItem	PK=PlayerID	SK="#METADATA#playerID"	UpdateExpression: "SET currency = currency - :amount" Condition Expression: "currency >= :minAmount"

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
updateItem mCount	Base table	UpdateItem	PK=PlayerID	SK = "ITEMS#ItemID"	update-expression: "SET ItemCount = ItemCount - :incr" expression-attribute-values : '{":incr": {"N":"1"}}'

Gaming profile final schema

Here is the final schema design. To download this schema design as a JSON file, see [DynamoDB Examples](#) on GitHub.

Base table:

Primary key		Attributes										
Partition key: PK	Sort key: SK	Type	CreatedAt	UpdatedAt	Nickname	Email	Gender	Avatar	Currency	PlayerLevel	PlayerHealth	PlayerExperience
player001	#METADATA#player001	Metadata	1618500000	1620000000	John	john@example.com	male	s3://gaming-blki65wn3bgclob-avatar/playerr001.png	500	10	80	1000
	ACTIVITY#001	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
	ACTIVITY#002	Activity	1647475199	Hunting Trip	{"M":{"Gold":{"N":"50"}}, "XP":{"N":"200"}}}	1647388800	Hunting					
	ACTIVITY#003	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
	EQUIPMENT#001	Activity	1647647999	Mining Adventure	{"M":{"Gold":{"N":"1000"}}, "XP":{"N":"500"}}}	1647561600	Mining					
	EQUIPMENT#002	Type	ActivityEndTime	Arena Challenge	{"M":{"Gold":{"N":"2000"}}, "XP":{"N":"1000"}}}	1647734400	Arena					
	EQUIPMENT#003	Type	EquipmentName	EquipmentType	EquipmentAttributes							
	FRIENDS#player001	Equipment	Sword of the Dragon	Weapon	{"M":{"ATK":{"N":"100"}, "DEF":{"N":"50"}}}							
	ITEMS#001	Type	EquipmentName	EquipmentType	EquipmentAttributes							
	ITEMS#002	Equipment	Armor of the Knight	Armor	{"M":{"DEF":{"N":"100"}}}							
	ITEMS#003	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Ring of the Mage	Accessory	{"M":{"SP":{"N":"50"}}}							
player001	FRIENDS#player001	Type	FriendList									
		Friends	[{"M": {"FriendId": "player002", "S": "player002", "FriendName": "Alice"}, {"M": {"FriendId": "player003", "S": "player003", "FriendName": "Bob"}}]									
	ITEMS#001	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Health Potion	Consumable	5	{"M":{"HP":{"N":"50"}}}						
	ITEMS#002	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Armor of the Knight	Armor	1	{"M":{"DEF":{"N":"100"}}}						
	ITEMS#003	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Sword of the Dragon	Weapon	1	{"M":{"ATK":{"N":"100"}, "DEF":{"N":"50"}}}						
	TASK#001	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Find the Lost Treasure	Find the Lost Treasure	InProgress	{"M":{"Gold":{"N":"100"}, "XP":{"N":"50"}}}						
	TASK#002	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Defeat Magic Monsters	Go to the Magic Forest and defeat three magic monsters.	Completed	{"M":{"Gold":{"N":"200"}, "XP":{"N":"100"}}}						
	TASK#003	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Rescue the Princess	Go to the Demon King's Castle and rescue the princess who is being held captive by the Demon King.	Available	{"M":{"Gold":{"N":"500"}, "XP":{"N":"200"}}}						

Using NoSQL Workbench with this schema design

You can import this final schema into [NoSQL Workbench](#), a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see [the section called “Download”](#).
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see [the section called “Importing an existing model”](#).
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see [the section called “Editing an existing model”](#).
5. To visualize your data model, add sample data, or import sample data from a CSV file, use the [Data Visualizer](#) feature of NoSQL Workbench.

Complaint management system schema design in DynamoDB

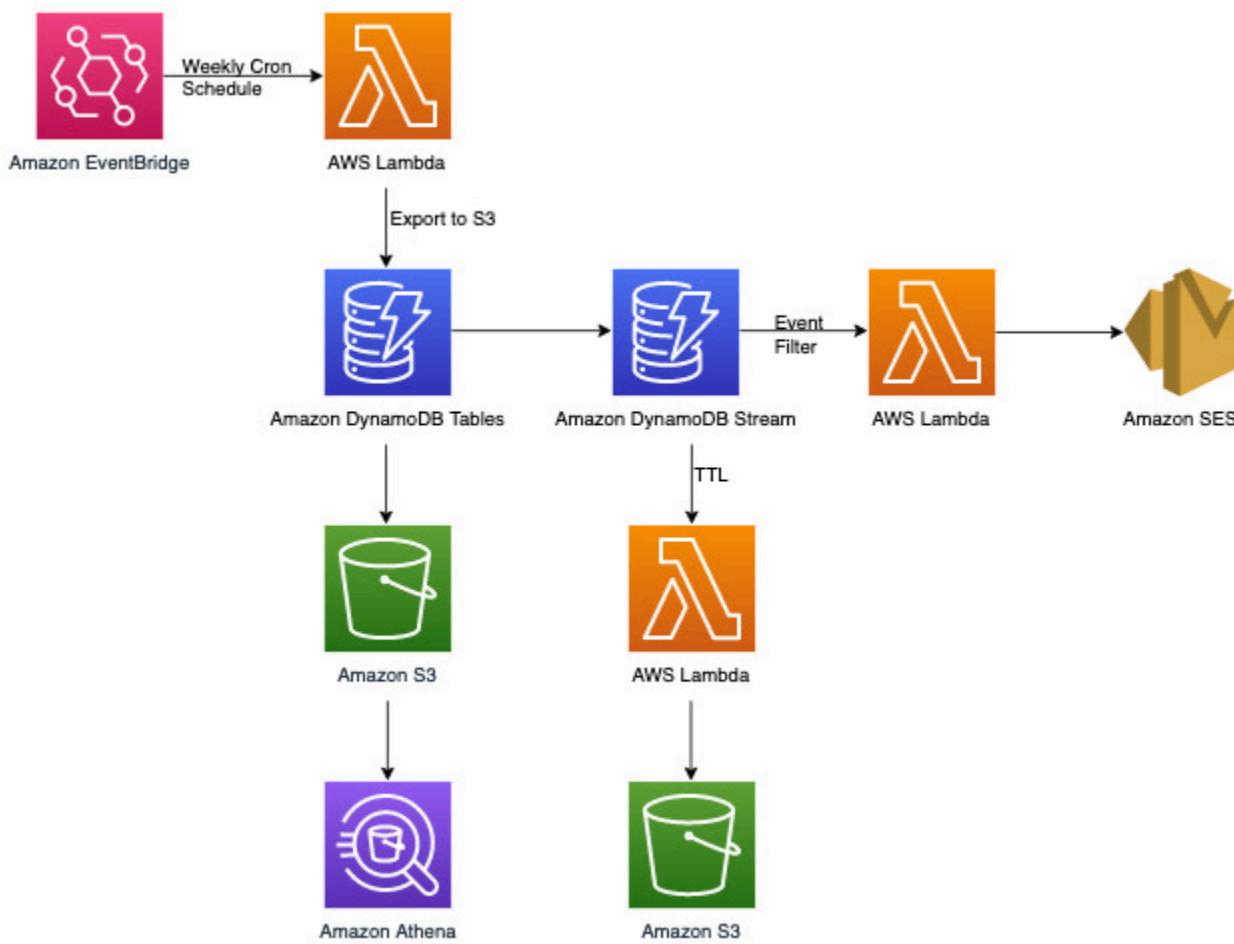
Complaint management system business use case

DynamoDB is a database well-suited for a complaint management system (or contact center) use case as most access patterns associated with them would be key-value based transactional lookups. The typical access patterns in this scenario would be to:

- Create and update complaints
- Escalate a complaint
- Create and read comments on a complaint
- Get all complaints by a customer
- Get all comments by an agent and get all escalations

Some comments may have attachments describing the complaint or solution. While these are all key-value access patterns, there can be additional requirements such as sending out notifications when a new comment is added to a complaint or running analytical queries to find complaint distribution by severity (or agent performance) per week. An additional requirement related to lifecycle management or compliance would be to archive complaint data after three years of logging the complaint.

Complaint management system architecture diagram



Apart from the key-value transactional access patterns that we will be handling in the DynamoDB data modeling section later, we have three non-transactional requirements. The architecture diagram above can be broken down into the following three workflows:

1. Send a notification when a new comment is added to a complaint
2. Run analytical queries on weekly data
3. Archive data older than three years

Let's take a more in-depth look at each one.

Send a notification when a new comment is added to a complaint

We can use the below workflow to achieve this requirement:



[DynamoDB Streams](#) is a change data capture mechanism to record all write activity on your DynamoDB tables. You can configure Lambda functions to trigger on some or all of these changes. An [event filter](#) can be configured on Lambda triggers to filter out events that are not relevant to the use-case. In this instance, we can use a filter to trigger Lambda only when a new comment is added and send out notification to relevant email ID(s) which can be fetched from [AWS Secrets Manager](#) or any other credential store.

Run analytical queries on weekly data

DynamoDB is suitable for workloads that are primarily focused on online transactional processing (OLTP). For the other 10-20% access patterns with analytical requirements, data can be exported to S3 with the managed [Export to Amazon S3](#) feature with no impact to the live traffic on DynamoDB table. Take a look at this workflow below:



[Amazon EventBridge](#) can be used to trigger AWS Lambda on schedule - it allows you to configure a cron expression for Lambda invocation to take place periodically. Lambda can invoke the ExportToS3 API call and store DynamoDB data in S3. This S3 data can then be accessed by a SQL engine such as [Amazon Athena](#) to run analytical queries on DynamoDB data without affecting the live transactional workload on the table. A sample Athena query to find number of complaints per severity level would look like this:

```
SELECT Item.severity.S as "Severity", COUNT(Item) as "Count"
FROM "complaint_management"."data"
WHERE NOT Item.severity.S = ''
GROUP BY Item.severity.S ;
```

This results in the following Athena query result:

Results (3)

Search rows

#	Severity	Count
1	P3	1
2	P2	2
3	P1	1

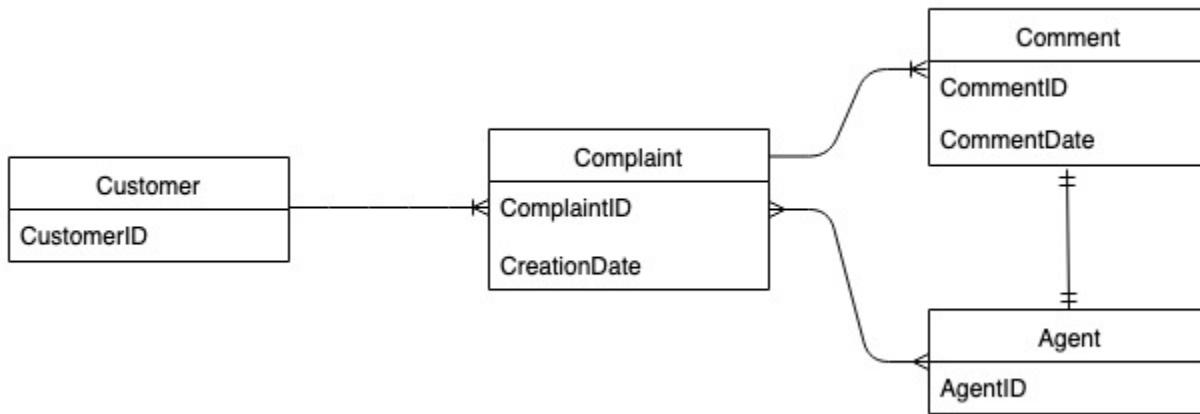
Archive data older than three years

You can leverage the DynamoDB [Time to Live \(TTL\)](#) feature to delete obsolete data from your DynamoDB table at no additional cost (except in the case of global tables replicas for the 2019.11.21 (Current) version, where TTL deletes replicated to other Regions consume write capacity). This data appears and can be consumed from DynamoDB Streams to be archived off into Amazon S3. The workflow for this requirement is as follows:



Complaint management system entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the complaint management system schema design.



Complaint management system access patterns

These are the access patterns we'll be considering for the complaint management schema design.

1. `createComplaint`
2. `updateComplaint`
3. `updateSeveritybyComplaintID`
4. `getComplaintByComplaintID`
5. `addCommentByComplaintID`
6. `getAllCommentsByComplaintID`
7. `getLatestCommentByComplaintID`
8. `getAComplaintbyCustomerIDAndComplaintID`
9. `getAllComplaintsByCustomerID`
10. `escalateComplaintByComplaintID`
11. `getAllEscalatedComplaints`
12. `getEscalatedComplaintsByAgentID` (order from newest to oldest)
13. `getCommentsByAgentID` (between two dates)

Complaint management system schema design evolution

Since this is a complaint management system, most access patterns revolve around a complaint as the primary entity. The `ComplaintID` being highly cardinal will ensure even distribution of data

in the underlying partitions and is also the most common search criteria for our identified access patterns. Therefore, ComplaintID is a good partition key candidate in this data set.

Step 1: Address access patterns 1 (`createComplaint`), 2 (`updateComplaint`), 3 (`updateSeveritybyComplaintID`), and 4 (`getComplaintByComplaintID`)

We can use a generic sort key valued called "metadata" (or "AA") to store complaint-specific information such as CustomerID, State, Severity, and CreationDate. We use singleton operations with PK=ComplaintID and SK="metadata" to do the following:

1. [PutItem](#) to create a new complaint
2. [UpdateItem](#) to update the severity or other fields in the complaint metadata
3. [GetItem](#) to fetch metadata for the complaint

Primary key		Attributes				
Partition key: PK	Sort key: SK	customer_id	current_state	creation_time	severity	complaint_description
Complaint1321	metadata	custXYZ	assigned	2023-05-10T15:58:00	P2	<Complaint Description>

Step 2: Address access pattern 5 (`addCommentByComplaintID`)

This access pattern requires a one-to-many relationship model between a complaint and comments on the complaint. We will use the [vertical partitioning](#) technique here to use a sort key and create an item collection with different types of data. If we look at access patterns 6 (`getAllCommentsByComplaintID`) and 7 (`getLatestCommentByComplaintID`), we know that comments will need to be sorted by time. We can also have multiple comments coming in at the same time so we can use the [composite sort key](#) technique to append time and CommentID in the sort key attribute.

Other options to deal with such possible comment collisions would be to increase the granularity for the timestamp or add an incremental number as a suffix instead of using Comment_ID. In this case, we'll prefix the sort key value for items corresponding to comments with "comm#" to enable range-based operations.

We also need to ensure that the currentState in the complaint metadata reflects the state when a new comment is added. Adding a comment might indicate that the complaint has been assigned to an agent or it has been resolved and so on. In order to bundle the addition of comment and

update of current state in the complaint metadata, in an all-or-nothing manner, we will use the TransactWriteItems API. The resulting table state now looks like this:

Primary key		Attributes				
Partition key: PK	Sort key: SK	comm_id	comm_date	complaint_state	comm_text	agentID
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	2023-05-10T16:00:00	investigating	<Comment text>	AgentB
		customer_id	current_state	creation_time	severity	complaint_description
	metadata	custXYZ	investigating	2023-05-10T15:58:00	P2	<Complaint Description>

Let's add some more data in the table and also add ComplaintID as a separate field from our PK for future-proofing the model in case we need additional indexes on ComplaintID. Also note that some comments may have attachments which we will store in Amazon Simple Storage Service and only maintain their references or URLs in DynamoDB. It's a best practice to keep the transactional database as lean as possible to optimize cost and performance. The data now looks like this:

Primary key		Attributes				
Partition key: PK	Sort key: SK	comm_id	comm_date	complaint_state	comm_text	agentID
Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA
		comm_id	comm_date	complaint_state	comm_text	attachments
	comm#2023-04-30T12:35:54#comm2	comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
		customer_id	complaint_id	current_state	creation_time	severity
	metadata	custABC	Complaint123	resolved	2023-04-30T12:00:00	P2
						<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB
		customer_id	complaint_id	current_state	creation_time	severity
	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2
						<descr_text>

Step 3: Address access patterns 6 (getAllCommentsByComplaintID) and 7 (getLatestCommentByComplaintID)

In order to get all comments for a complaint, we can use the query operation with the begins_with condition on the sort key. Instead of consuming additional read capacity to read

the metadata entry and then having the overhead of filtering the relevant results, having a sort key condition like this help us only read what we need. For example, a [query](#) operation with PK=Complaint123 and SK begins_with comm# would return the following while skipping the metadata entry:

Primary key		Attributes					
Partition key: PK	Sort key: SK						
Complaint123	comm#2023-04-30T12:00:24 #comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54 #comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
Complaint1321	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
	metadata	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	

Since we need the latest comment for a complaint in pattern 7

(getLatestCommentByComplaintID), let's use two additional query parameters:

1. ScanIndexForward should be set to False to get results sorted in a descending order
2. Limit should be set to 1 to get the latest (only one) comment

Similar to access pattern 6 (getAllCommentsByComplaintID), we skip the metadata entry using begins_with comm# as the sort key condition. Now, you can perform access pattern 7 on this design using the query operation with PK=Complaint123 and SK=begins_with comm#, ScanIndexForward=False, and Limit 1. The following targeted item will be returned as a result:

Partition key: PK	Sort key: SK	Attributes					
	comm#2023-04-30T12:00:24 #comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
Complaint123	comm#2023-04-30T12:35:54 #comm2	comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
		comm_id	comm_date	complaint_state	comm_text	attachments	agentID
	metadata	comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
		customer_id	complaint_id	current_state	creation_time	severity	complaint_description
Complaint1321	comm#2023-05-10T16:00:00 #comm3	custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
		comm_id	comm_date	complaint_state	comm_text	agentID	
	metadata	comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
		customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Let's add more dummy data to the table.

Primary key		Attributes					
Partition key: PK	Sort key: SK						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>
Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text		
		comm4	2022-12-31T19:32:00	waiting	<comm text>		
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
Complaint0987	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint0987	assigned	2023-06-10T12:30:08	P3	<description text>

Step 4: Address access patterns 8 (getAComplaintbyCustomerIDAndComplaintID) and 9 (getAllComplaintsByCustomerID)

Access patterns 8 (getAComplaintbyCustomerIDAndComplaintID) and 9 (getAllComplaintsByCustomerID) introduces new a search criteria: CustomerID. Fetching it from the existing table requires an expensive [Scan](#) to read all data and then filter relevant items for the CustomerID in question. We can make this search more efficient by creating

a [global secondary index \(GSI\)](#) with CustomerID as the partition key. Keeping in mind the one-to-many relationship between customer and complaints as well as access pattern 9 (getAllComplaintsByCustomerID), ComplaintID would be the right candidate for the sort key.

The data in the GSI would look like this:

Primary key		Attributes					
Partition key: customer_id	Sort key: complaint_id	PK	SK	current_state	creation_time	severity	complaint_description
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXYZ32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

An example query on this GSI for access pattern 8

(getAComplaintbyCustomerIDAndComplaintID) would be: `customer_id=custXYZ, sort key=Complaint1321`. The result would be:

Primary key		Attributes					
Partition key: customer_id	Sort key: complaint_id	PK	SK	current_state	creation_time	severity	complaint_description
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXYZ32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

To get all complaints for a customer for access pattern 9 (`getAllComplaintsByCustomerID`), the query on the GSI would be: `customer_id=custXYZ` as the partition key condition. The result would be:

Primary key		Attributes					
Partition key: <code>customer_id</code>	Sort key: <code>complaint_id</code>	PK	SK	current_state	creation_time	severity	complaint_description
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXYZ32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

Step 5: Address access pattern 10 (`escalateComplaintByComplaintID`)

This access introduces the escalation aspect. To escalate a complaint, we can use `UpdateItem` to add attributes such as `escalated_to` and `escalation_time` to the existing complaint metadata item. DynamoDB provides flexible schema design which means a set of non-key attributes can be uniform or discrete across different items. See below for an example:

`UpdateItem` with `PK=Complaint1444`, `SK=metadata`

Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text				
	comm4	2022-12-31T19:32:00	waiting	<comm text>					
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID		
	comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC			
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
	custXYZ32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07	

Step 6: Address access patterns 11 (`getAllEscalatedComplaints`) and 12 (`getEscalatedComplaintsByAgentID`)

Only a handful of complaints are expected to be escalated out of the whole data set. Therefore, creating an index on the escalation-related attributes would lead to efficient lookups as well as

cost-effective GSI storage. We can do this by leveraging the [sparse index](#) technique. The GSI with partition key as `escalated_to` and sort key as `escalation_time` would look like this:

Primary key		Attributes								
Partition key: <code>escalated_to</code>	Sort key: <code>escalation_time</code>	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	
AgentB	2023-01-03T04:00:07	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	
		Complaint1444	metadata	custXYZ2	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	
AgentB	2023-05-15T14:00:00	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	
		Complaint1321	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>	

To get all escalated complaints for access pattern 11 (`getAllEscalatedComplaints`), we simply scan this GSI. Note that this scan will be performant and cost-efficient due to the size of the GSI. To get escalated complaints for a specific agent (access pattern 12 (`getEscalatedComplaintsByAgentID`)), the partition key would be `escalated_to=agentID` and we set `ScanIndexForward` to `False` for ordering from newest to oldest.

Step 7: Address access pattern 13 (`getCommentsByAgentID`)

For the last access pattern, we need to perform a lookup by a new dimension: `AgentID`. We also need time-based ordering to read comments between two dates so we create a GSI with `agent_id` as the partition key and `comm_date` as the sort key. The data in this GSI will look like the following:

Primary key		Attributes						
Partition key: <code>agentID</code>	Sort key: <code>comm_date</code>	PK	SK	comm_id	complaint_state	comm_text		
AgentA	2023-04-30T12:00:24	PK	SK	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
		PK	SK	comm_id	complaint_state	comm_text	attachments	
AgentB	2023-05-10T16:00:00	PK	SK	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
		PK	SK	comm_id	complaint_state	comm_text	attachments	
AgentC	2022-12-31T19:40:00	PK	SK	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

An example query on this GSI would be `partition key agentID=AgentA and sort key=comm_date between (2023-04-30T12:30:00, 2023-05-01T09:00:00)`, the result of which is:

Primary key		Attributes					
Partition key: agentID	Sort key: comm_date	PK	SK	comm_id	complaint_state	comm_text	
AgentA	2023-04-30T12:00:24	Complaint1	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
		Complaint1	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/filters
createComplaint	Base table	PutItem	PK=complaint_id	SK=metadata	
updateComplaint	Base table	UpdateItem	PK=complaint_id	SK=metadata	
updateSeveritybyComplaintID	Base table	UpdateItem	PK=complaint_id	SK=metadata	
getComplaintByComplaintID	Base table	GetItem	PK=complaint_id	SK=metadata	
addCommentByComplaintID	Base table	TransactWriteItems	PK=complaint_id	SK=metadata, SK=comm#comm_date#comm_id	

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getAllCommentsByComplaintID	Base table	Query	PK=complaint_id	SK begins_with "comm#"	
getLatestCommentByComplaintID	Base table	Query	PK=complaint_id	SK begins_with "comm#"	scan_index_forward=False, Limit 1
getAComplaintbyCustomerIDAndComplaintID	Customer_complaint_GSI	Query	customer_id=customer_id	complaint_id = complaint_id	
getAllComplaintsByCustomerID	Customer_complaint_GSI	Query	customer_id=customer_id	N/A	
escalateComplaintByComplaintID	Base table	UpdateItem	PK=complaint_id	SK=metadata	
getAllEscalatedComplaints	Escalations_GSI	Scan	N/A	N/A	
getEscalatedComplaintsByAgentID (order from newest to oldest)	Escalations_GSI	Query	escalated_to=agent_id	N/A	scan_index_forward=False

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getCommentsByAgentID (between two dates)	Agents_Comments_GSI	Query	agent_id=agent_id	SK between (date1, date2)	

Complaint management system final schema

Here are the final schema designs. To download this schema design as a JSON file, see [DynamoDB Examples](#) on GitHub.

Base table

Primary key		Attributes						
Partition key: PK	Sort key: SK	comm_id	comm_date	complaint_state	comm_text	agentID		
Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA		
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA	
	comm#2023-04-30T12:35:54#comm2	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>	
	metadata	comm_id	comm_date	complaint_state	comm_text	agentID		
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB		
Complaint1321	comm#2023-05-10T16:00:00#comm3	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>	AgentB
	metadata	comm_id	comm_date	complaint_state	comm_text			2023-05-15T14:00:00
		comm4	2022-12-31T19:32:00	waiting	<comm text>			
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID	
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC	
Complaint1444	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to
		custXYZ2	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB
	metadata	customer_id	complaint_id	current_state	creation_time	severity		2023-01-03T04:00:07
		custXYZ	Complaint0987	assigned	2023-06-10T12:30:08	P3	<description text>	
Complaint0987	metadata							

Customer_Complaint_GSI

Primary key		Attributes							
Partition key: customer_id	Sort key: complaint_id	PK	SK	current_state	creation_time	severity	complaint_description		
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description		
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>		
custXYZ32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description		
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>		
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>	AgentB	2023-05-15T14:00:00

Escalations_GSI

Primary key		Attributes							
Partition key: escalated_to	Sort key: escalation_time	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
AgentB	2023-01-03T04:00:07	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	custXYZ32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
	2023-05-15T14:00:00	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Agents_Comments_GSI

Primary key		Attributes							
Partition key: agentID	Sort key: comm_date	PK	SK	comm_id	complaint_state	comm_text			
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text			
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>			
AgentB	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments		
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]		
AgentC	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text			
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>			
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments		
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]		

Using NoSQL Workbench with this schema design

You can import this final schema into [NoSQL Workbench](#), a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see [the section called "Download"](#).
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see [the section called "Importing an existing model"](#).
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see [the section called "Editing an existing model"](#).
5. To visualize your data model, add sample data, or import sample data from a CSV file, use the [Data Visualizer](#) feature of NoSQL Workbench.

Recurring payments schema design in DynamoDB

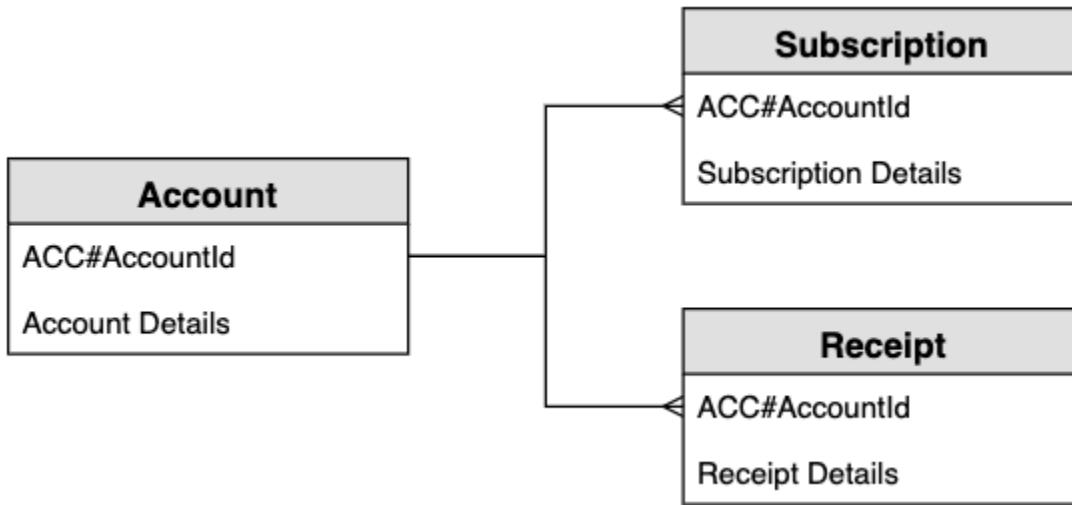
Recurring payments business use case

This use case talks about using DynamoDB to implement a recurring payments system. The data model has the following entities: *accounts*, *subscriptions*, and *receipts*. The specifics for our use case include the following:

- Each *account* can have multiple *subscriptions*
- The *subscription* has a `NextPaymentDate` when the next payment needs to be processed and a `NextReminderDate` when an email reminder is sent to the customer
- There is an item for the *subscription* that is stored and updated when the payment been processed (the average item size is around 1KB and the throughput depends on the number of *accounts* and *subscriptions*)
- The *payment processor* will also create a *receipt* as part of the process which is stored in the table and are set to expire after a period of time by using a [TTL](#) attribute

Recurring payments entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the recurring payments system schema design.



Recurring payments system access patterns

These are the access patterns we'll be considering for the recurring payments system schema design.

1. `createSubscription`
2. `createReceipt`
3. `updateSubscription`
4. `getDueRemindersByDate`
5. `getDuePaymentsByDate`
6. `getSubscriptionsByAccount`
7. `getReceiptsByAccount`

Recurring payments schema design

The generic names `PK` and `SK` are used for key attributes to allow storing different types of entities in the same table such as the account, subscription, and receipt entities. The user first creates a subscription, which is where the user agrees to pay an amount on the same day each month in return for a product. They get the choice on which day of the month to process the payment. There is also a reminder that will be sent prior to the payment being processed. The application works by having two batch jobs that run each day: one batch job sends reminders due that day and the other batch job processes any payments due that day.

Step 1: Address access pattern 1 (`createSubscription`)

Access pattern 1 (`createSubscription`) is used to initially create the subscription, and the details including SKU, NextPaymentDate, NextReminderDate and PaymentDetails are set. This step shows the state of the table for just one account with one subscription. There can be multiple subscriptions in the item collection so this is a one-to-many relationship.

Primary key		Attributes										
Partition key: PK	Sort key: SK	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate	
ACC#123	SUB#123#SKU#999 s@s.com	28	12.99	1970-01-01T00:00:00.000Z	2023-05-28	1970-01-01T00:00:00.000Z	2023-05-21	999	{"default-card": {"\$": "1234123412341234"}, "default-address": {"\$": "12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z		

Step 2: Address access patterns 2 (`createReceipt`) and 3 (`updateSubscription`)

Access pattern 2 (`createReceipt`) is used to create the receipt item. After the payment is processed each month, the payment processor will write a receipt back to the base table. There, could be multiple receipts in the item collection so this is a one-to-many relationship. The payment processor will also update the subscription item (access Pattern 3 (`updateSubscription`)) to update for the `NextReminderDate` or the `NextPaymentDate` for the next month.

Primary key		Attributes										
Partition key: PK	Sort key: SK	Email	SKU	ProcessedDate	ProcessedAmount	TTL						
ACC#123	REC#12023-05-28T14:15:39.24#SKU#999 s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200							
	SUB#123#SKU#999 s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card": {"\$": "1234123412341234"}, "default-address": {"\$": "12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z		

Step 3: Address access pattern 4 (`getDueRemindersByDate`)

The application processes reminders for the payment in batches for the current day. Therefore the application needs to access the subscriptions on a different dimension: date rather than account. This is a good use case for a [global secondary index \(GSI\)](#). In this step we add the index GSI-1, which uses the `NextReminderDate` as the GSI partition key. We do not need to replicate all the items. This GSI is a [sparse index](#) and the receipts items are not replicated. We also do not need to project all the attributes—we only need to include a subset of the attributes. The image below shows the schema of GSI-1 and it gives the information needed for the application to send the reminder email.

Primary key		Attributes				
Partition key: NextReminderDate	Sort key: LastReminderDate	SK	PK	SKU	Email	NextPaymentDate
2023-06-21	2023-05-21T14:15:39.247Z	SUB#123#SKU#999	ACC#123	999	s@s.com	2023-06-28

Step 4: Address access pattern 5 (`getDuePaymentsByDate`)

The application processes the payments in batches for the current day in the same way it does with reminders. We add GSI-2 in this step, and it uses the `NextPaymentDate` as the GSI partition key. We do not need to replicate all the items. This GSI is a sparse index as the receipts items are not replicated. The image below shows the schema of GSI-2.

Primary key		Attributes						
Partition key: <code>NextPaymentDate</code>	Sort key: <code>LastPaymentDate</code>	PK	SK	Email	PaymentDay	PaymentAmount	SKU	PaymentDetails
2023-06-28	2023-05-18T14:15:39.247Z	ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	999	{"default-card":{"S":"1234123412341234"}, "default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}

Step 5: Address access patterns 6 (`getSubscriptionsByAccount`) and 7 (`getReceiptsByAccount`)

The application can retrieve all the subscriptions for an account by using a [query](#) on the base table that targets the account identifier (the PK) and uses the range operator to get all the items where the SK begins with “SUB#”. The application can also use the same query structure to retrieve all the receipts by using a range operator to get all the items where the SK begins with “REC#”. This allows us to satisfy access patterns 6 (`getSubscriptionsByAccount`) and 7 (`getReceiptsByAccount`). The application uses these access patterns so the user can see their current subscriptions and their past receipts for the last six months. There is no change to the table schema in this step and we can see below how we target just the subscription item(s) in access pattern 6 (`getSubscriptionsByAccount`).

Primary key		Attributes								
Partition key: PK	Sort key: SK	Email	SKU	ProcessedDate	ProcessedAmount	TTL				
	REC#12023-05-28T14:15:39.24#SKU#999	s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200				
ACC#123	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"}, "default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}
										2023-05-18T09:41:25.856Z

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/GSI/ LSI	Operation	Partition key value	Sort key value
createSubscription	Base table	PutItem	ACC#account_id	SUB#<SUBID>#SKU<SKU>

Access pattern	Base table/GSI/ LSI	Operation	Partition key value	Sort key value
createReceipt	Base table	PutItem	ACC#account_id	REC#<ReceiptDate>#SKU<SKUID>
updateSubscription	Base table	UpdateItem	ACC#account_id	SUB#<SUBID>#SKU<SKUID>
getDueRemindersByDate	GSI-1	Query	<NextReminderDate>	
getDuePaymentsByDate	GSI-2	Query	<NextPaymentDate>	
getSubscriptionsByAccount	Base table	Query	ACC#account_id	SK begins_with "SUB#"
getReceiptsByAccount	Base table	Query	ACC#account_id	SK begins_with "REC#"

Recurring payments final schema

Here are the final schema designs. To download this schema design as a JSON file, see [DynamoDB Examples](#) on GitHub.

Base table

Primary key		Attributes										
Partition key: PK	Sort key: SK	Email	SKU	ProcessedDate	ProcessedAmount	TTL	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate	
ACC#123	REC#12023-05-28T14:15:39.24#SKU#999	s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200						
	SUB#123#SKU#999	s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card": {"\$": "1234123412341234"}, "default-address": {"\$": "12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z	

GSI-1

Primary key		Attributes					
Partition key: NextReminderDate	Sort key: LastReminderDate	SK	PK	SKU	Email	NextPaymentDate	
2023-06-21	2023-05-21T14:15:39.247Z	SUB#123#SKU#999	ACC#123	999	s@s.com	2023-06-28	

GSI-2

Primary key		Attributes						
Partition key: NextPaymentDate	Sort key: LastPaymentDate	PK	SK	Email	PaymentDay	PaymentAmount	SKU	PaymentDetails
2023-06-28	2023-05-18T14:15:39.247Z	ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	999	{"default-card":{"\$": "1234123412341234"}, "default-address":{"\$": "12 Bridge Street, Birmingham, B12 7ST"}}

Using NoSQL Workbench with this schema design

You can import this final schema into [NoSQL Workbench](#), a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see [the section called “Download”](#).
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see [the section called “Importing an existing model”](#).
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see [the section called “Editing an existing model”](#).
5. To visualize your data model, add sample data, or import sample data from a CSV file, use the [Data Visualizer](#) feature of NoSQL Workbench.

Monitoring device status updates in DynamoDB

This use case talks about using DynamoDB to monitor device status updates (or changes in device state) in DynamoDB.

Use case

In IoT use-cases (a smart factory for instance) many devices need to be monitored by operators and they periodically send their status or logs to a monitoring system. When there is a problem with a device, the status for the device changes from *normal* to *warning*. There are different log levels or statuses depending on the severity and type of abnormal behavior in the device. The system then

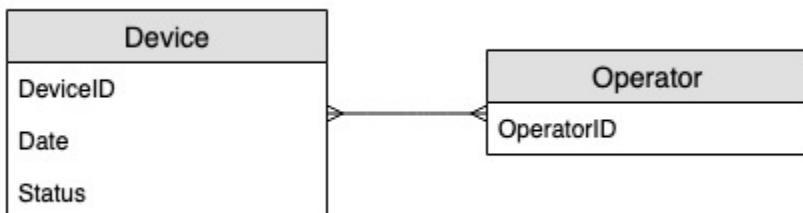
assigns an operator to check on the device and they may escalate the problem to their supervisor if needed.

Some typical access patterns for this system include:

- Create log entry for a device
- Get all logs for a specific device state showing the most recent logs first
- Get all logs for a given operator between two dates
- Get all escalated logs for a given supervisor
- Get all escalated logs with a specific device state for a given supervisor
- Get all escalated logs with a specific device state for a given supervisor for a specific date

Entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for monitoring device status updates.



Access patterns

These are the access patterns we'll be considering for monitoring device status updates.

1. `createLogEntryForSpecificDevice`
2. `getLogsForSpecificDevice`
3. `getWarningLogsForSpecificDevice`
4. `getLogsForOperatorBetweenTwoDates`
5. `getEscalatedLogsForSupervisor`
6. `getEscalatedLogsWithSpecificStatusForSupervisor`
7. `getEscalatedLogsWithSpecificStatusForSupervisorForDate`

Schema design evolution

Step 1: Address access patterns 1 (`createLogEntryForSpecificDevice`) and 2 (`getLogsForSpecificDevice`)

The unit of scaling for a device tracking system would be individual devices. In this system, a `deviceID` uniquely identifies a device. This makes `deviceID` a good candidate for the partition key. Each device sends information to the tracking system periodically (say, every five minutes or so). This ordering makes date a logical sorting criterion and therefore, the sort key. The sample data in this case would look something like this:

Primary key		Attributes
Partition key: DeviceID	Sort key: Date	
d#12345	2020-04-24T14:40:00	State WARNING1
	2020-04-24T14:45:00	State WARNING1
	2020-04-24T14:50:00	State WARNING1
	2020-04-24T14:55:00	State NORMAL
d#54321	2020-04-11T05:50:00	State WARNING3
	2020-04-11T05:55:00	State WARNING3
	2020-04-11T06:00:00	State NORMAL
	2020-04-11T09:25:00	State WARNING2
d#11223	2020-04-11T09:30:00	State NORMAL
	2020-04-27T16:10:00	State WARNING4
	2020-04-27T16:15:00	State WARNING4

To fetch log entries for a specific device, we can perform a [query](#) operation with partition key DeviceID="d#12345".

Step 2: Address access pattern 3 (getWarningLogsForSpecificDevice)

Since State is a non-key attribute, addressing access pattern 3 with the current schema would require a [filter expression](#). In DynamoDB, filter expressions are applied after data is read using key condition expressions. For example, if we were to fetch warning logs for d#12345, the query operation with partition key DeviceID="d#12345" will read four items from the above table and then filter out the one item with the *warning* status. This approach is not efficient at scale. A filter expression can be a good way to exclude items that are queried if the ratio of excluded items is low or the query is performed infrequently. However, for cases where many items are retrieved from a table and the majority of the items are filtered out, we can continue evolving our table design so it runs more efficiently.

Let's change how to handle this access pattern by using [composite sort keys](#). You can import sample data from [DeviceStateLog_3.json](#) where the sort key is changed to State#Date. This sort key is the composition of the attributes State, #, and Date. In this example, # is used as a delimiter. The data now looks something like this:

Primary key	
Partition key: DeviceID	Sort key: State#Date
d#12345	NORMAL#2020-04-24T14:55:00
	WARNING1#2020-04-24T14:40:00
	WARNING1#2020-04-24T14:45:00
	WARNING1#2020-04-24T14:50:00

To fetch only warning logs for a device, the query becomes more targeted with this schema. The key condition for the query uses partition key DeviceID="d#12345" and sort key State#Date

`begins_with "WARNING"`. This query will only read the relevant three items with the *warning* state.

Step 3: Address access pattern 4 (`getLogsForOperatorBetweenTwoDates`)

You can import [DeviceStateLog_4.json](#) where the `Operator` attribute was added to the `DeviceStateLog` table with example data.

Primary key		Attributes		
Partition key: DeviceID	Sort key: State#Date	Operator	Date	State
d#12345	NORMAL#2020-04-24T14:55:00	Liz	2020-04-24T14:55:00	NORMAL
		Liz	2020-04-24T14:40:00	WARNING1
	WARNING1#2020-04-24T14:45:00	Liz	2020-04-24T14:45:00	WARNING1
		Liz	2020-04-24T14:50:00	WARNING1
	NORMAL#2020-04-11T06:00:00	Liz	2020-04-11T06:00:00	NORMAL
		Sue	2020-04-11T09:30:00	NORMAL
	WARNING2#2020-04-11T09:25:00	Sue	2020-04-11T09:25:00	WARNING2
		Sue	2020-04-11T05:50:00	WARNING3
d#54321	WARNING3#2020-04-11T05:55:00	Liz	2020-04-11T05:55:00	WARNING3
		Sue	2020-04-27T16:10:00	WARNING4
	WARNING4#2020-04-27T16:15:00	Sue	2020-04-27T16:15:00	WARNING4
		Sue	2020-04-27T16:15:00	WARNING4
d#11223	Since Operator is not currently a partition key, there is no way to perform a direct key-value lookup on this table based on Operator ID. We'll need to create a new item collection with a			

global secondary index on OperatorID. The access pattern requires a lookup based on dates so Date is the sort key attribute for the [global secondary index \(GSI\)](#). This is what the GSI now looks like:

Primary key		Attributes		
Partition key: Operator	Sort key: Date			
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

For access pattern 4 (getLogsForOperatorBetweenTwoDates), you can query this GSI with partition key OperatorID=Liz and sort key Date between 2020-04-11T05:58:00 and 2020-04-24T14:50:00.

Primary key		Attributes		
Partition key: Operator	Sort key: Date	DeviceID	State#Date	State
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
Sue	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

Step 4: Address access patterns 5 (`getEscalatedLogsForSupervisor`) 6 (`getEscalatedLogsWithSpecificStatusForSupervisor`), and 7 (`getEscalatedLogsWithSpecificStatusForSupervisorForDate`)

We'll be using a [sparse index](#) to address these access patterns.

Global secondary indexes are sparse by default, so only items in the base table that contain primary key attributes of the index will actually appear in the index. This is another way of excluding items that are not relevant for the access pattern being modeled.

You can import [DeviceStateLog_6.json](#) where the `EscalatedTo` attribute was added to the `DeviceStateLog` table with example data. As mentioned earlier, not all of the logs gets escalated to a supervisor.

Primary key		Attributes		
Partition key: DeviceID	Sort key: State#Date	Operator	Date	State
d#12345	NORMAL#2020-04-24T14:55:00	Liz	2020-04-24T14:55:00	NORMAL
	WARNING1#2020-04-24T14:40:00	Liz	2020-04-24T14:40:00	WARNING1
	WARNING1#2020-04-24T14:45:00	Liz	2020-04-24T14:45:00	WARNING1
	WARNING1#2020-04-24T14:50:00	Liz	2020-04-24T14:50:00	WARNING1
	NORMAL#2020-04-11T06:00:00	Liz	2020-04-11T06:00:00	NORMAL
	NORMAL#2020-04-11T09:30:00	Sue	2020-04-11T09:30:00	NORMAL
	WARNING2#2020-04-11T09:25:00	Sue	2020-04-11T09:25:00	WARNING2
	WARNING3#2020-04-11T05:50:00	Sue	2020-04-11T05:50:00	WARNING3
d#54321	WARNING3#2020-04-11T05:55:00	Liz	2020-04-11T05:55:00	WARNING3
	WARNING4#2020-04-27T16:10:00	Sue	2020-04-27T16:10:00	WARNING4
	WARNING4#2020-04-27T16:15:00	Sue	2020-04-27T16:15:00	WARNING4
				Sara
d#11223	EscalatedTo			

You can now create a new GSI where EscalatedTo is the partition key and State#Date is the sort key. Notice that only items that have both EscalatedTo and State#Date attributes appear in the index.

Primary key		Attributes			
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator	Date	State
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue	2020-04-27T16:15:00	WARNING4

The rest of the access patterns are summarized as follows:

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
createLogEntryForSpecificDevice	Base table	PutItem	DeviceID=deviceld	State#Date=e=state#date	
getLogsForSpecificDevice	Base table	Query	DeviceID=deviceld	State#Date begins_with "state1#"	ScanIndexForward = False
getWarningLogsForSpecificDevice	Base table	Query	DeviceID=deviceld	State#Date begins_with "WARNING"	
getLogsForOperatorBetweenTwoDates	GSI-1	Query	Operator=operatorName	Date between date1 and date2	
getEscalatedLogsForSupervisor	GSI-2	Query	EscalatedTo=supervisorName		

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getEscalatedLogsWithSpecificStatusForSupervisor	GSI-2	Query	Escalated To=supervisorName	State#Date begins_with "state1#"	
getEscalatedLogsWithSpecificStatusForSupervisorForDate	GSI-2	Query	Escalated To=supervisorName	State#Date begins_with "state1#date1"	

Final schema

Here are the final schema designs. To download this schema design as a JSON file, see [DynamoDB Examples](#) on GitHub.

Base table

Primary key		Attributes		
Partition key: DeviceID	Sort key: State#Date	Operator	Date	State
d#12345	NORMAL#2020-04-24T14:55:00	Liz	2020-04-24T14:55:00	NORMAL
	WARNING1#2020-04-24T14:40:00	Liz	2020-04-24T14:40:00	WARNING1
	WARNING1#2020-04-24T14:45:00	Liz	2020-04-24T14:45:00	WARNING1
	WARNING1#2020-04-24T14:50:00	Liz	2020-04-24T14:50:00	WARNING1
	NORMAL#2020-04-11T06:00:00	Liz	2020-04-11T06:00:00	NORMAL
	NORMAL#2020-04-11T09:30:00	Sue	2020-04-11T09:30:00	NORMAL
	WARNING2#2020-04-11T09:25:00	Sue	2020-04-11T09:25:00	WARNING2
	WARNING3#2020-04-11T05:50:00	Sue	2020-04-11T05:50:00	WARNING3
d#54321	WARNING3#2020-04-11T05:55:00	Liz	2020-04-11T05:55:00	WARNING3
	WARNING4#2020-04-27T16:10:00	Sue	2020-04-27T16:10:00	WARNING4
	WARNING4#2020-04-27T16:15:00	Sue	2020-04-27T16:15:00	WARNING4
				EscalatedTo
d#11223				Sara

GSI-1

Primary key		Attributes		
Partition key: Operator	Sort key: Date	DeviceID	State#Date	State
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

GSI-2

Primary key		Attributes			
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator	Date	State
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue	2020-04-27T16:15:00	WARNING4

Using NoSQL Workbench with this schema design

You can import this final schema into [NoSQL Workbench](#), a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see [the section called “Download”](#).
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see [the section called “Importing an existing model”](#).
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see [the section called “Editing an existing model”](#).
5. To visualize your data model, add sample data, or import sample data from a CSV file, use the [Data Visualizer](#) feature of NoSQL Workbench.

Using DynamoDB as a data store for an online shop

This use case talks about using DynamoDB as a data store for an online shop (or e-store).

Use case

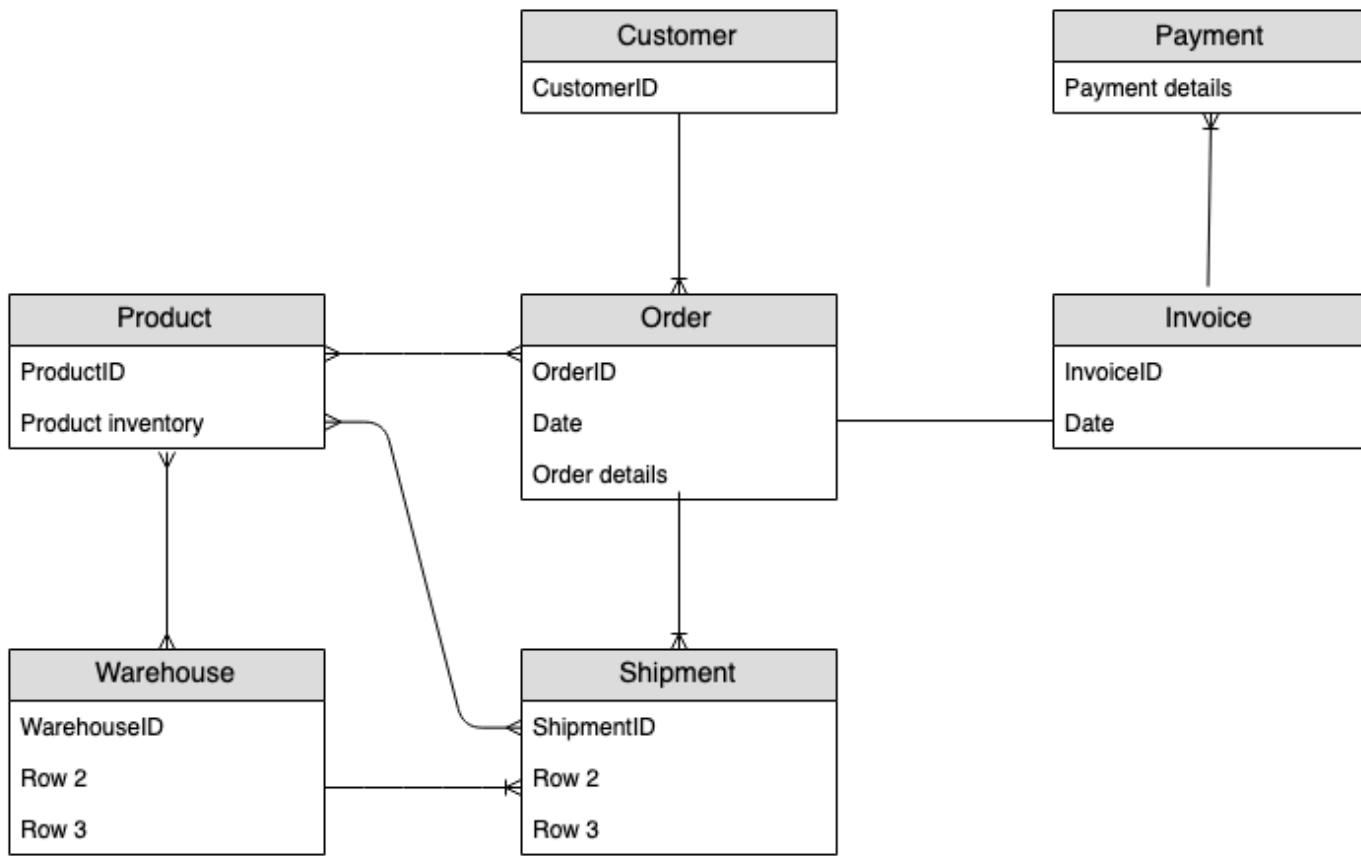
An online store lets users browse through different products and eventually purchase them. Based on the generated invoice, a customer can pay using a discount code or gift card and then pay the remaining amount with a credit card. Purchased products will be picked from one of several warehouses and will be shipped to the provided address. Typical access patterns for an online store include:

- Get customer for a given customerId

- Get product for a given productId
- Get warehouse for a given warehouseId
- Get a product inventory for all warehouses by a productId
- Get order for a given orderId
- Get all products for a given orderId
- Get invoice for a given orderId
- Get all shipments for a given orderId
- Get all orders for a given productId for a given date range
- Get invoice for a given invoiceId
- Get all payments for a given invoiceId
- Get shipment details for a given shipmentId
- Get all shipments for a given warehouseId
- Get inventory of all products for a given warehouseId
- Get all invoices for a given customerId for a given date range
- Get all products ordered by a given customerId for a given date range

Entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using to model DynamoDB as a data store for an online shop.



Access patterns

These are the access patterns we'll be considering when using DynamoDB as a data store for an online shop.

1. `getCustomerByCustomerId`
2. `getProductByProductId`
3. `getWarehouseByWarehouseId`
4. `getProductInventoryByProductId`
5. `getOrderDetailsByOrderId`
6. `getProductByOrderId`
7. `getInvoiceByOrderId`
8. `getShipmentByOrderId`
9. `getOrderByProductIdForDateRange`
10. `getInvoiceByInvoiceId`

```
11getPaymentByInvoiceId  
12getShipmentDetailsByShipmentId  
13getShipmentByWarehouseId  
14getProductInventoryByWarehouseId  
15getInvoiceByCustomerIdForDateRange  
16getProductsByCustomerIdForDateRange
```

Schema design evolution

Using [NoSQL Workbench for DynamoDB](#), import [AnOnlineShop_1.json](#) to create a new data model called AnOnlineShop and a new table called OnlineShop. Note that we use the generic names PK and SK for the partition key and sort key. This is a practice used in order to store different types of entities in the same table.

Step 1: Address access pattern 1 (getCustomerById)

Import [AnOnlineShop_2.json](#) to handle access pattern 1 (getCustomerById). Some entities do not have relationships to other entities, so we will use the same value of PK and SK for them. In the example data, note that the keys use a prefix c# in order to distinguish the customerId from other entities that will be added later. This practice is repeated for other entities as well.

To address this access pattern, a [GetItem](#) operation can be used with PK=customerId and SK=customerId.

Step 2: Address access pattern 2 (getProductById)

Import [AnOnlineShop_3.json](#) to address access pattern 2 (getProductById) for the product entity. The product entities are prefixed by p# and the same sort key attribute has been used to store customerID as well as productId. Generic naming and [vertical partitioning](#) allows us to create such item collections for an effective single table design.

To address this access pattern, a GetItem operation can be used with PK=productId and SK=productId.

Step 3: Address access pattern 3 (getWarehouseById)

Import [AnOnlineShop_4.json](#) to address access pattern 3 (getWarehouseById) for the warehouse entity. We currently have the customer, product, and warehouse entities added

to the same table. They are distinguished using prefixes and the `EntityType` attribute. A type attribute (or prefix naming) improves the model's readability. The readability would be affected if we simply stored alphanumeric IDs for different entities in the same attribute. It would be difficult to tell one entity from the other in the absence of these identifiers.

To address this access pattern, a `GetItem` operation can be used with `PK=warehouseId` and `SK=warehouseId`.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
c#12345	c#12345	EntityType	Email	Name
		customer	samaneh@example.com	Samaneh Utter
p#12345	p#12345	EntityType	Detail	Price
		product	{"Name": {"S": "Options Open"}, "Description": {"S": "The latest album"}}	100
w#12345	w#12345	EntityType	Address	
		warehouse	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "MainStreet"}, "Number": {"S": "20"}, "ZipCode": {"S": "41111"}}	

Step 4: Address access pattern 4 (`getProductInventoryByProductId`)

Import [AnOnlineShop_5.json](#) to address access pattern 4 (`getProductInventoryByProductId`). `warehouseItem` entity is used to keep track of the number of products in each warehouse. This item would normally be updated when a product is added or removed from a warehouse. As seen in the ERD, there is a many-to-many relationship between `product` and `warehouse`. Here, the one-to-many relationship from `product` to `warehouse` is modeled as `warehouseItem`. Later on, the one-to-many relationship from `warehouse` to `product` will be modeled as well.

Access pattern 4 can be addressed with a query on `PK=ProductId` and `SK begins_with "w#"`.

For more information about `begins_with()` and other expressions that can be applied to sort keys, see [Key Condition Expressions](#).

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
c#12345	c#12345	EntityType	Email	Name
		customer	samaneh@example.com	Samaneh Utter
p#12345	p#12345	EntityType	Detail	Price
		product	{"Name": {"S": "Options Open"}, "Description": {"S": "The latest album"}}	100
p#12345	w#12345	EntityType	Quantity	
		warehousesItem	50	
w#12345	w#12345	EntityType	Address	
		warehouse	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "MainStreet"}, "Number": {"S": "20"}, "ZipCode": {"S": "41111"}}	

Step 5: Address access patterns 5 (`getOrderDetailsByOrderId`) and 6 (`getProductByOrderId`)

Add some more customer, product, and warehouse items to the table by importing [AnOnlineShop_6.json](#). Then, import [AnOnlineShop_7.json](#) to build an item collection for order that can address access patterns 5 (`getOrderDetailsByOrderId`) and 6 (`getProductByOrderId`). You can see the one-to-many relationship between order and product modeled as orderItem entities.

To address access pattern 5 (`getOrderDetailsByOrderId`), query the table with `PK=orderId`. This will provide all information about the order including `customerId` and ordered products.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

To address access pattern 6 (getProductByOrderId), we need to read products in an order only. Query the table with PK=orderId and SK begins_with “p#” to accomplish this.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

Step 6: Address access pattern 7 (getInvoiceByOrderId)

Import [AnOnlineShop_8.json](#) to add an invoice entity to the order item collection to handle access pattern 7 (getInvoiceByOrderId). To address this access pattern, you can use a query operation with PK=orderId and SK begins_with “i#”.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	i#55443	EntityType	Amount	Date
		invoice	400	2020-06-21T19:18:00
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

Step 7: Address access pattern 8 (getShipmentByOrderId)

Import [AnOnlineShop_9.json](#) to add shipment entities to the `order` item collection to address access pattern 8 (getShipmentByOrderId). We are extending the same vertically partitioned model by adding more types of entities in the single table design. Notice how the `order` item collection contains the different relationships that an `order` entity has with the `shipment`, `orderItem`, and `invoice` entities.

To get shipments by `orderId`, you can perform a query operation with `PK=orderId` and `SK begins_with "sh#"`.

Base table:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
o#12345	c#12345	EntityType	Date		
		order	2020-06-21T19:10:00		
o#12345	i#55443	EntityType	Amount	Date	
		invoice	400	2020-06-21T19:18:00	
o#12345	p#12345	EntityType	Price	Quantity	
		orderItem	100	2	
o#12345	p#99887	EntityType	Price	Quantity	
		orderItem	40	5	
o#12345	sh#88899	EntityType	Address	Type	Date
		shipment	{"Country":{"S":"Sweden"}, "County":{"S":"Vastra Gotaland"}, "City":{"S":"Goteborg"}, "Street":{"S":"Slanbarsvagen"}, "Number":{"S":"34"}, "ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00
	sh#98765	EntityType	Address	Type	Date
		shipment	{"Country":{"S":"Sweden"}, "County":{"S":"Vastra Gotaland"}, "City":{"S":"Goteborg"}, "Street":{"S":"Slanbarsvagen"}, "Number":{"S":"34"}, "ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00

Step 8: Address access pattern 9 (getOrderByIdForDateRange)

We created an *order* item collection in the previous step. This access pattern has new lookup dimensions (ProductID and Date) which requires you to scan the whole table and filter out relevant records to fetch targeted items. In order to address this access pattern, we'll need to create a [global secondary index \(GSI\)](#). Import [AnOnlineShop_10.json](#) to create a new item collection using the GSI that makes it possible to retrieve *orderItem* data from several *order* item collections. The data now has GSI1-PK and GSI1-SK which will be GSI1's partition key and sort key, respectively.

DynamoDB automatically populates items which contain a GSI's key attributes from the table to the GSI. There is no need to manually do any additional inserts into the GSI.

To address access pattern 9, perform a query on GSI1 with GSI1-PK=productId and GSI1-SK between (date1, date2).

Base table:

Primary key		Attributes				
Partition key: PK	Sort key: SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
o#12345	p#12345	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
		orderItem	p#12345	2020-06-21T19:18:00	100	2
	p#99887	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
		orderItem	p#99887	2020-06-21T19:20:00	40	5

GSI1:

Primary key		Attributes				
Partition key: GSI1-PK	Sort key: GSI1-SK	PK	SK	EntityType	Quantity	Price
p#12345	2020-06-21T19:18:00	PK	SK	EntityType	Quantity	Price
		o#12345	p#12345	orderItem	2	100
p#99887	2020-06-21T19:20:00	PK	SK	EntityType	Quantity	Price
		o#12345	p#99887	orderItem	5	40

Step 9: Address access patterns 10 (`getInvoiceByInvoiceId`) and 11 (`getPaymentByInvoiceId`)

Import [AnOnlineShop_11.json](#) to address access patterns 10 (`getInvoiceByInvoiceId`) and 11 (`getPaymentByInvoiceId`), both of which are related to `invoice`. Even though these are two different access patterns, they are realized using the same key condition. Payments are defined as an attribute with the map data type on the `invoice` entity.

Note

GSI1-PK and GSI1-SK is overloaded to store information about different entities so that multiple access patterns can be served from the same GSI. For more information about GSI overloading, see [Overloading Global Secondary Indexes](#).

To address access pattern 10 and 11, query GSI1 with GSI1-PK=invoiceId and GSI1-SK=invoiceId.

GSI1:

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK	PK	SK	EntityType	GSI2-PK	GSI2-SK	Detail	Amount	Date
i#55443	i#55443	o#12345	i#55443	invoice	c#12345	i#2020-06-21T19:18:00	{"Payments": [{"L": [{"M": {"Type": "S": "GiftCard"}, "Amount": "N": "100"}, {"Data": {"S": "GiftCard data here..."}}, {"M": {"Type": "S": "MasterCard"}, "Amount": "N": "300"}, {"Data": {"S": "Payment data here..."}]}]}	400	2020-06-21T19:18:00

Step 10: Address access patterns 12 (getShipmentDetailsByShipmentId) and 13 (getShipmentByWarehouseId)

Import [AnOnlineShop_12.json](#) to address access patterns 12 (getShipmentDetailsByShipmentId) and 13 (getShipmentByWarehouseId).

Notice that shipmentItem entities are added to the *order* item collection on the base table in order to be able to retrieve all details about an order in a single query operation.

Base table:

Primary key		Attributes							
Partition key: PK	Sort key: SK	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
o#12345	sh#88899	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
		shipment	sh#88899	sh#88899	w#12376	sh#88899	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "Slanbarsvagen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T08:20:00
	sh#98765	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
		shipment	sh#98765	sh#98765	w#12345	sh#98765	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "Slanbarsvagen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T10:20:00
	shp#12345	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#98765	p#99887	3				
	shp#54321	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#88899	p#99887	2				
	shp#55555	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#98765	p#12345	2				

The GSI1 partition and sort keys have already been used to model a one-to-many relationship between `shipment` and `shipmentItem`. To address access pattern 12 (`getShipmentDetailsByShipmentId`), query GSI1 with `GSI1-PK=shipmentId` and `GSI1-SK=shipmentId`.

GSI1:

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK	PK	SK	EntityType	Quantity				
sh#88899	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#54321	shipmentItem	2				
sh#98765	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#88899	shipment	w#12376	sh#88899	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "Slanbarsvagen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T08:20:00
sh#98765	p#12345	PK	SK	EntityType	Quantity				
		o#12345	shp#55555	shipmentItem	2				
sh#98765	sh#98765	PK	SK	EntityType	Quantity				
		o#12345	shp#12345	shipmentItem	3				
sh#98765	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12345	sh#98765	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "Slanbarsvagen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T10:20:00

We'll need to create another GSI (GSI2) to model the new one-to-many relationship between warehouse and shipment for access pattern 13 (getShipmentByWarehouseId). To address this access pattern, query GSI2 with GSI2-PK=warehouseId and GSI2-SK begins_with "sh#".

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
w#12376	sh#88899	o#12345	sh#88899	shipment	sh#88899	sh#88899	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "Slanbarsvagen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}}	Express	2020-06-22T08:20:00
w#12345	sh#98765	o#12345	sh#98765	shipment	sh#98765	sh#98765	{"Country": {"S": "Sweden"}, "County": {"S": "Vastra Gotaland"}, "City": {"S": "Goteborg"}, "Street": {"S": "Slanbarsvagen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}}	Express	2020-06-22T10:20:00

Step 11: Address access patterns 14 (`getProductInventoryByWarehouseId`) 15 (`getInvoiceByCustomerIdForDateRange`), and 16 (`getProductsByCustomerIdForDateRange`)

Import [AnOnlineShop_13.json](#) to add data related to the next set of access patterns. To address access pattern 14 (`getProductInventoryByWarehouseId`), query GSI2 with GSI2-PK=warehouseId and GSI2-SK begins_with “p#”.

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK	PK	SK	EntityType	Quantity				
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouselt em	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouselt em	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments": [{"L": [{"M": {"Type": {"S": "GiftCard"}, "Amoun": {"N": "100"}, "Data": {"S": "GiftCard data here..."}}, {"M": {"Type": {"S": "MasterCard"}, "Amout": {"N": "300"}, "Data": {"S": "Payment data here..."}}}]}], "M": {"Type": {"S": "MasterCard"}, "Amout": {"N": "300"}, "Data": {"S": "Payment data here..."}}}}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

To address access pattern 15 (getInvoiceByCustomerIdForDateRange), query GSI2 with GSI2-PK=customerId and GSI2-SK between (i#date1, i#date2).

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK	PK	SK	EntityType	Quantity				
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehousetem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehousetem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments": [{"L": [{"M": {"Type": {"S": "GiftCard"}, "Amoun": {"N": "100"}, "Data": {"S": "GiftCard data here..."}}, {"M": {"Type": {"S": "MasterCard"}, "Amoun": {"N": "300"}, "Data": {"S": "Payment data here..."}}}]}], "M": {"Type": {"S": "MasterCard"}, "Amoun": {"N": "300"}, "Data": {"S": "Payment data here..."}}}}}	400	2020-06-21T19:18:00
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

To address access pattern 16 (getProductsByCustomerIdForDateRange), query GSI2 with GSI2-PK=customerId and GSI2-SK between (p#date1, p#date2).

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK	PK	SK	EntityType	Quantity				
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehousetem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehousetem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments": [{"L": [{"M": {"Type": {"S": "GiftCard"}, "Amount": {"N": "100"}, "Data": {"S": "GiftCard data here..."}}, {"M": {"Type": {"S": "MasterCard"}, "Amount": {"N": "300"}, "Data": {"S": "Payment data here..."}}}]}], "M": {"Type": {"S": "MasterCard"}, "Amount": {"N": "300"}, "Data": {"S": "Payment data here..."}}}	400	2020-06-21T19:18:00
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

Note

In [NoSQL Workbench](#), *facets* represent an application's different data access patterns for DynamoDB. Facets give you a way to view a subset of the data in a table, without having to see records that don't meet the constraints of the facet. Facets are considered a visual data modeling tool, and don't exist as a usable construct in DynamoDB as they are purely an aid for modeling access patterns.

Import [AnOnlineShop_facets.json](#) to see the facets for this use case.

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/GSI/ LSI	Operation	Partition key value	Sort key value
getCustomerByCustomerId	Base table	GetItem	PK=customerId	SK=customerId
getProductByProductId	Base table	GetItem	PK=productId	SK=productId
getWarehouseByWarehouseId	Base table	GetItem	PK=warehouseId	SK=warehouseId
getProductInventoryByProductId	Base table	Query	PK=productId	SK begins_with "w#"
getOrderDetailsByOrderId	Base table	Query	PK=orderId	
getProductByOrderId	Base table	Query	PK=orderId	SK begins_with "p#"
getInvoiceByOrderId	Base table	Query	PK=orderId	SK begins_with "i#"
getShipmentByOrderId	Base table	Query	PK=orderId	SK begins_with "sh#"
getOrderByProductIdForDateRange	GSI1	Query	PK=productId	SK between date1 and date2
getInvoiceByInvoiceId	GSI1	Query	PK=invoiceld	SK=invoiceld

Access pattern	Base table/GSI/ LSI	Operation	Partition key value	Sort key value
getPaymentByInvoiceId	GSI1	Query	PK=invoiceId	SK=invoiceId
getShipmentDetailsByShipmentId	GSI1	Query	PK=shipmentId	SK=shipmentId
getShipmentByWarehouseId	GSI2	Query	PK=warehouseId	SK begins_with "sh#"
getProductInventoryByWarehouseId	GSI2	Query	PK=warehouseId	SK begins_with "p#"
getInvoiceByCustomerIdForDateRange	GSI2	Query	PK=customerId	SK between i#date1 and i#date2
getProductsByCustomerIdForDateRange	GSI2	Query	PK=customerId	SK between p#date1 and p#date2

Online shop final schema

Here are the final schema designs. To download this schema design as a JSON file, see [DynamoDB Design Patterns](#) on GitHub.

Base table

Primary key		Attributes			
Partition key: PK	Sort key: SK				
c#12345	c#12345	EntityType	Email	Name	
		customer	samaneh@example.com	Samaneh	
c#23456	c#23456	EntityType	Email	Name	
		customer	kathleen@example.com	Kathleen	
c#54321	c#54321	EntityType	Email	Name	
		customer	henrik@example.com	Henrik	
p#12345	p#12345	EntityType	Detail	Price	
		product	{"Name": {"S": "Options Open"}, "Description": {"S": "The latest album"}}	100	
	w#12345	EntityType	GSI2-PK	GSI2-SK	Quantity
		warehousesItem	w#12345	p#12345	50
	p#99887	EntityType	Detail	Price	
		product	{"Name": {"S": "The Book"}, "Description": {"S": "The best book ever"}}	40	
		EntityType	GSI2-PK	GSI2-SK	Quantity
p#99887	w#12345	warehousesItem	w#12345	p#99887	4
		EntityType	Quantity		
	w#12376	warehousesItem	4		
w#12345	w#12345	EntityType	Address		
		warehouse	{"Country": {"S": "Sweden"}, "County": {"S": "Västra Götaland"}, "City": {"S": "Göteborg"}, "Street": {"S": "Main Street"}, "Number": {"S": "20"}, "ZipCode": {"S": "41111"}}		
Online shop		EntityType	Address		
			{"Country": {"S": "Sweden"}, "County": {"S": "Västra Götaland"}, "City": {"S": "Göteborg"}, "Street": {"S": "Main Street"}, "Number": {"S": "20"}, "ZipCode": {"S": "41111"}}		

GSI1

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK								
p#12345	2020-06-21T19:18:00	PK	SK	EntityType	GSI2-PK	GSI2-SK	Price	Quantity	
		o#12345	p#12345	orderItem	c#12345	2020-06-21T19:18:00	100	2	
p#99887	2020-06-21T19:20:00	PK	SK	EntityType	GSI2-PK	GSI2-SK	Price	Quantity	
		o#12345	p#99887	orderItem	c#12345	2020-06-21T19:20:00	40	5	
i#55443	i#55443	PK	SK	EntityType	GSI2-PK	GSI2-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	c#12345	2020-06-21T19:18:00	{"Payments": [{"L": [{"M": {"Type": {"S": "GiftCard"}, "Amount": {"N": "100"}, "Data": {"S": "GiftCard data here..."}}, {"M": {"Type": {"S": "MasterCard"}, "Amount": {"N": "300"}, "Data": {"S": "Payment data here..."}}}]}]}	400	2020-06-21T19:18:00
sh#88899	sh#88899	PK	SK	EntityType	Quantity				
		o#12345	shp#54321	shipmentItem	2				
sh#98765	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12376	sh#88899	{"Country": {"S": "Sweden"}, "County": {"S": "Västra Götaland"}, "City": {"S": "Göteborg"}, "Street": {"S": "Slanbarsvägen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T08:20:00
Online shop	sh#98765	PK	SK	EntityType	Quantity				
		o#12345	shp#55555	shipmentItem	2				
Online shop	sh#98765	PK	SK	EntityType	Quantity				
		o#12345	shp#12345	shipmentItem	3				
Online shop	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12345	sh#98765	{"Country": {"S": "Sweden"}, "County": {"S": "Västra Götaland"}, "City": {"S": "Göteborg"}, "Street": {"S": "Slanbarsvägen"}, "Number": {"S": "34"}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T10:20:00

GSI2

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouselt em	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouselt em	4				
	sh#98765	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
		o#12345	sh#98765	shipment	sh#98765	sh#98765	{"Country": {"S": "Sweden"}, "County": {"S": "Västra Götaland"}, "City": {"S": "Göteborg"}, "Street": {"S": "Slanbarsvägen"}, "Number": {"N": 34}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T10:20:00
c#12345	2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments": {"L": [{"M": {"Type": {"S": "GiftCard"}, "Amount": {"N": 100}, "Data": {"S": "GiftCard data here..."}}, {"M": {"Type": {"S": "MasterCard"}, "Amount": {"N": 300}, "Data": {"S": "Payment data here..."}}}]}, {"M": {"Type": {"S": "GiftCard"}, "Amount": {"N": 100}, "Data": {"S": "GiftCard data here..."}}, {"M": {"Type": {"S": "MasterCard"}, "Amount": {"N": 300}, "Data": {"S": "Payment data here..."}}}]}	400	2020-06-21T19:18:00
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	
w#12376	sh#88899	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
		o#12345	sh#88899	shipment	sh#88899	sh#88899	{"Country": {"S": "Sweden"}, "County": {"S": "Västra Götaland"}, "City": {"S": "Göteborg"}, "Street": {"S": "Slanbarsvägen"}, "Number": {"N": 34}, "ZipCode": {"S": "41787"}}	Express	2020-06-22T08:20:00
Online shop							rsvagen"}, "Number": {"S": 34}, "ZipCode": {"S": "41787"}}	API Version 2012-08-10 1453	

Using NoSQL Workbench with this schema design

You can import this final schema into [NoSQL Workbench](#), a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see [the section called “Download”](#).
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see [the section called “Importing an existing model”](#).
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see [the section called “Editing an existing model”](#).
5. To visualize your data model, add sample data, or import sample data from a CSV file, use the [Data Visualizer](#) feature of NoSQL Workbench.

Migrating to DynamoDB from a relational database

Migrating a relational database into DynamoDB requires careful planning to ensure a successful outcome. This guide will help you understand how this process works, what tools you have available, and then how to evaluate potential migration strategies and select one that'll fit your requirements.

Topics

- [Reasons to migrate to DynamoDB](#)
- [Considerations when migrating a relational database to DynamoDB](#)
- [Understanding how a migration to DynamoDB works](#)
- [Tools to help migrate to DynamoDB](#)
- [Choosing the appropriate strategy to migrate to DynamoDB](#)
- [Performing an offline migration to DynamoDB](#)
- [Performing a hybrid migration to DynamoDB](#)
- [Performing an online migration to DynamoDB by migrating each table 1:1](#)
- [Perform an online migration to DynamoDB using a custom staging table](#)

Reasons to migrate to DynamoDB

Migrating to Amazon DynamoDB presents a range of compelling benefits for businesses and organizations. Here are some key advantages that make DynamoDB an attractive choice for database migration:

- **Scalability:** DynamoDB is designed to handle massive workloads and scale seamlessly to accommodate growing data volumes and traffic. With DynamoDB, you can easily scale your database up or down based on demand, ensuring that your applications can handle sudden spikes in traffic without compromising performance.
- **Performance:** DynamoDB offers low-latency data access, enabling applications to retrieve and process data with exceptional speed. Its distributed architecture ensures that read and write operations are distributed across multiple nodes, delivering consistent, single-digit millisecond response times even at high request rates.
- **Fully managed:** DynamoDB is a fully managed service provided by AWS. This means that AWS handles the operational aspects of database management, including provisioning, configuration,

patching, backups, and scaling. This allows you to focus more on developing your applications and less on database administration tasks.

- **Serverless architecture:** DynamoDB supports a serverless model, known as [DynamoDB on-demand](#), where you pay only for the actual read and write requests your application makes with no upfront capacity provisioning required. This pay-per-request model offers cost efficiency and minimal operational overhead, as you only pay for the resources you consume without the need to provision and monitor capacity.
- **NoSQL flexibility:** Unlike traditional relational databases, DynamoDB follows a NoSQL data model, providing flexibility in schema design. With DynamoDB, you can store structured, semi-structured, and unstructured data, making it well-suited for handling diverse and evolving data types. This flexibility enables faster development cycles and easier adaptation to changing business requirements.
- **High availability and durability:** DynamoDB replicates data across multiple availability zones within a Region, ensuring high availability and data durability. It automatically handles replication, failover, and recovery, minimizing the risk of data loss or service disruptions. DynamoDB provides an availability SLA of up to 99.999%.
- **Security and compliance:** DynamoDB integrates with AWS Identity and Access Management for fine-grained access control. It provides encryption at rest and in-transit, ensuring the security of your data. DynamoDB also adheres to various compliance standards, including HIPAA, PCI DSS, and GDPR, enabling you to meet regulatory requirements.
- **Integration with AWS Ecosystem:** As part of the AWS ecosystem, DynamoDB seamlessly integrates with other AWS services, such as AWS Lambda, AWS CloudFormation, and AWS AppSync. This integration enables you to build serverless architectures, leverage infrastructure as code, and create real-time data-driven applications.

Considerations when migrating a relational database to DynamoDB

Relational database systems and NoSQL databases have different strengths and weaknesses. These differences make database design different between the two systems.

	Relational database	NoSQL database
Querying the database	In relational databases, data can be queried flexibly,	In a NoSQL database such as DynamoDB, data can be

	Relational database	NoSQL database
	<p>but queries are relatively expensive and don't scale well in high-traffic situations (see First steps for modeling relational data in DynamoDB). A relational database application may implement business logic in stored procedures, SQL subqueries, bulk update queries, and aggregation queries.</p>	<p>queried efficiently in a limited number of ways, outside of which queries can be expensive and slow. Writes to DynamoDB are singletons. Application business logic that formerly ran in stored procedures must be refactored to run outside of DynamoDB in custom code running on a host such as Amazon Amazon EC2 or AWS Lambda.</p>
Designing the database	<p>You design for flexibility without worrying about implementation details or performance. Query optimization generally doesn't affect schema design, but normalization is important.</p>	<p>You design your schema specifically to make the most common and important queries as fast and as inexpensive as possible. Your data structures are tailored to the specific requirements of your business use cases.</p>

Designing for NoSQL database requires a different mindset than designing for a relational database management system (RDBMS). For an RDBMS, you can create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. You can organize each type of data into its own table.

With NoSQL design, you shouldn't start designing your schema for DynamoDB until you know the questions it will need to answer. Understanding the business problems and the application read and write patterns is essential. You should also aim to maintain as few tables as possible in a DynamoDB application. Having fewer tables keeps things more scalable, requires less permissions management, and reduces overhead for your DynamoDB application. It can also help keep backup costs lower overall.

The task of modeling relational data for DynamoDB and building a new version of the front-end application is a [separate topic](#). This guide assumes you have a new version of your application built to use DynamoDB, but you still need to determine how best to migrate and synchronize historical data during the cutover.

Sizing Considerations

The maximum size of each item (row) that you store in a DynamoDB table is 400KB. For more information, see [Quotas and limits](#). The item size is determined by the total size of all attribute names and attribute values in an item. For more information, see [the section called “Item sizes and formats”](#).

If your application needs to store more data in an item than the DynamoDB size limit permits, break the item into an item collection, compress the item data, or store the item as an object in Amazon Simple Storage Service (Amazon S3) while storing the Amazon S3 object identifier in your DynamoDB item. See [the section called “Large items”](#). The cost to update an item is based on the full size of the item. For workloads that require frequent updates to existing items, having small items of one or two KB will cost less to update than larger items. See [the section called “Working with Item Collections”](#) for more information on item collections.

When choosing the partition and sort key attributes, other table settings, item size and structure, and whether to create secondary indexes, be sure to review the [DynamoDB Modeling documentation](#) as well as the guide for [the section called “Cost optimization”](#). Be sure to test your migration plan so your DynamoDB solution is cost efficient and fits within DynamoDB's features and limitations.

Understanding how a migration to DynamoDB works

Before reviewing the migration tools available to us, consider how writes are processed by DynamoDB.

Note

DynamoDB automatically shards and distributes your data to multiple shared servers and storage locations so there isn't a direct way to bulk-import a large dataset directly onto a production server.

The default and most common write operation is a single [PutItem](#) API operation. You can perform a PutItem operation in a loop to process data sets. DynamoDB supports virtually unlimited concurrent connections, so assuming you can configure and run a massively multi-threaded loading routine such as MapReduce or Spark, the velocity of writes is only limited by the capacity of the target table (which is also generally unlimited).

When loading data into DynamoDB, it's important to understand your loader's write velocity. If the items (rows) you are loading are 1KB in size or less, this velocity is simply the number of items per second. The target table can then be provisioned with sufficient WCU (write capacity units) to handle this rate. If your loader exceeds the provisioned capacity in any given second, the extra requests may be throttled or rejected altogether. You can check for throttles in the CloudWatch charts found in the DynamoDB console monitoring tab.

The second operation that can be performed is with a related API called [BatchWriteItem](#). BatchWriteItem allows you to combine up to 25 write requests into one API call. These are received by the service and processed as separate PutItem requests to the table. When choosing BatchWriteItem, you will not get the advantage of automatic retries that is included with the AWS SDK when making singleton calls with PutItem. So if there are any errors (such as throttling exceptions), you'll have to look for the list of any failed writes on the response call to BatchWriteItem. For more information on handling throttling warnings in case these are detected in the CloudWatch throttling charts, see [the section called "Throttling"](#).

The third type of data import is possible with the [DynamoDB Import from S3 feature](#). This feature allows you to stage a large dataset in Amazon S3 and ask DynamoDB to automatically import the data into a new table. The import is not instant and will take time proportional to the size of the dataset. However, it provides convenience since it requires no ETL platform or custom DynamoDB code to be written. The import feature has limitations that make it suitable for migrations when downtime is acceptable. The data from S3 is loaded into a new table that is created by the import, and is not available to load data into any existing table. There is no transformation of data performed, so it requires an upstream process to prepare and store the data in the final format to an S3 bucket.

Tools to help migrate to DynamoDB

There are several common migration and ETL tools you can use to migrate data into DynamoDB.

Many customers choose to write their own migration scripts and jobs in order to build custom data transformations for the migration process. If you plan to operate a high volume DynamoDB

table with heavy write traffic or regular large bulk load jobs, you may wish to build migration tools yourself in order to gain confidence in the behavior of DynamoDB under heavy write traffic. Scenarios such as throttle handling and efficient table provisioning can be experienced early in the project when performing a practice migration.

Amazon provides a host of data tools that can be leveraged, including [AWS Database Migration Service \(DMS\)](#), [AWS Glue](#), [Amazon EMR](#), and [Amazon Managed Streaming for Apache Kafka](#). All of these tools can be used to perform a downtime migration, and certain tools that can leverage relational database Change Data Capture (CDC) features can support online migrations as well. When choosing the best tool, it will help to consider the skill set and experience your organization has with each tool along with the features, performance and cost of each one.

Choosing the appropriate strategy to migrate to DynamoDB

A large relational database application may span a hundred or more tables and support several different application functions. When approaching a large migration, consider breaking your application into smaller components or micro-services, and migrating a small set of tables at a time. You can then migrate additional components to DynamoDB in waves.

When selecting a migration strategy, certain parameters may steer you towards one solution or another. We can present these options in a decision tree to simplify the options available to us given our requirements and resources available. The concepts are briefly mentioned here (but will be covered in more depth later in the guide):

- **Offline migration**: if your application can tolerate some downtime during the migration, it will greatly simplify the migration process.
- **Hybrid migration**: this approach would allow for partial uptime during a migration, such as allowing reads but not writes, or allowing reads and inserts but not updates and deletes.
- **Online migration**: applications that require zero downtime during migration are less easy to migrate, and may require significant planning and custom development. One key decision will be to estimate and weigh the costs of building a custom migration process versus the cost to the business of having a downtime window during cutover.

If	And	Then
You are okay to		Use AWS DMS and perform an offline

If	And	Then
take the application down for some time during a maintenance window to perform the data migration . This is an offline migration		migration using a full load task. Pre-shape the source data with a SQL VIEW if desired.
You are okay to run the application in read-only mode during migration . This is a hybrid migration		Disable writes within the application or source database. Use AWS DMS and perform an offline migration using a full load task.

If	And	Then
You are okay to run the application with reads and new record inserts, but no updates or deletes, during the migration. This is a hybrid migration	You have application development skills and can update the existing relational app to perform dual writes including to DynamoDB, for all new records	Use AWS DMS and perform an offline migration using a full load task. Concurrently, deploy a version of the existing app that allows reads and performs dual writes.
You require a migration with minimal downtime. This is an online migration	You are migrating source tables 1-for-1 into DynamoDB without major schema changes	Use AWS DMS to perform an online data migration. Run a bulk load task followed by CDC sync task
	You are combining source tables into fewer DynamoDB tables following single table philosophy	<p>You have backend database development skills and spare capacity on the SQL host</p> <p>Create the NoSQL-ready table within the SQL database. Populate and synchronize it using JOINS, UNIONs, VIEWS, triggers, stored procedures</p>

If	And	Then
	You do not have backend database development skills and spare capacity on the SQL host	Consider the hybrid or offline migration approaches
	You are okay to skip migrating historical transaction data, or can archive it in Amazon S3 in lieu of migrating it. You just need to migrate a few small static tables	Write a script or use any ETL tool to migrate the tables. Pre-shape the source data with a SQL VIEW if desired.

Performing an offline migration to DynamoDB

Offline migrations are suitable for when you can allow a downtime window to perform the migration. Relational databases commonly take at least some downtime each month for maintenance and patching, or may take longer downtimes for hardware upgrades or major release upgrades.

Amazon S3 can be used as a staging area during a migration. Data stored in CSV (comma separated values) or DynamoDB JSON format can be automatically imported into a new DynamoDB table using the [DynamoDB import from S3 feature](#).

Plan

Perform an offline migration using Amazon S3

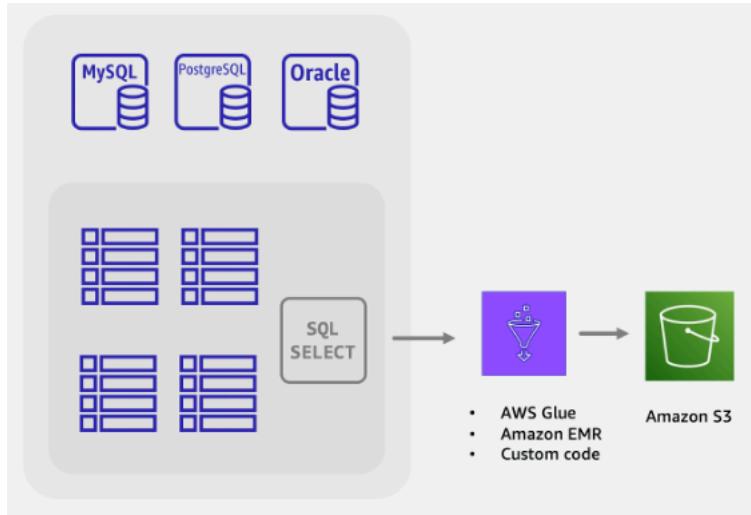
Tools

- An ETL job to extract and transform SQL data and store it in an S3 bucket such as:
 - AWS Glue

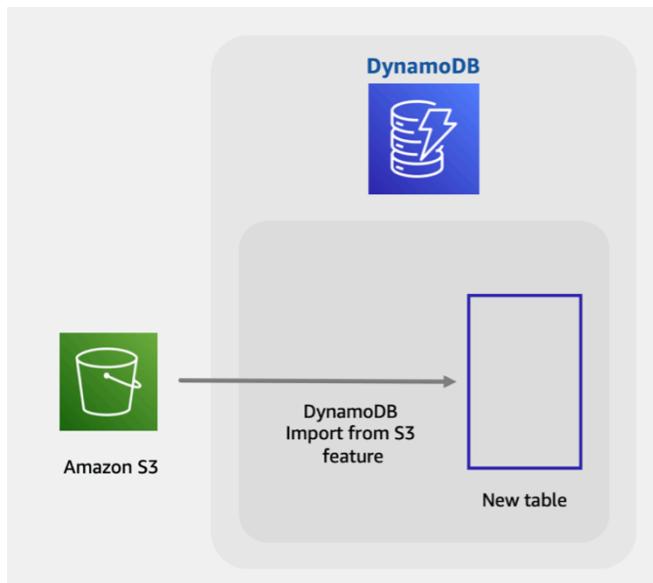
- Amazon EMR
- Your own custom code
- The DynamoDB import from S3 feature

Offline migration steps:

1. Build an ETL job that can query the SQL database, transform table data into DynamoDB JSON or CSV format, and save it to an S3 bucket.



2. The DynamoDB Import from S3 feature is invoked to create a new table and automatically load data from your S3 bucket.



The fully offline migration is simple and straightforward, but it may not be popular with application owners and users. Users would benefit if the application could provide reduced levels of service during the migration, instead of no service at all.

You could add functionality to disable writes during the offline migration, while allowing reads to continue as normal. Application users could still safely browse and query for existing data while the relational data is being migrated. If this is what you're looking for, continue reading to learn about [hybrid migrations](#).

Performing a hybrid migration to DynamoDB

While all database applications perform read and write operations, the types of write operations being performed should be considered when planning a hybrid or online migration. Database writes can be classified into three buckets: inserts, updates, and deletes. Some applications do not perform any updates to existing records. Other applications may not require deletes to be processed immediately, and could defer deletes to a bulk clean up process at the end of the month, for example. These types of applications can be simpler to migrate while allowing partial uptime.

Plan

Perform a hybrid online/offline migration with application dual writes

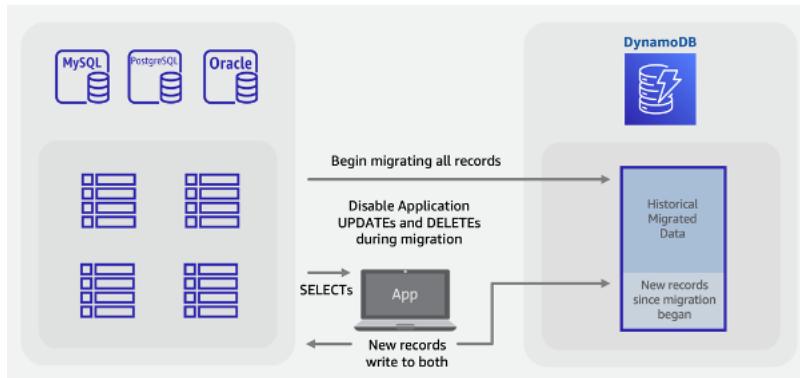
Tools

- An ETL job to extract and transform SQL data and store it in an S3 bucket such as:
 - AWS Glue
 - Amazon EMR
 - Your own custom code

Hybrid migration steps:

1. Create the target DynamoDB table. This table will receive both historical bulk data and new, live data
2. Create a version of the legacy application that has deletes and updates disabled while performing all inserts as dual writes to both the SQL database and DynamoDB
3. Begin the ETL job to migrate existing data and deploy the new application version at the same time

- When the ETL job completes, DynamoDB will have all existing and new records and be ready for application cutover



Note

The ETL job writes directly from SQL to DynamoDB. We are unable to use the S3 import feature as in the offline migration example, since the target table does not become public and available for other writes until the entire import completes.

Performing an online migration to DynamoDB by migrating each table 1:1

Many relational databases have a feature called Change Data Capture (CDC), where the database allows users to request a list of the changes to a table that happened before or after a specific point in time. CDC uses internal logs to enable this feature and it does not require the table to have any timestamp column in order to work.

When migrating a schema of SQL tables to a NoSQL database, you might want to combine and reshape your data into fewer tables. Doing so will allow you to collect data in a single place and avoid having to manually join related data in multi-step read operations. However, single table data shaping is not always required and sometimes you'll migrate tables 1-for-1 into DynamoDB. These 1-to-1 table migrations are less complicated as you can leverage the source database CDC feature, using common ETL tools that support this type of migration. The data for each row may still be transformed into new formats, but the scope of each table remains the same.

Consider migrating SQL tables 1-to-1 into DynamoDB, with the caveat that there are no server-side joins.

Plan

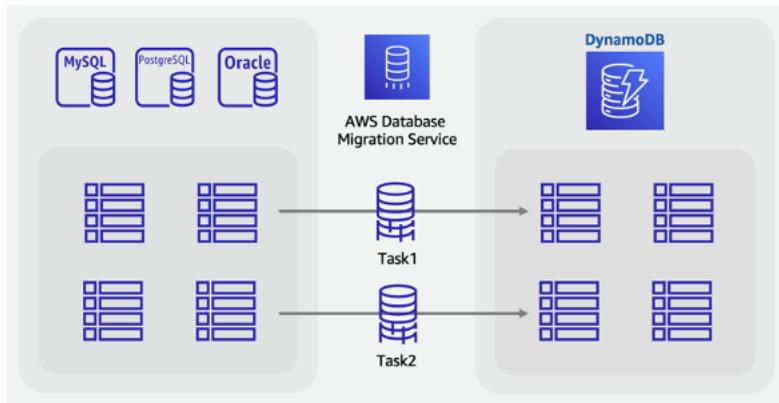
Perform an online migration of each table into DynamoDB using AWS DMS

Tools

- [AWS Database Migration Service \(DMS\)](#), an ETL tool that can both bulk load historical data and also leverage CDC to synchronize source and target tables

Online migration steps:

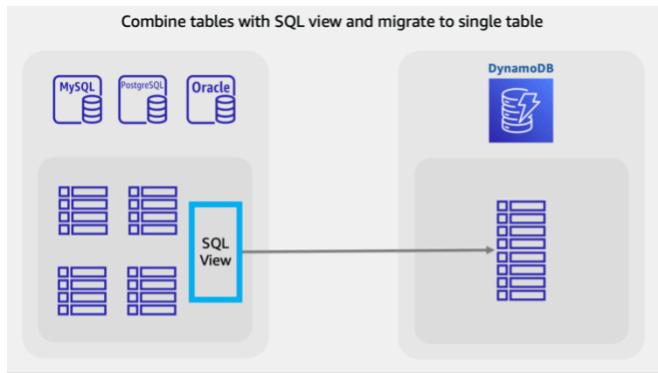
1. Identify the tables in your source schema that will be migrated
2. Create the same number of tables in DynamoDB with a similar key structure
3. Create a replication server in AWS DMS and configure the source and target endpoints
4. Define any per-row transformations required (such as concatenated columns or conversion of dates to ISO-8601 string format)
5. Create a migration task for each table for **Full Load and Change Data Capture**
6. Monitor these tasks until the ongoing replication phase has begun
7. At this point you may perform any validation audits and then switch users to the application that reads and writes to DynamoDB



Perform an online migration to DynamoDB using a custom staging table

You may wish to combine tables to leverage unique NoSQL access patterns (for example, transforming four legacy tables into one single DynamoDB table). A single key-value document

request, or a query for a pre-grouped item collection will usually return with better latency than a SQL database that performs a multi-table join. However, this makes the migration task becomes more difficult. A SQL VIEW could do the work within the source database to prepare a single dataset representing all four tables in one set.



This view might JOIN tables into a denormalized form, or could instead keep the entities normalized and stack tables using a SQL UNION. Key decisions around re-shaping relational data are covered in [this video](#). For offline migrations, using a view to combine tables is a great way to shape data for a DynamoDB single table schema.

However, for online migrations with live, changing data, you are unable to leverage CDC features as they are only supported for single table queries, not from within a VIEW. If your tables include a last-updated timestamp column, and these are incorporated into the VIEW, you can then build a custom ETL job that uses these to achieve a bulk load with synchronization.

A novel approach to this challenge would be to use standard SQL features such as views, stored procedures, and triggers to create a new SQL table that is in the final desired DynamoDB NoSQL format.

If your database server can allocate an additional amount of storage space, it is possible to create this single staging table before migration begins. This would be accomplished by writing a stored procedure that will read from existing tables, transform data as needed, and write to the new staging table. A set of triggers could be added to replicate changes in tables into the staging table in real time. If triggers are not allowed per company policy, changes to stored procedures can accomplish the same result. You would add a few lines of code to any procedure that writes data, to additionally write the same changes into the staging table.

Having this staging table in place that is fully synchronized with the legacy application tables will give you a great starting point for a live migration. Tools using database CDC to accomplish live migrations, such as AWS DMS, can now be used against this table. An advantage of this approach is that it uses well-known SQL skills and features available in the relational database engine.

Plan

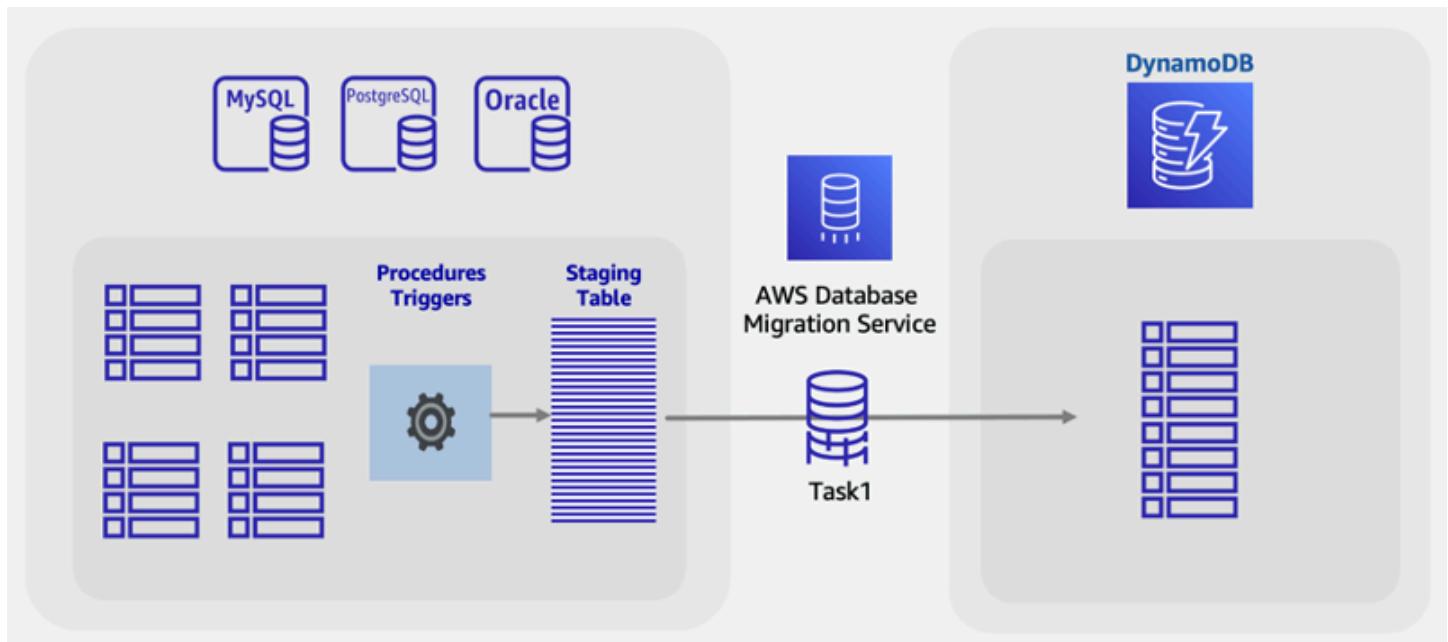
Perform an online migration with an SQL staging table using AWS DMS

Tools

- Custom SQL stored procedures or triggers
- [AWS Database Migration Service \(DMS\)](#), an ETL tool that can migrate a live staging table into DynamoDB

Online migration steps:

1. Within the source relational database engine, ensure there is some extra disk space and processing capacity.
2. Create a new staging table in the SQL database, with timestamps or CDC features enabled
3. Write and run a stored procedure to copy existing relational table data into the staging table
4. Deploy triggers or modify existing procedures to dual write into the new staging table while performing normal writes to existing tables
5. Run AWS DMS to migrate and synchronize this source table to a target DynamoDB table



This guide presented several considerations and approaches for migrating relational database data into DynamoDB, with a focus on minimizing downtime and using common database tools and techniques. For more information, see the following:

- [AWS DMS User Guide](#)
- [AWS Glue User Guide](#)
- [Best Practices for Migrating from RDBMS to DynamoDB](#)

NoSQL Workbench for DynamoDB

NoSQL Workbench for Amazon DynamoDB is a cross-platform, client-side GUI application that you can use for modern database development and operations. It's available for Windows, macOS, and Linux. NoSQL Workbench is a visual development tool that provides data modeling, data visualization, and query development features to help you design, create, query, and manage DynamoDB tables. NoSQL Workbench now includes DynamoDB local as an optional part of the installation process, which makes it easier to model your data in DynamoDB local. To learn more about DynamoDB local and its requirements, see [Setting up DynamoDB local \(downloadable version\)](#).

Data modeling

With NoSQL Workbench for DynamoDB, you can build new data models from, or design models based on, existing data models that satisfy your application's data access patterns. You can also import and export the designed data model at the end of the process. For more information, see [Building data models with NoSQL Workbench](#).

Data visualization

The data model visualizer provides a canvas where you can map queries and visualize the access patterns (facets) of the application without having to write code. Every facet corresponds to a different access pattern in DynamoDB. You can autogenerate sample data for use in your data model. For more information, see [Visualizing data access patterns](#).

Operation building

NoSQL Workbench provides a rich graphical user interface for you to develop and test queries. You can use the *operation builder* to view, explore, and query live datasets. The structured operation builder supports projection expression, condition expression, and generates sample code in multiple languages. You can directly clone tables from one Amazon DynamoDB account to another one in different Regions. You can also directly clone tables between DynamoDB local and an Amazon DynamoDB account for faster copying of your table's key schema (and optionally GSI schema and items) between your development environments. For more information, see [Exploring datasets and building operations with NoSQL Workbench](#).

Topics

- [Download NoSQL Workbench for DynamoDB](#)

- [Install NoSQL Workbench for DynamoDB](#)
- [Building data models with NoSQL Workbench](#)
- [Visualizing data access patterns](#)
- [Exploring datasets and building operations with NoSQL Workbench](#)
- [Sample data models for NoSQL Workbench](#)
- [Release history for NoSQL Workbench](#)

Download NoSQL Workbench for DynamoDB

Follow these instructions to download NoSQL Workbench and DynamoDB local* for Amazon DynamoDB.

Prerequisites

There are two prerequisite pieces of software required for Ubuntu installs: libfuse2 and curl.

libfuse2

As of Ubuntu 22.04, libfuse2 is no longer installed by default. To solve this, run `sudo add-apt-repository universe && sudo apt install libfuse2` to install for the [newest Ubuntu version](#).

curl

Update Ubuntu, run `sudo apt update && sudo apt upgrade`

Next, install curl, execute: `sudo apt install curl`

To download NoSQL Workbench and DynamoDB local

1. Download the appropriate version of NoSQL Workbench for your operating system.

Operating system	Download link
macOS (Intel)**	Download for macOS (Intel)
macOS (Apple silicon)	Download for macOS (Apple silicon)
Windows	Download for Windows

Operating system	Download link
Linux***	Download for Linux

* NoSQL Workbench includes DynamoDB local as an optional part of the installation process.

** If a warning message appears when you try to open NoSQL Workbench stating that the app isn't registered with Apple by an identified developer, do the following:

1. Locate the app and then open it.
2. Control+click the app icon, then choose Open from the shortcut menu.

This saves the app as an exception to your security settings. Open the app by double-clicking it just as you can open any registered app.

*** NoSQL Workbench supports Ubuntu 12.04, Fedora 21, and Debian 8, or any newer versions of these Linux distributions.

2. Start the application that you downloaded, and then follow the steps in [Install NoSQL Workbench](#).

 **Note**

Java Runtime Environment (JRE) version 11.x or newer is required for running DynamoDB local.

Install NoSQL Workbench for DynamoDB

Follow these steps to install NoSQL Workbench and DynamoDB local on a supported platform.

Windows

To install NoSQL Workbench on Windows

1. Run the NoSQL Workbench installer application and choose the setup language. Then choose **OK** to begin the setup. For more information about downloading NoSQL Workbench, see [Download NoSQL Workbench for DynamoDB](#).

2. Choose **Next** to continue the setup, and then choose **Next** on the following screen.
3. By default, the **Install DynamoDB Local** check box is selected to include DynamoDB local as part of the installation. Keeping this option selected ensures that DynamoDB local will be installed, and the destination path will be the same as the installation path of NoSQL Workbench. Clearing the check box for this option will skip the installation of DynamoDB local, and the installation path will be for NoSQL Workbench only.

Choose the destination where you want the software installed, and choose **Next**.

 **Note**

If you opted to not include DynamoDB local as part of the setup, clear the **Install DynamoDB Local** check box, choose **Next**, and skip to step 6. You can download DynamoDB local separately as a standalone installation at a later time. For more information, see [Setting up DynamoDB local \(downloadable version\)](#).

Configure your installation path during this step.

4. Choose the port number for DynamoDB local to use. The default port is 8000. After you enter the port number, choose **Next**.
5. Choose **Next** to begin setup.
6. When the setup has completed, choose **Finish** to close the setup screen.
7. Open the application in your installation path, such as `/programs/DynamoDBWorkbench/`.

macOS

To install NoSQL Workbench on macOS

1. Run the NoSQL Workbench installer application and choose the setup language. Then choose **OK** to begin the setup. For more information about downloading NoSQL Workbench, see [Download NoSQL Workbench for DynamoDB](#).
2. Choose **Next** to continue the setup, and then choose **Next** on the following screen.
3. By default, the **Install DynamoDB local** check box is selected to include DynamoDB local as part of the installation. Keeping this option selected ensures that DynamoDB local will be installed, and the destination path will be the same as the installation path of NoSQL

Workbench. Clearing this option will skip the installation of DynamoDB local, and the installation path will be for NoSQL Workbench only.

Choose the destination where you want the software installed, and choose **Next**.

 **Note**

If you opted to not include DynamoDB local as part of the setup, clear the **Install DynamoDB local** check box, choose **Next**, and skip to step 6. You can download DynamoDB local separately as a standalone installation at a later time. For more information, see [Setting up DynamoDB local \(downloadable version\)](#).

Configure your installation path during this step.

4. Choose the port number for DynamoDB local to use. The default port is 8000. After you enter the port number, choose **Next**.
5. Choose **Next** to begin setup.
6. When the setup has completed, choose **Finish** to close the setup screen.
7. Open the application in your installation path, such as */Applications/DynamoDBWorkbench/*.

 **Note**

NoSQL Workbench for macOS performs auto-updates. To get notification about updates, enable notification access to NoSQL Workbench in System Preferences > Notifications.

Linux

To install NoSQL Workbench on Linux

1. Run the NoSQL Workbench installer application and choose the setup language. Then choose **OK** to begin the setup. For more information about downloading NoSQL Workbench, see [Download NoSQL Workbench for DynamoDB](#).
2. Choose **Forward** to continue the setup, and choose **Forward** on the following screen.

3. By default, the **Install DynamoDB local** check box is selected to include DynamoDB local as part of the installation. Keeping this option selected ensures that DynamoDB local will be installed, and the destination path will be the same as the installation path of NoSQL Workbench. Clearing this option will skip the installation of DynamoDB local, and the installation path will be for NoSQL Workbench only.

Choose the destination where you want the software installed, and choose **Forward**.

 **Note**

If you opted to not include DynamoDB local as part of the setup, clear the **Install DynamoDB local** check box, choose **Forward**, and skip to step 6. You can download DynamoDB local separately as a standalone installation at a later time. For more information, see [Setting up DynamoDB local \(downloadable version\)](#).

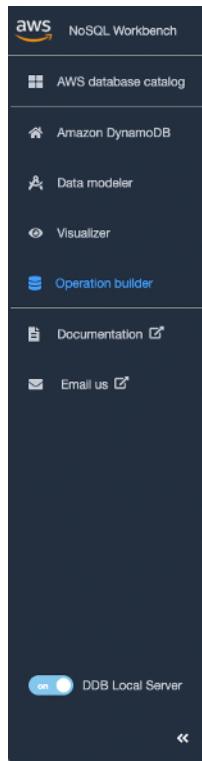
Configure your installation path during this step.

4. Choose the port number for DynamoDB local to use. The default port is 8000. After you enter the port number entered, choose **Forward**.
5. Choose **Forward** to begin setup.
6. When the setup has completed, choose **Finish** to close the setup screen.
7. Open the application in your installation path, such as `/usr/local/programs/DynamoDBWorkbench/`.

 **Note**

If you opted to install DynamoDB local as part of the installation of NoSQL Workbench, DynamoDB local will be preconfigured with default options. To edit the default options, modify the `DDBLocalStart` script located in the `/resources/DDBLocal_Scripts/` directory. You can find this in the path that you provided during installation. To learn more about DynamoDB local options, see [DynamoDB local usage notes](#).

If you opted to install DynamoDB local as part of the NoSQL Workbench installation, you will have access to a toggle to enable and disable DynamoDB local as shown in the following image.



Building data models with NoSQL Workbench

You can use the data modeler tool in NoSQL Workbench for Amazon DynamoDB to build new data models, or to design models based on existing data models that satisfy your applications' data access patterns. The data modeler includes a few sample data models to help you get started.

Topics

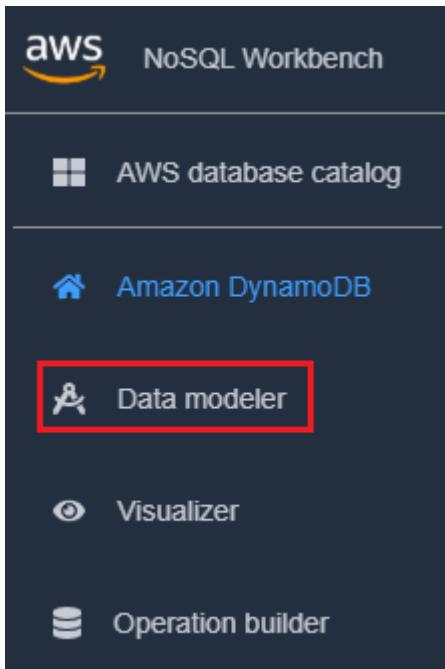
- [Creating a new data model](#)
- [Importing an existing data model](#)
- [Exporting a data model](#)
- [Editing an existing data model](#)

Creating a new data model

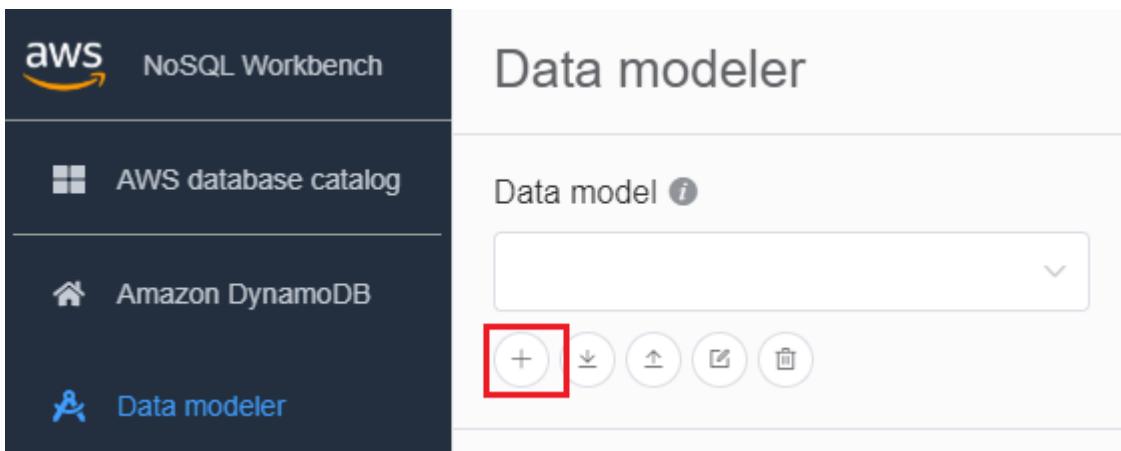
Follow these steps to create a new data model in Amazon DynamoDB using NoSQL Workbench.

To create a new data model

1. Open NoSQL Workbench, and in the navigation pane on the left side, choose the **Data modeler** icon.



2. Choose **Create data model**.



Create data model has two choices: Make model from scratch and Start from a template.

Create data model for Amazon DynamoDB

Make model from scratch



Selecting this option means you will have to create all tables, GSIs, attributes and elements yourself.

[Select](#)

Start from a template

Start with tables, GSIs and attributes to help guide you without losing any freedom to change everything.

AWS Discussion Forum Data Model

[Select](#)

This data model represents Amazon DynamoDB schema for AWS discussion forums, an example of an application for discussion forums or message boards....

Bookmarks Data Model

[Select](#)

This model is about storing URL bookmarks for customers. Even if the use case is relatively simple, there are still many interesting...

Employee Data Model

[Select](#)

This data model represents an Amazon DynamoDB schema for an employee database application. The important access patterns facilitated by this data...

[More templates](#)

OR

[Cancel](#)

Make model from scratch

To make a model from scratch, enter a name, author, and description for the data model. Choose **Create** when finished.

Create data model for Amazon DynamoDB

* Name

Author

Description

[Back](#) [Cancel](#) [Create](#)

Start from a template

Starting from a template lets you choose a sample model to start from. Choose **More templates** to see more template options. Choose **Select** for the template that you want to use.

Enter a data model name, author, and description for the template you selected. You can choose between **Schema only** and **Schema with sample data**.

- **Schema only** creates an empty data model with the primary key (partition and sort key) and other attributes.
- **Schema with sample data** will create a data model complete with sample data for the primary key (partition and sort key) and other attributes.

When this information is complete, choose **Create** to create the model.

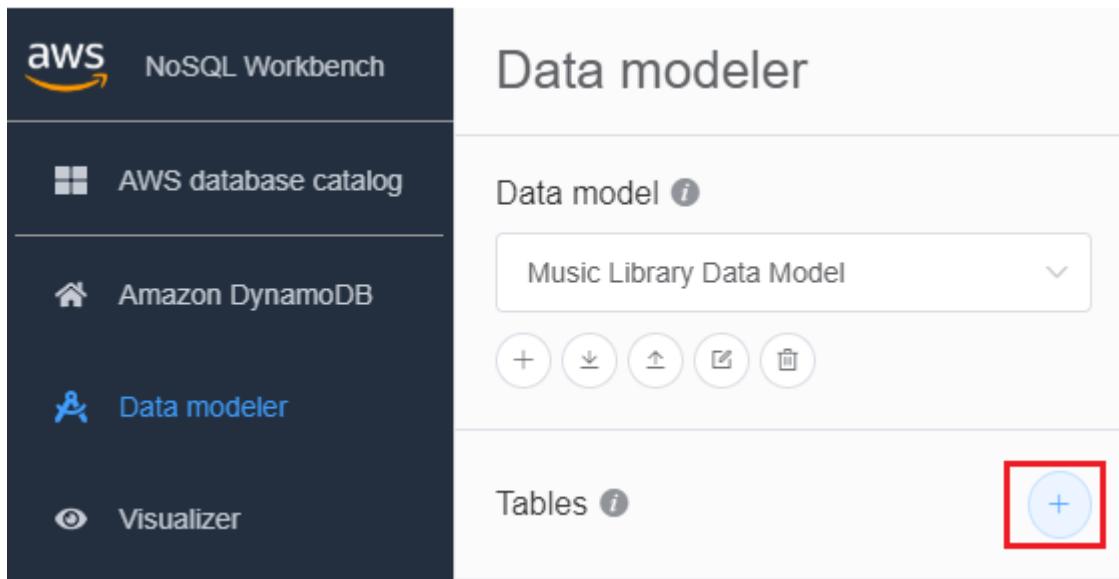
Create data model for Amazon DynamoDB

Data Model	New model	From template
Template	Ski Resort Data Model	▼
* Save as	Enter data model name	
Author	Enter author name	
Description	Describe this data model	
Sample Data	Schema only	Schema with sample data

Schema with sample data will create a data model complete with sample data for the primary keys (partition key and/or sort key) and other attributes.

[Back](#) [Cancel](#) [Create](#)

3. With the model created, choose **Add table**.



For more information about tables, see [Working with tables in DynamoDB](#).

4. Specify the following:

- **Table name** – Enter a unique name for the table.
- **Partition key** – Enter a partition key name, and specify its type. Optionally, you can also select a more granular data type format for sample data generation.
- If you want to add a sort key:
 1. Select **Add sort key**.
 2. Specify the sort key name and its type. Optionally, you can select a more granular data type format for sample data generation.

Note

To learn more about primary key design, designing and using partition keys effectively, and using sort keys, see the following:

- [Primary key](#)
- [Best practices for designing and using partition keys effectively](#)
- [Best practices for using sort keys to organize data](#)

5. To add other attributes, do the following for each attribute:

1. Choose **Add an attribute**.

2. Specify the attribute name and its type. Optionally, you can select a more granular data type format for sample data generation.
6. Add a facet:

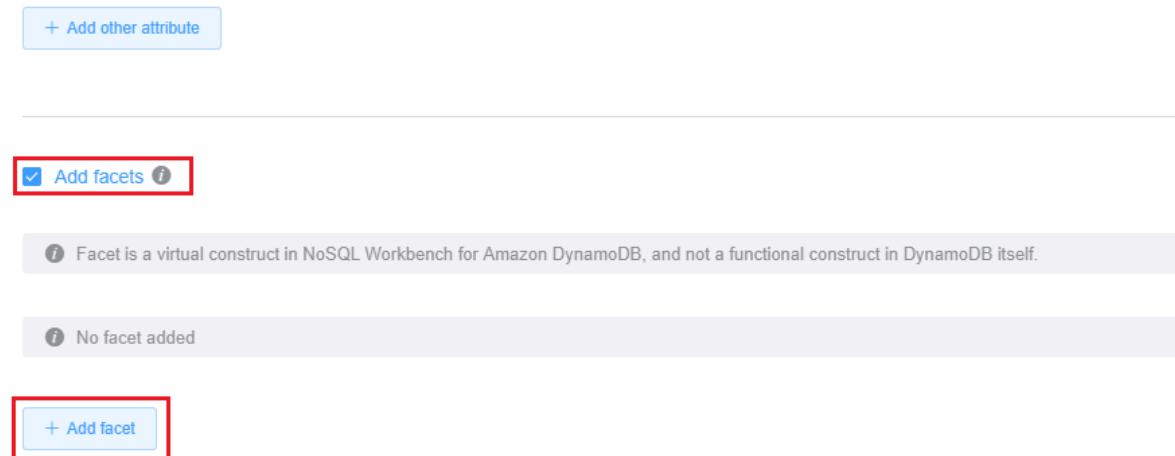
You can optionally add a facet. A facet is a virtual construct in NoSQL Workbench. It is not a functional construct in DynamoDB itself.

 **Note**

Facets in NoSQL Workbench help you visualize an application's different data access patterns for Amazon DynamoDB with only a subset of the data in a table. To learn more about facets, see [Viewing data access patterns](#).

To add a facet,

- Select **Add facets**.
- Choose **Add facet**.



- Specify the following:
 - The **Facet name**.
 - A **Partition key alias** to help distinguish this facet view.
 - A **Sort key alias**.
 - Choose the **Other attributes** that are part of this facet.

Choose Add facet.

[Add facets](#) i

i Facet is a virtual construct in NoSQL Workbench for Amazon DynamoDB, and not a functional construct in DynamoDB itself.

i No facet added

Add facet

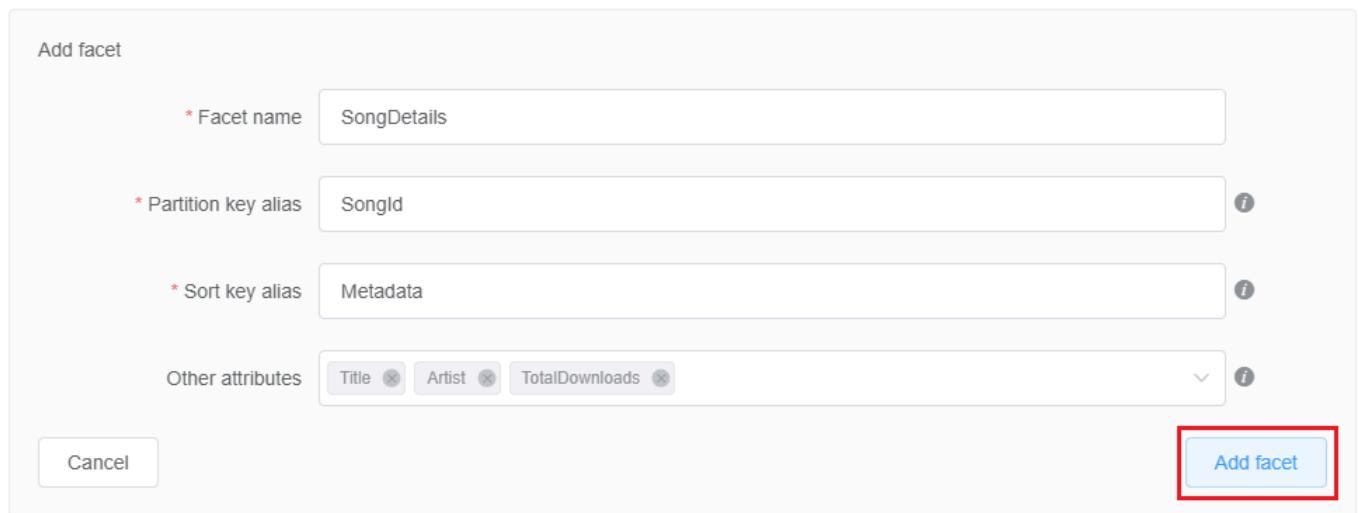
* Facet name

* Partition key alias i

* Sort key alias i

Other attributes v i

[Cancel](#) [Add facet](#)



Repeat this step if you want to add more facets.

7. If you want to add a global secondary index, choose **Add global secondary index**.

Specify the **Global secondary index name**, **Partition key** attribute, and **Projection type**.

Global secondary indexes

The screenshot shows the 'Global secondary indexes' configuration screen. It includes fields for 'Name' (index name), 'Partition key' (set to 'FirstName'), 'Add sort key' (checked), 'Sort key' (set to 'LastName'), and 'Projection type' (set to 'ALL'). A red box highlights the '+ Add global secondary index' button at the bottom left.

For more information about working with global secondary indexes in DynamoDB, see [Global secondary indexes](#).

8. By default, your table will use provisioned capacity mode with auto scaling enabled on both read and write capacity. If you want to change these settings, clear **Inherit capacity settings from base table** under **Capacity settings**.

Select your desired capacity mode, read and write capacity, and auto scaling IAM role (if applicable).

For more information about DynamoDB capacity settings, see [Read/write capacity mode](#).

9. Save the edits to your table settings..

A modal dialog box with 'Cancel' and 'Save edits' buttons.

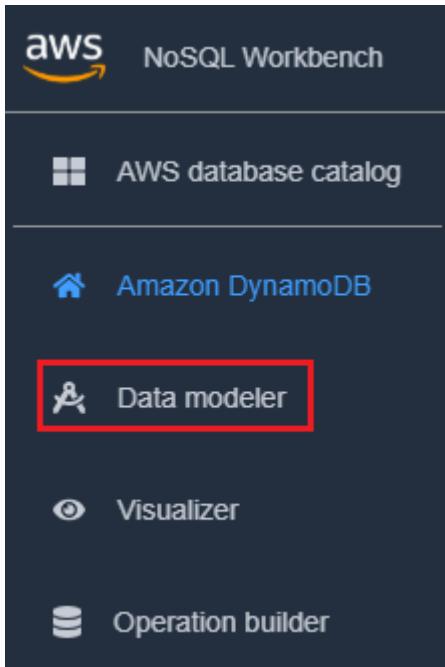
For more information about the `CreateTable` API operation, see [CreateTable](#) in the [Amazon DynamoDB API Reference](#).

Importing an existing data model

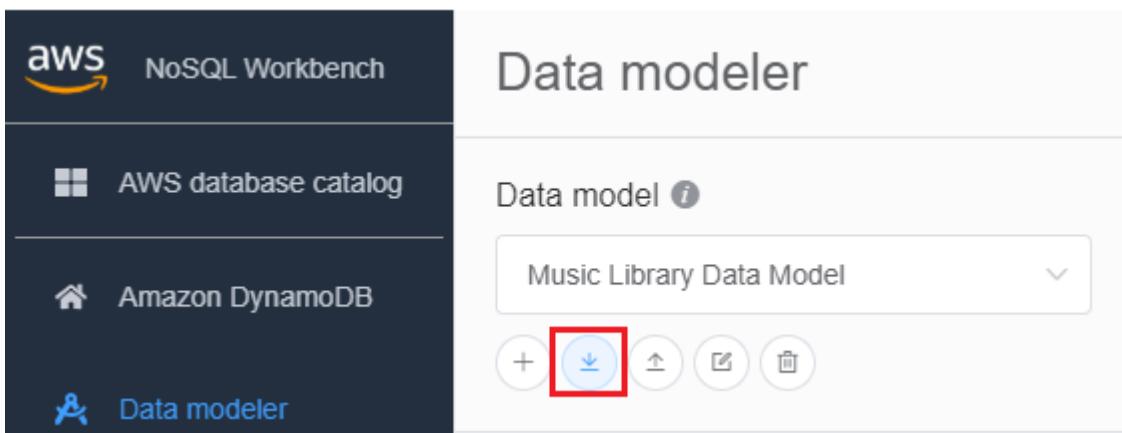
You can use NoSQL Workbench for Amazon DynamoDB to build a data model by importing and modifying an existing model. You can import data models in either NoSQL Workbench model format or in [AWS CloudFormation JSON template format](#).

To import a data model

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Data modeler** icon.

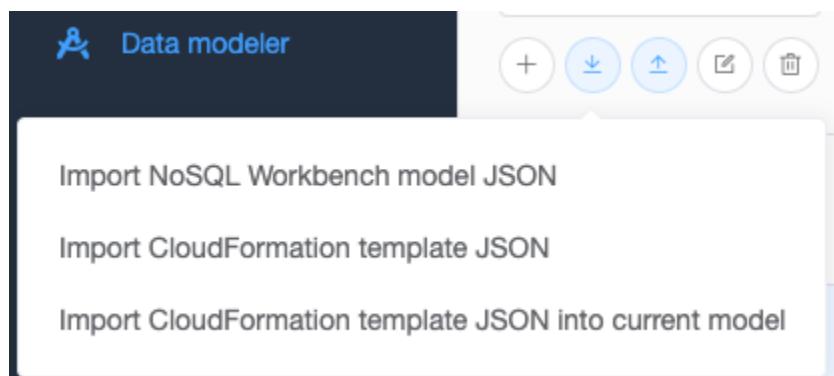


2. Hover your pointer over **Import data model**.

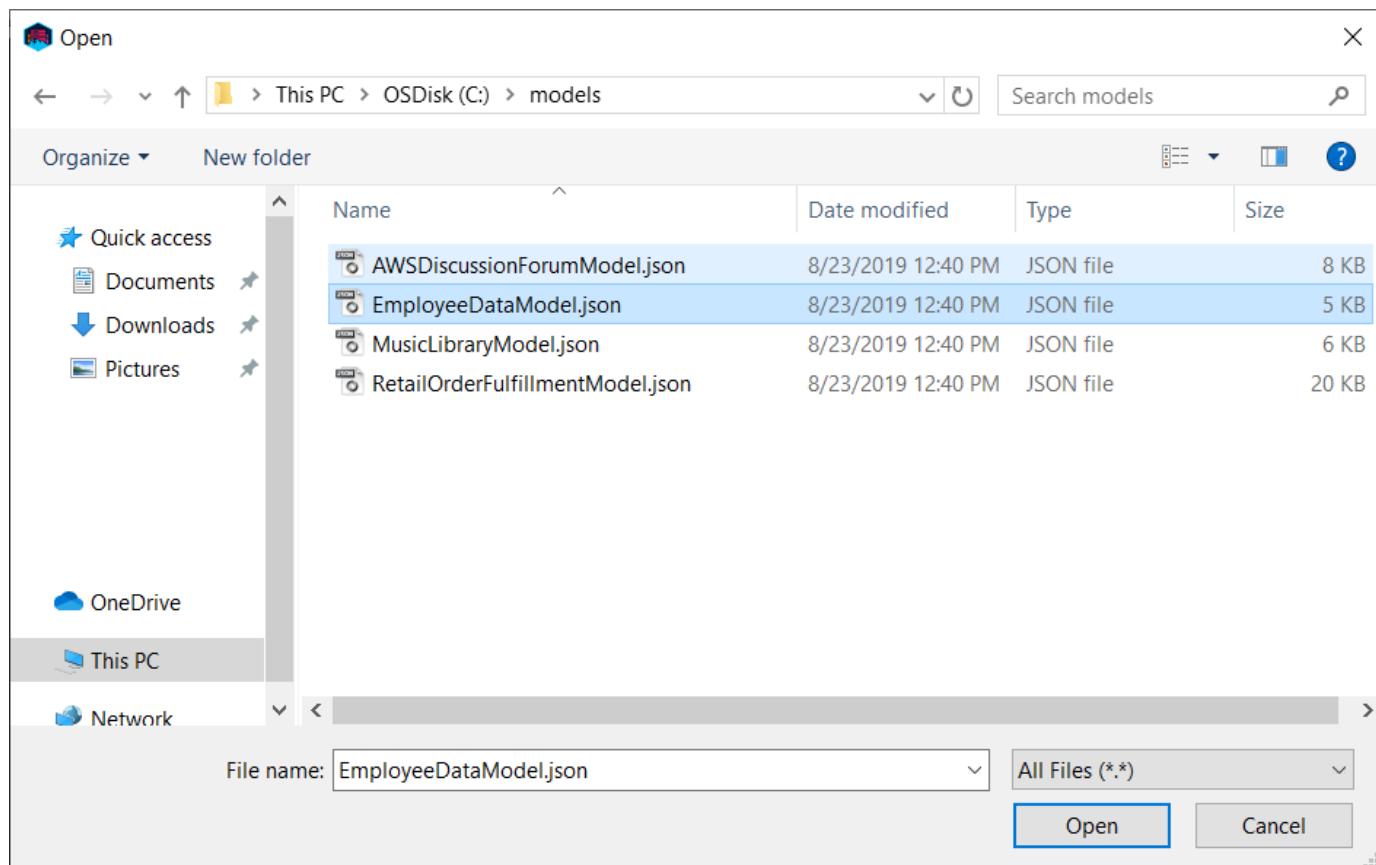


In the dropdown list, choose whether the model you want to import is in NoSQL Workbench model format or CloudFormation JSON template format. If you have an existing data model

open in NoSQL Workbench, you'll have the option to import a CloudFormation template into the current model.



3. Choose a model to import.



4. If the model you're importing is in CloudFormation template format, you'll see a list of tables to be imported and have an opportunity to specify a data model name, author, and description.

Create data model for Amazon DynamoDB

Only CloudFormation resources related to DynamoDB: tables and any related application auto scaling, **i** will be imported. Some fields within these resources are not supported by NoSQL Workbench and will also not be imported, including LocalSecondaryIndexes, RoleARN, and PolicyName.

Successfully imported tables (1)



Data model information

* Name

Enter data model name

Author

Enter author name

Description

Describe this data model

Cancel

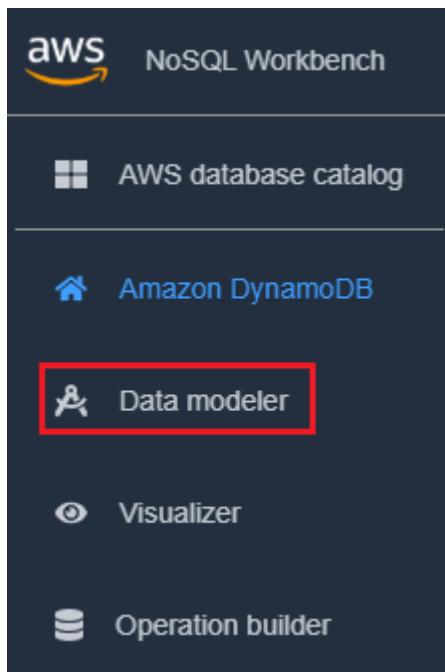
Create

Exporting a data model

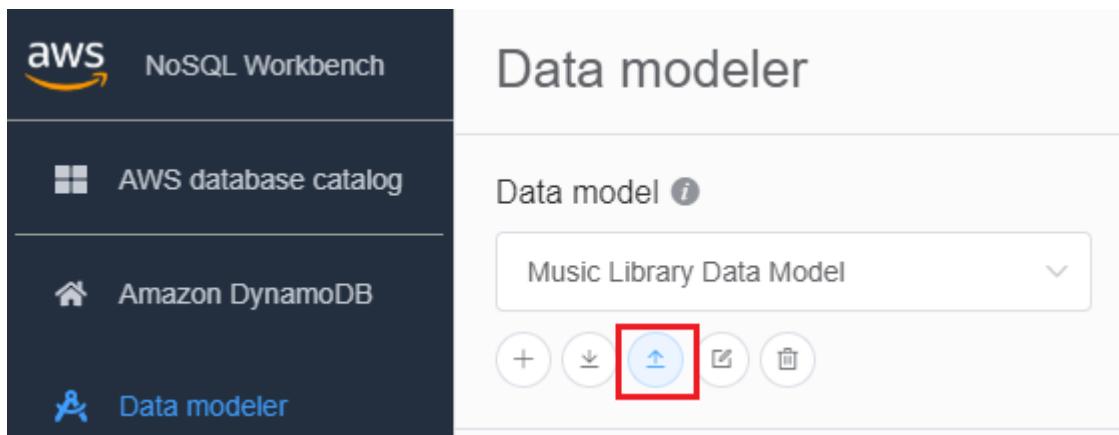
After you create a data model using NoSQL Workbench for Amazon DynamoDB, you can save and export the model in either NoSQL Workbench model format or [AWS CloudFormation JSON template format](#).

To export a data model

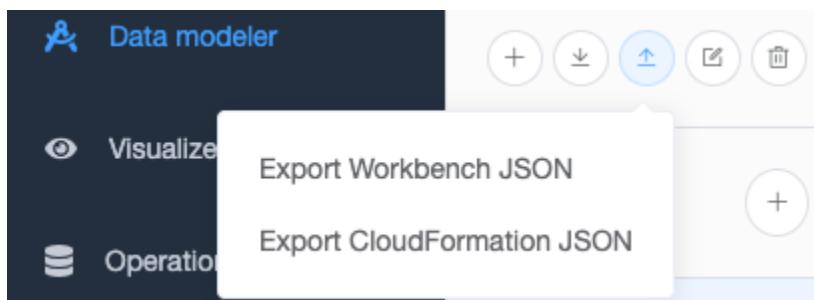
1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Data modeler** icon.



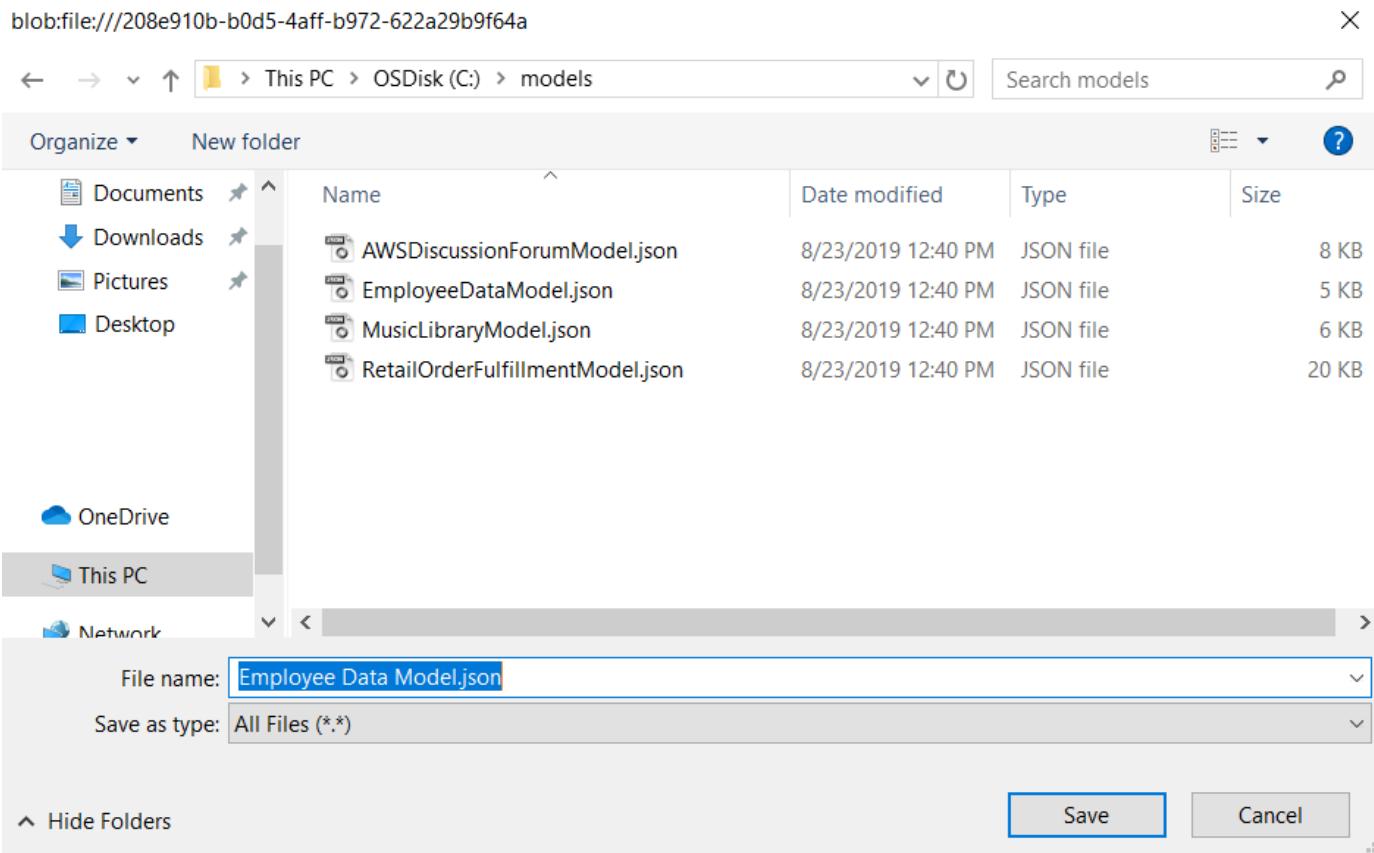
2. Hover your pointer over **Export data model**.



In the dropdown list, choose whether to export your data model in NoSQL Workbench model format or CloudFormation JSON template format.



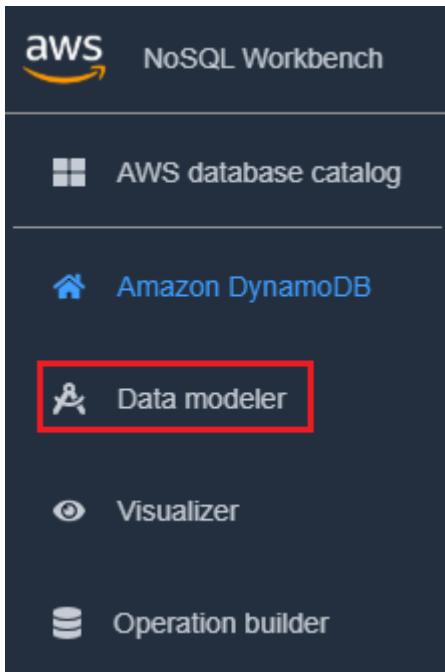
3. Choose a location to save your model.



Editing an existing data model

To edit an existing model

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Data modeler** button.



2. Select the data model and choose the table that you want to edit. Choose **Edit model**

The screenshot shows the Data modeler interface for the 'Songs' table. The 'Edit model' button is highlighted with a red box.

Attribute name	Attribute type	Key type	Sample data format
Songs			

3. Make the needed edits, and then choose **Save edits**.

To manually edit an existing model and add a facet

1. Export your model. For more information, see [Exporting a data model](#).
2. Open the exported file in an editor.
3. Locate the DataModel Object for the table that you want to create a facet for.

Add a TableFacets array representing all the facets for the table.

For each facet, add an object to the TableFacets array. Each array element has the following properties:

- FacetName – A name for your facet. This value must be unique across the model.

- **PartitionKeyAlias** – A friendly name for the table's partition key. This alias is displayed when you view the facet in NoSQL Workbench.
- **SortKeyAlias** – A friendly name for the table's sort key. This alias is displayed when you view the facet in NoSQL Workbench. This property is not needed if the table has no sort key defined.
- **NonKeyAttributes** – An array of attribute names that are needed for the access pattern. These names must map to the attribute names that are defined for the table.

```
{  
    "ModelName": "Music Library Data Model",  
    "DataModel": [  
        {  
            "TableName": "Songs",  
            "KeyAttributes": {  
                "PartitionKey": {  
                    "AttributeName": "Id",  
                    "AttributeType": "S"  
                },  
                "SortKey": {  
                    "AttributeName": "Metadata",  
                    "AttributeType": "S"  
                }  
            },  
            "NonKeyAttributes": [  
                {  
                    "AttributeName": "DownloadMonth",  
                    "AttributeType": "S"  
                },  
                {  
                    "AttributeName": "TotalDownloadsInMonth",  
                    "AttributeType": "S"  
                },  
                {  
                    "AttributeName": "Title",  
                    "AttributeType": "S"  
                },  
                {  
                    "AttributeName": "Artist",  
                    "AttributeType": "S"  
                },  
                {  
                    "AttributeName": "SongLength",  
                    "AttributeType": "N"  
                }  
            ]  
        }  
    ]  
}
```

```
        "AttributeName": "TotalDownloads",
        "AttributeType": "S"
    },
    {
        "AttributeName": "DownloadTimestamp",
        "AttributeType": "S"
    }
],
"TableFacets": [
    {
        "FacetName": "SongDetails",
        "KeyAttributeAlias": {
            "PartitionKeyAlias": "SongId",
            "SortKeyAlias": "Metadata"
        },
        "NonKeyAttributes": [
            "Title",
            "Artist",
            "TotalDownloads"
        ]
    },
    {
        "FacetName": "Downloads",
        "KeyAttributeAlias": {
            "PartitionKeyAlias": "SongId",
            "SortKeyAlias": "Metadata"
        },
        "NonKeyAttributes": [
            "DownloadTimestamp"
        ]
    }
]
```

4. You can now import the modified model into NoSQL Workbench. For more information, see [Importing an existing data model](#).

Visualizing data access patterns

You can use the visualizer tool in NoSQL Workbench for Amazon DynamoDB to map queries and visualize different access patterns (known as *facets*) of an application. Every facet corresponds to a different access pattern in DynamoDB. You can also manually add data to your data model or import data from MySQL.

Topics

- [Adding sample data to a data model](#)
- [Importing sample data from a CSV file](#)
- [Viewing data access patterns](#)
- [Viewing all tables in a data model using aggregate view](#)
- [Committing a data model to DynamoDB](#)

Adding sample data to a data model

By adding sample data to your model, you can display data when visualizing the model and its various data access patterns, or *facets*.

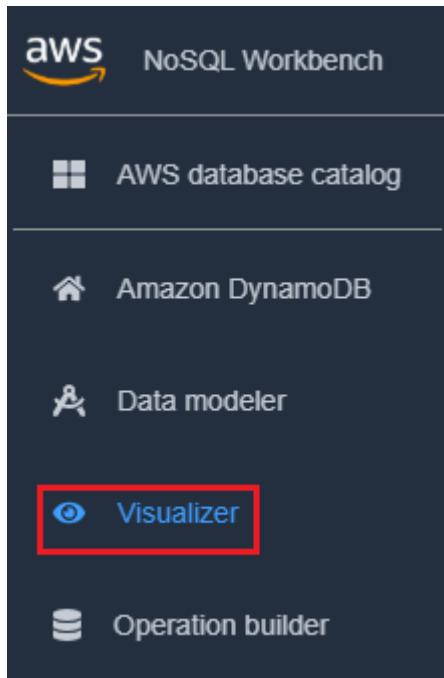
There are two ways to add sample data. One is using our sample data auto generation tool. The other is adding data one at a time.

Follow these steps to add sample data to a data model using NoSQL Workbench for Amazon DynamoDB.

To auto generate sample data

Auto generating sample data helps you generate between 1 to 5000 rows of data for immediate use. You can specify a granular sample data type to create realistic data based on your design and testing needs. To utilize the capability to generate realistic data, you need to specify the sample data type format for your attributes in the Data modeler. See [Creating a new data model](#) for specifying sample data type formats.

1. In the navigation pane on the left side, choose the **visualizer** icon.



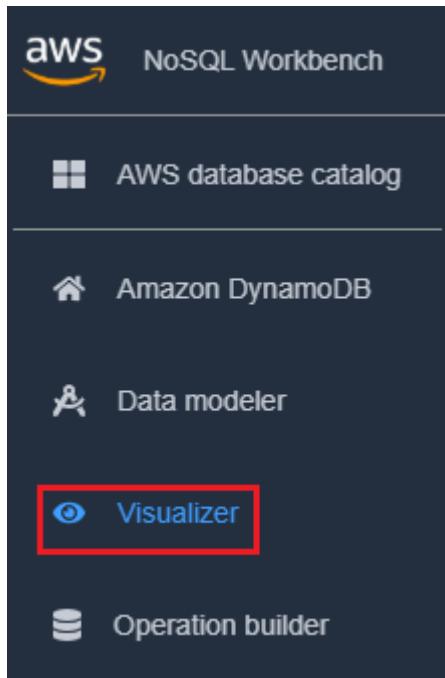
2. In the visualizer, select the data model and choose the table.
3. Choose the **Action** drop down, and select **Add sample data**.

A screenshot of the 'Songs' table in the Visualizer. The table has a primary key of 'Id'. A context menu is open at the top right of the table, listing: 'Actions', 'Configure sample data', 'Add sample data' (which is highlighted in blue), 'Edit data', and 'Import CSV file'.

4. Enter the number or items of sample data that you would like to generate, then select **Confirm**.

To add sample data one at a time

1. In the navigation pane on the left side, choose the **visualizer** icon.



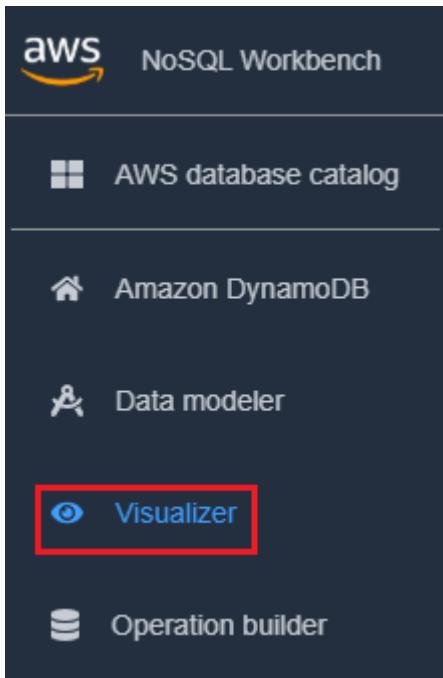
2. In the visualizer, select the data model and choose the table.
3. Choose the **Action** drop down, and select **Edit data**.

A screenshot of the AWS NoSQL Workbench Visualizer for the 'Songs' table. The table has a primary key of 'Id' and a sort key of 'Metadata'. A context menu is open over the table, with the 'Actions' dropdown showing options: 'Configure sample data', 'Add sample data', 'Edit data', and 'Import CSV file'.

4. Choose **Add new row**. Enter the sample data into the empty text boxes, and choose **Add new row** again to add additional rows. When done choose **Save changes**.

To delete sample data

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. In the visualizer, select the data model and choose the table.
3. Choose the **Action** drop down, and select **Edit data**.

A screenshot of the NoSQL Workbench Visualizer interface for the Songs table. The table has a Primary key (Partition key: Id) and Sort key: Metadata. The Attributes section shows a list of attributes. An 'Actions' dropdown menu is open on the right, listing: Configure sample data, Add sample data, Edit data, and Import CSV file.

4. Select the delete icon next to each row of data you want to delete.

Importing sample data from a CSV file

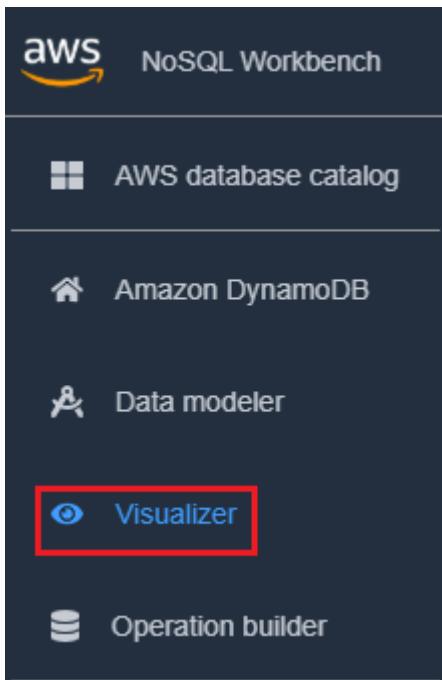
If you have preexisting sample data in a CSV file, you can import it into NoSQL Workbench. This enables you to quickly populate your model with sample data without having to enter it line by line.

The column names in the CSV file must match the attribute names in your data model, but they do not need to be in the same order. For example, if your data model has attributes called LoginAlias, FirstName, and LastName, your CSV columns could be LastName, FirstName, and LoginAlias.

Data import from a CSV file is limited to 150 rows at a time.

To import data from a CSV file into NoSQL Workbench

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. In the visualizer, select the data model and choose the table.
3. Choose the **Action** drop down, and select **Edit Data**.
4. Choose the **Action** drop down again, and select **Import CSV file**.
5. Select your CSV file and choose **Open**. The data in the CSV file will be appended to your table.

Note

If your CSV file contains one or more rows that have the same keys as items already in your table, you will have the option of overwriting the existing items or appending them to the end of the table. If you choose to append the items, the suffix "-Copy" will be added to each duplicate item's key to differentiate the duplicate items from the items that were already in the table.

Viewing data access patterns

In NoSQL Workbench, *facets* represent an application's different data access patterns for Amazon DynamoDB. Facets can help you visualize your data model when multiple data types are represented by a sort key. Facets give you a way to view a subset of the data in a table, without

having to see records that don't meet the constraints of the facet. Facets are considered a visual data modeling tool, and don't exist as a usable construct in DynamoDB, as they are purely an aid to modeling of access patterns.

To see an example of facets, you can import one of our sample data models with facets as part of the data model template.

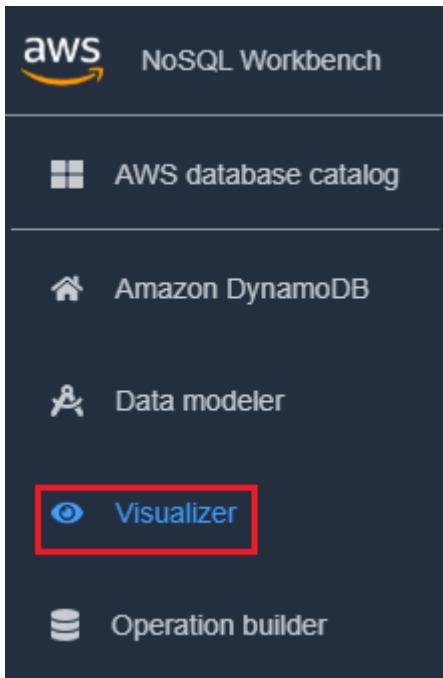
Import sample data model

1. On the left, choose **Amazon DynamoDB**.
2. In the Sample data models section, hover your pointer over Music Library Data Model and choose **Import**.

The screenshot shows the AWS NoSQL Workbench interface. On the left, there's a navigation pane with icons for NoSQL Workbench, AWS database catalog, Amazon DynamoDB (which is highlighted with a red box), Data modeler, Visualizer, Operation builder, Documentation (with a dropdown arrow), and Email us. The main area has two sections: 'Sample data models' and 'Recent data models'. The 'Sample data models' section lists several data models with their names, authors, and last modified dates. The 'Music Library Data Model' by 'Bobby' is selected and shown in more detail. An 'Import' button is highlighted with a red box at the bottom right of its row. The 'Recent data models' section shows two entries: 'Employee Data Model' by 'Amazon Web Services, Inc.' and 'Music' by 'Bobby'.

Data model name	Skill level
AWS Discussion Forum Data Model	Introductory
Bookmarks Data Model	Introductory
Employee Data Model	Introductory
Ski Resort Data Model	Introductory
Credit Card Offers Data Model	Advanced
Music Library Data Model	Advanced

3. In the navigation pane on the left side, choose the **visualizer** icon.



- Choose the Songs table to expand it. You'll be shown an aggregate view of your data.

The screenshot shows the Visualizer interface. On the left, the "Data model" dropdown is set to "Music Library Data Model". The "Songs" table is selected and highlighted with a red box. The main area displays an "Aggregate view" of the "Songs" table, specifically the "GS: DownloadsByMonth" facet. The data is presented in a table with columns: Primary key (Partition key: Id, Sort key: Metadata), Details (Title, Artist, TotalDownloads), DownloadTimestamp, and DownloadTimestamp.

Primary key	Attributes
Partition key: Id	Sort key: Metadata
Details	Title: Wild Love, Artist: Argyboots, TotalDownloads: 3
DId-9349823681	DownloadTimestamp: 2018-01-01T00:00:07
DId-9349823682	DownloadTimestamp: 2018-01-01T00:01:08

- Choose **Facets** drop-down arrow to expand the available facets.
- Choose the SongDetails facet to visualize the data with the SongDetails facet applied.

The screenshot shows the Visualizer interface with the "Facets" dropdown expanded. The "SongDetails" facet is selected and highlighted with a red box. The main area displays the "SongDetails" facet data for the "Songs" table. The data is presented in a table with columns: SongId (Partition key), Metadata (Sort key), Title, Artist, and TotalDownloads.

SongId (Partition key) : String	Metadata (Sort key) : String	Title : String	Artist : String	TotalDownloads : String
1	Details	Wild Love	Argyboots	3
2	Details	Example Song Title	Jorge Souza	4
12	ACME Album	ACME Best Song	ACME	4

You can also edit the facet definitions using the Data Modeler. For more information, see [Editing an existing data model](#).

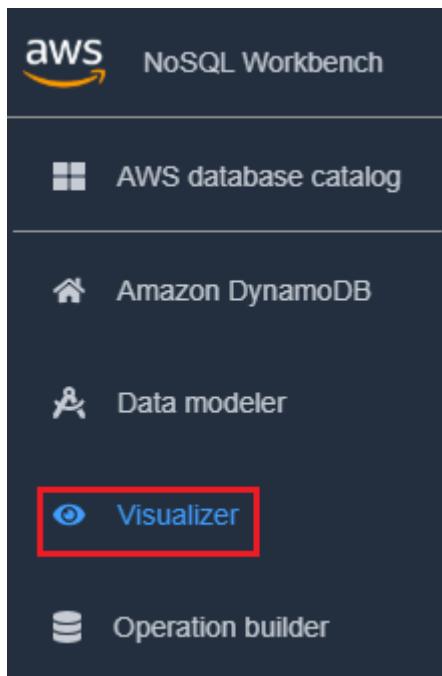
Viewing all tables in a data model using aggregate view

The aggregate view in NoSQL Workbench for Amazon DynamoDB represents all the tables in a data model. For each table, the following information appears:

- Table column names
- Sample data
- All global secondary indexes that are associated with the table. The following information is displayed for each index:
 - Index column names
 - Sample data

To view all table information

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. In the visualizer, choose **Aggregate view**.

The screenshot shows the AWS NoSQL Workbench Visualizer interface. On the left, a navigation pane lists various tools: AWS database catalog, Amazon DynamoDB (selected), Data modeler, Visualizer (selected), Operation builder, Documentation, and Share feedback. A status bar at the bottom indicates "DDB local". The main area is titled "Visualizer" and contains a "Data model" section with a dropdown menu set to "Discussion Forum". Below this are sections for "Forum", "Thread", and "Reply". Under "Aggregate view", there is a table for "Forum" with the following data:

Primary key	Attributes			
	Partition key: ForumName	Category	Threads	Messages
Amazon DynamoDB	Amazon Web Services	2	4	1000
Amazon Simple Notification Service	Category	Threads	Messages	Views
Amazon Simple Queue Service	Amazon Web Services	5	5	1200
Amazon MQ	Category	Threads	Messages	Views
Amazon EMR	Amazon Web Services	9	6	1300
AWS Data Pipeline	Category	Threads	Messages	Views
Amazon Athena	Amazon Web Services	22	7	1400
AWS Step Functions	Category	Threads	Messages	Views
	Amazon Web Services	15	8	600
	Category	Threads	Messages	Views
	Amazon Web Services	19	9	500
	Category	Threads	Messages	Views
	Amazon Web Services	43	10	55
	Category	Threads	Messages	Views
	Amazon Web Services	30	11	99

Below the "Forum" table is a "Thread" section with a table header:

Primary key	Attributes			
-------------	------------	--	--	--

At the bottom of the main area, there are buttons for "Commit to Amazon DynamoDB" and "Share feedback".

Committing a data model to DynamoDB

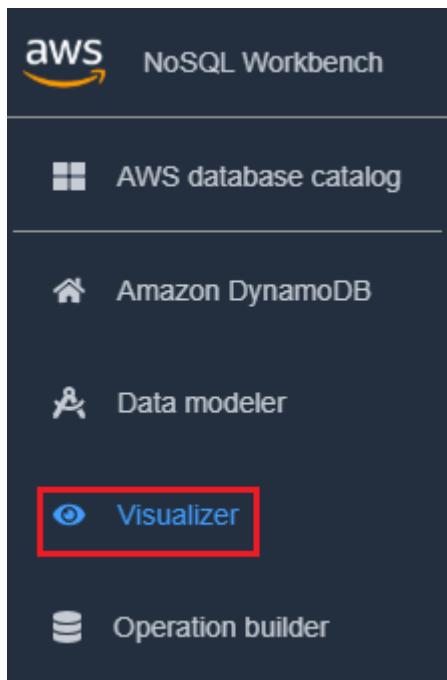
When you are satisfied with your data model, you can commit the model to Amazon DynamoDB.

Note

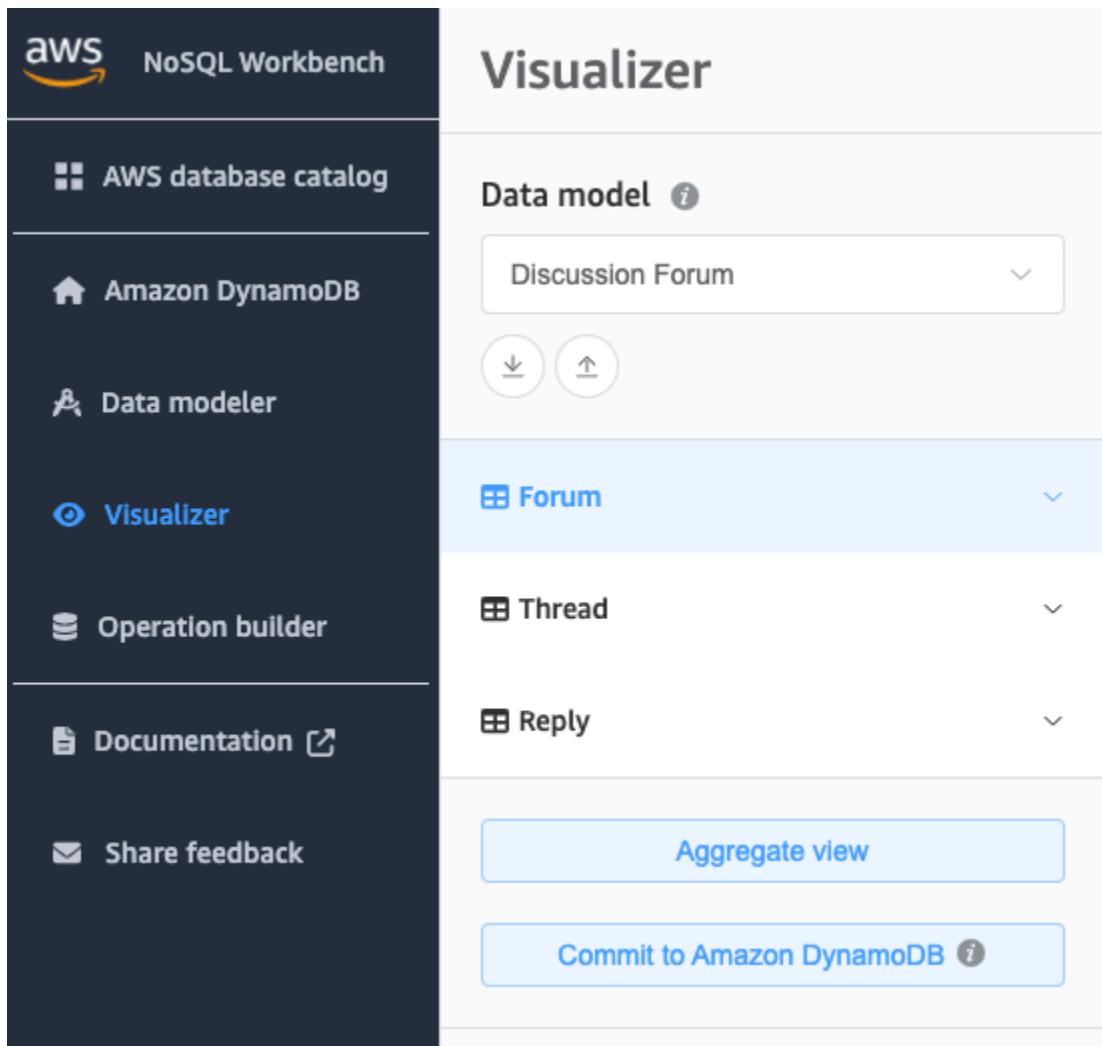
- This action results in the creation of server-side resources in AWS for the tables and global secondary indexes represented in the data model.
- Tables are created with the following characteristics:
 - Auto scaling is set to 70 percent target utilization.
 - Provisioned capacity is set to 5 read capacity units and 5 write capacity units.
 - Global secondary indexes are created with provisioned capacity of 10 read capacity units and 5 write capacity units.

To commit the data model to DynamoDB

- In the navigation pane on the left side, choose the **visualizer** icon.



2. Choose **Commit to DynamoDB**.



3. Choose an already existing connection, or create a new connection by choosing the **Add new remote connection** tab.

- To add a new connection, specify the following information:
 - **Account Alias**
 - **AWS Region**
 - **Access key ID**
 - **Secret access key**

For more information about how to obtain the access keys, see [Getting an AWS access key](#).

- You can optionally specify the following:
 - [**Session token**](#)
 - [**IAM role ARN**](#)

- If you don't want to sign up for a free tier account, and prefer to use [DynamoDB local \(downloadable version\)](#):
 1. Choose the **Add a new DynamoDB local connection** tab.
 2. Specify the **Connection name** and **Port**.

4. Choose **Commit**.

 **Note**

If you installed DynamoDB local as part of the NoSQL Workbench setup, you'll need to turn DynamoDB local on by using the **DynamoDB local Server** toggle at the bottom left of the NoSQL Workbench screen. See [Install NoSQL Workbench for DynamoDB](#) for more information on this toggle.

Exploring datasets and building operations with NoSQL Workbench

NoSQL Workbench for Amazon DynamoDB provides a rich graphical user interface for developing and testing queries. You can use the operation builder in NoSQL Workbench to view, explore, and query live datasets. The structured operation builder supports projection expression, condition expression, and generates sample code in multiple languages. You can directly clone tables from one Amazon DynamoDB account to another one in different Regions. You can also directly clone tables between DynamoDB local and an Amazon DynamoDB account for faster copying of your table's key schema (and optionally GSI schema and items) between your development environments. You can save as many as 50 DynamoDB data operations in the operation builder.

Topics

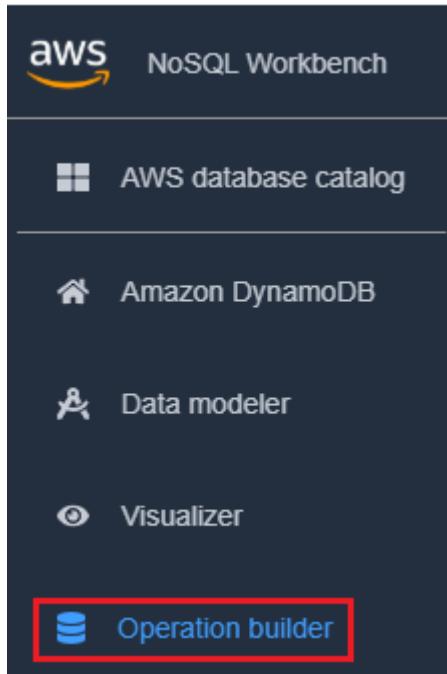
- [Connecting to live datasets](#)
- [Building complex operations](#)
- [Cloning tables with NoSQL Workbench](#)
- [Exporting data to a CSV file](#)

Connecting to live datasets

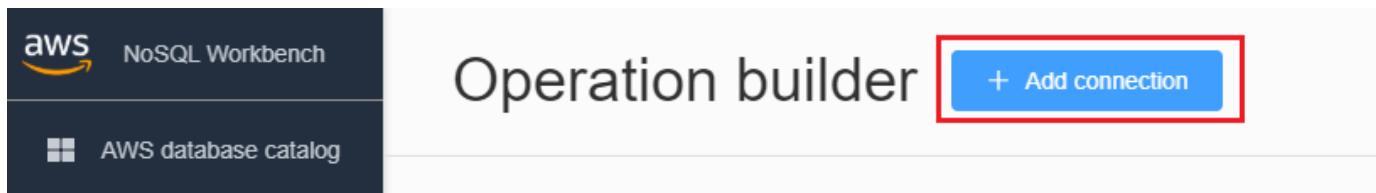
To connect to your Amazon DynamoDB tables with NoSQL Workbench, you must first connect to your AWS account.

To add a connection to your database

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Operation builder** icon.



2. Choose **Add connection**.



3. Specify the following information:

- **Account alias**
- **AWS Region**
- **Access key ID**
- **Secret access key**

For more information about how to obtain the access keys, see [Getting an AWS access key](#).

You can optionally, specify the following:

- [**Session token**](#)
- [**IAM role ARN**](#)

4. Choose **Connect**.

Add a new database connection



A remote connection lets you access the DynamoDB web service in different AWS Regions.

A DynamoDB local connection lets you access the DynamoDB local server running on your computer.

Remote

DynamoDB local

* Connection name

Connection 3

* Default AWS Region

Default AWS Region



* Access key ID

AWS Access key ID

* Secret access key

AWS secret access key

Session token

AWS session token



IAM role ARN

IAM role ARN



Persist connection



If you select this check box, AWS connection secrets will be persisted in

C:\Users\m*****\ml.aws\credentials

Cancel

Reset

Connect

If you don't want to sign up for a free tier account, and prefer to use [DynamoDB local \(downloadable version\)](#):

- a. Choose the **Local** tab on the connection screen.
- b. Specify the following information:
 - **Connection name**
 - **Port**
- c. Choose the **connect** button.

Add a new database connection

i A remote connection can let you interact with DynamoDB servers in different regions.
A local connection can let you interact with the DynamoDB Local server on your machine.

Remote **Local**

i Please setup the dynamoDB local server on your machine before adding a connection.
[Setting Up DynamoDB Local](#)

* Connection name	Connection 1
* Hostname	localhost
* Port	8000

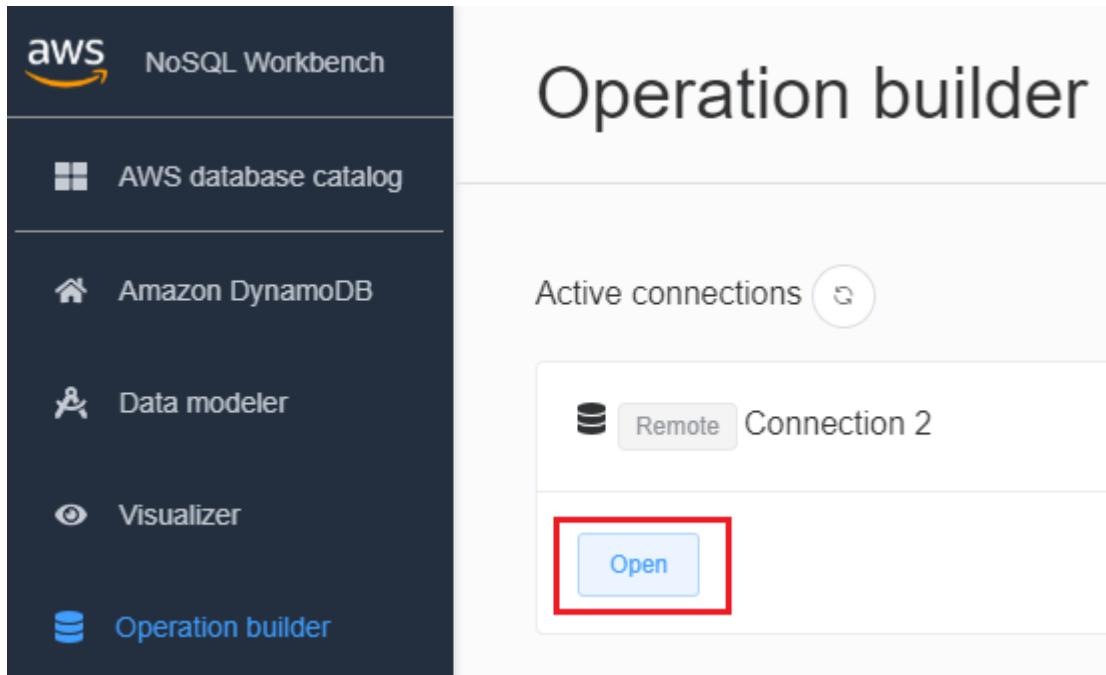
Cancel **Reset** **Connect**

i **Note**

To connect to DynamoDB local, either manually launch DynamoDB local using your terminal (see [deploying DynamoDB local on your computer](#)) or launch DynamoDB local

directly using the DDB local toggle in the NoSQL Workbench navigation menu. Ensure the connection port is the same as your DynamoDB local port.

5. On the created connection, choose **Open**.



After connecting to your DynamoDB database, the list of available tables appears in the left pane. Choose one of the tables to return a sample of the data stored in the table.

You can now run queries against the selected table.

To run queries on a table

1. In the **Attribute name** list, choose the attribute that you want to query on.
2. Specify the comparison operator.
3. Specify the data type of the value.
4. Specify the value to query for.
5. Choose **Scan**.

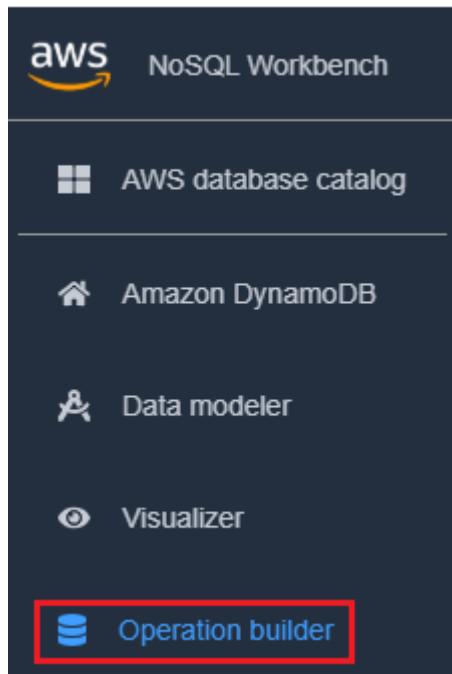
For more information about this operation, see [Scan](#) in the *Amazon DynamoDB API Reference*.

Building complex operations

The operation builder in NoSQL Workbench for Amazon DynamoDB provides a visual interface where you can perform complex data plane operations. It includes support for projection expressions and condition expressions. Once you've built an operation, you can save it for later use (up to 50 operations can be saved). You can then browse a list of your frequently used data-plane operations in the **Saved Operations** menu, and use them to automatically populate and build a new operation. You can also generate sample code for these operations, in multiple languages.

NoSQL Workbench supports building [PartiQL](#) for DynamoDB statements, which allows you to interact with DynamoDB using a SQL-compatible query language. NoSQL Workbench also supports building DynamoDB CRUD API operations.

To use NoSQL Workbench to build operations, in the navigation pane on the left side, choose the **Operation builder** icon.



Topics

- [Building PartiQL statements](#)
- [Building API operations](#)

Building PartiQL statements

To use NoSQL Workbench to build [PartiQL for DynamoDB](#) statements choose **PartiQL operations** at the top right corner of NoSQL Workbench.

You can build the following PartiQL statement types in the operation builder.

Topics

- [Singleton statements](#)
- [Transactions](#)
- [Batch](#)

Singleton statements

To run or generate code for a PartiQL statement, do the following.

● PartiQL statement ○ PartiQL transaction ○ PartiQL batch

1

Statement

```
1 SELECT *
2 FROM Music
3 WHERE Artist=? and SongTitle=?
```

Optional request parameters 3.a

Enable strongly consistent reads ⓘ

Parameters ⓘ

Attribute type	Attribute value
String	Acme Band 3.c
String	PartiQL Rocks

+ Add new parameter 3.b

5 4 6

Clear form Run Generate code Save operation

▲ Hide operation

1. Choose **PartiQL statement**.
2. Enter a valid [PartiQL statement](#).
3. If your statement uses parameters:
 - a. Choose **Optional request parameters**.
 - b. Choose **Add new parameters**.
 - c. Enter the attribute type and value.
 - d. If you want to add additional parameters, repeat steps b and c.
4. If you want to generate code, choose **Generate code**.

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

5. If you want the operation to be run immediately, choose **Run**.
6. If you want to save this operation for later use, choose **Save operation**. Then enter a name for your operation and choose **Save**.

Transactions

To run or generate code for a PartiQL transaction, do the following.

○ PartiQL statement **○ PartiQL transaction** ○ PartiQL batch
1

Delete all statements Collapse all statements Run Generate code Save operation
7 6 8

i Read and write operations are not supported in the same PartiQL transaction request. A SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements.

⊕ 1. Choose and then enter a title

Statement

```
1 SELECT * FROM Music WHERE Artist=? AND SongTitle=? 3
```

Optional request parameters 4.a

Parameters **i**

Attribute type	Attribute value
String	Acme Band 4.c
String	PartiQL Rocks 4.b

+ Add new parameter

Clear Delete

+ Add a new statement 2

▲ Hide operation

The screenshot shows the Amazon DynamoDB PartiQL transaction editor interface. At the top, there are three radio button options: 'PartiQL statement' (unchecked), 'PartiQL transaction' (checked, highlighted with a red box), and 'PartiQL batch' (unchecked). Below the radio buttons are four buttons: 'Delete all statements', 'Collapse all statements', 'Run' (highlighted with a red box), 'Generate code', and 'Save operation'. A note below the buttons states: 'Read and write operations are not supported in the same PartiQL transaction request. A SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements.' A numbered callout '1' is placed over the 'PartiQL transaction' radio button. A numbered callout '2' is placed over the '+ Add a new statement' button. A numbered callout '3' is placed over the first parameter value 'Acme Band'. A numbered callout '4.a' is placed over the 'Optional request parameters' section. A numbered callout '4.b' is placed over the second parameter value 'PartiQL Rocks'. A numbered callout '5' is placed over the 'Run' button. A numbered callout '6' is placed over the 'Generate code' button. A numbered callout '7' is placed over the 'Delete all statements' button. A numbered callout '8' is placed over the 'Save operation' button. A numbered callout '9' is placed over the 'Save operation' button.

1. Choose **PartiQL transaction**.
2. Choose **Add a new statement**.
3. Enter a valid PartiQL statement.

Note

Read and write operations are not supported in the same PartiQL transaction request. A SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements. See [Performing transactions with PartiQL for DynamoDB](#) for more details.

4. If your statement uses parameters
 - a. Choose **Optional request parameters**.
 - b. Choose **Add new parameters**.
 - c. Enter the attribute type and value.
 - d. If you want to add additional parameters, repeat steps b and c.
5. If you want to add more statements, repeat steps 2 to 4.
6. If you want to generate code, choose **Generate code**.

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

7. If you want the operation to be run immediately, choose **Run**.
8. If you want to save this operation for later use, choose **Save operation**. Then enter a name for your operation and choose **Save**.

Batch

To run or generate code for a PartiQL batch, do the following.

1 PartiQL statement PartiQL transaction PartiQL batch

Delete all statements Collapse all statements Run Generate code Save operation

7 6 8

Read and write operations are not supported in the same PartiQL batch request, which means a SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements. Write operations to the same item are not allowed. As with the BatchGetItem operation, only singleton read operations are supported. Scan and query operations are not supported.

⊕ 1. Choose and then enter a title

Statement

1 `SELECT * FROM Music WHERE Artist=? AND SongTitle=?` 3

Optional request parameters 4.a

Enable strongly consistent reads i

Parameters i

Attribute type	Attribute value
String	Acme Band 4.c
String	PartiQL Rocks

+ Add new parameter 4.b

Clear Delete

+ Add a new statement 2

▲ Hide operation

The screenshot shows the PartiQL batch editor interface. At the top, there are three radio buttons: 'PartiQL statement' (unchecked), 'PartiQL transaction' (unchecked), and 'PartiQL batch' (checked). Below the radio buttons are four buttons: 'Delete all statements', 'Collapse all statements', 'Run' (highlighted with a red box), 'Generate code', and 'Save operation'. A note below the buttons states: 'Read and write operations are not supported in the same PartiQL batch request, which means a SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements. Write operations to the same item are not allowed. As with the BatchGetItem operation, only singleton read operations are supported. Scan and query operations are not supported.' A numbered callout '1' is placed over the 'PartiQL batch' radio button. A numbered callout '2' is placed over the '+ Add a new statement' button. A numbered callout '3' is placed over the question marks in the SELECT statement. A numbered callout '4.a' is placed over the 'Optional request parameters' section. A numbered callout '4.b' is placed over the '+ Add new parameter' button. A numbered callout '4.c' is placed over the 'Attribute value' field for the first parameter. A numbered callout '5' is placed over the 'Run' button. A numbered callout '6' is placed over the 'Generate code' button. A numbered callout '7' is placed over the 'Delete all statements' button. A numbered callout '8' is placed over the 'Save operation' button.

1. Choose **PartiQL batch**.
2. Choose **Add a new statement**.

3. Enter a valid [PartiQL statement](#).

Note

Read and write operations are not supported in the same PartiQL batch request, which means a SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements. Write operations to the same item are not allowed. As with the BatchGetItem operation, only singleton read operations are supported. Scan and query operations are not supported. See [Running batch operations with PartiQL for DynamoDB](#) for more details.

4. If your statement uses parameters:

- a. Choose **Optional request parameters**.
- b. Choose **Add new parameters**.
- c. Enter the attribute type and value.
- d. If you want to add additional parameters, repeat steps b and c.

5. If you want to add more statements, repeat steps 2 to 4.

6. If you want to generate code, choose **Generate code**.

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

7. If you want the operation to be run immediately, choose **Run**.

8. If you want to save this operation for later use, choose **Save operation**. Then enter a name for your operation and choose **Save**.

Building API operations

To use NoSQL Workbench to build DynamoDB CRUD APIs, select **Interface-based operations** at the top right corner of NoSQL Workbench.

Then go to the **Operations** drop-down and choose the operation that you want to build.

The screenshot shows the AWS NoSQL Workbench interface with the 'Operation builder' selected in the sidebar. The main area displays connection settings (Name: user1, Region: us-east-1) and a search bar for tables. A dropdown menu titled 'Build operations' is open, showing a list of operations categorized into 'Control plane operations' and 'Data plane operations'. The 'UpdateTable' operation is currently selected.

Build operations ⓘ

Operations

Choose an operation

Control plane operations

- DeleteTable
- CreateTable
- UpdateTable

Data plane operations

- UpdateItem
- PutItem

Choose a table from the navigation bar

You can perform the following operations in the operation builder.

Table operations

- [Delete Table](#)
- [Create Table](#)
- [Update Table](#)

Item operations

- [Get Item](#)
- [Put Item](#)
- [Update Item](#)
- [Delete Item](#)
- [Query](#)
- [Scan](#)

- [Transact Get Items](#)
- [Transact Write Items](#)

Table operations

Delete table

To run or generate code for a Delete Table operation, do the following.

1. Select **Delete Table** from the Operations dropdown list.
2. Select the table from the **Table name** dropdown list.
3. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

4. If you want the operation to be run immediately, choose **Run**.

In the confirmation window that appears, confirm by entering the table name and selecting **Delete**.

The screenshot shows the AWS Lambda Operation builder interface. On the left, there's a sidebar titled 'Operation builder' with sections for 'Connection' (set to 'user1' and 'us-east-1'), 'Tables' (with a search bar), and a bottom navigation bar with icons for 'Building operations', 'Logs', and 'Metrics'. The main area is titled 'Build operations' with tabs for 'PartiQL operations' (selected) and 'Interface-based operations'. It has dropdowns for 'Operations' (set to 'DeleteTable') and 'Table name' (set to 'NoSQLTest1'). At the bottom right are 'Run' and 'Generate code' buttons. Below these buttons is a 'Results' section with a message: 'No result to display. Choose a table from the navigation pane to view all table data, or build an operation and view operation results.'

Results of the operation will appear in the results tab at the bottom of the screen.

For more information about this operation, see [Delete table](#) in the *Amazon DynamoDB API Reference*.

Create table

To run or generate code for a `Create Table` operation, do the following.

1. Select **Create Table** from the Operations dropdown list.
2. Enter the table name desired.
3. Create a partition key.
4. Configure the table as desired. You can create a Global Secondary Index, enable autoscaling, and other options.
5. To customize capacity settings, go to **Capacity Settings** and uncheck the box next to **Default settings**.
 - You can now select either **Provisioned** or **On-demand capacity**.

With Provisioned selected, you can set minimum and maximum read and write capacity units. You can also enable or disable auto scaling.

- If the table is currently set to OnDemand, you will be unable to specify a provisioned throughput.
 - If you switch from OnDemand to Provisioned throughput, then Autoscaling will automatically be applied to all GSIs with: min: 1, max: 10; target: 70%.
6. If you want to clear all new settings you've made and start over, select **Clear form**.
 7. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

8. If you want the operation to be run immediately, choose **Run**.

Results of the operation will appear in the results tab at the bottom of the screen.

9. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

The screenshot shows the 'Operation builder' interface for creating a new table. The 'Operations' dropdown is set to 'CreateTable'. The 'Table name' field is set to 'Music'. The 'Partition key' is defined as 'pk1' of type 'String'. There is an unchecked checkbox for 'Add sort key'. The 'Capacity settings' section has 'Default settings' checked. A note states: 'The default settings use provisioned capacity mode with auto scaling enabled on both read and write capacity.' At the bottom are buttons for 'Clear form', 'Run', 'Generate code', and 'Save operation'.

For more information about this operation, see [Create table](#) in the *Amazon DynamoDB API Reference*.

Update table

To run or generate code for an Update Table operation, do the following.

1. Select **Update Table** from the Operations dropdown list.
2. Select the desired operation from the **Update operations** dropdown list.
3. Select the table from the **Table name** dropdown list.
4. Update the selected table as desired. You can create a GSI, update provisioned throughputs and other options.

5. If you select Create GSI, Update Provisioned Throughput or Update Provisioned Throughput for GSI the Capacity settings option will appear.
6. To customize capacity settings, go to **Capacity Settings**. Then uncheck the option for either **Inherit capacity settings from base table** or **On-demand**.
 - You can now select either **Provisioned** or **On-demand capacity**.

With Provisioned selected, you can set minimum and maximum read and write capacity units. You can also enable or disable auto scaling.

- If the table is currently set to OnDemand, you will be unable to specify a provisioned throughput.
 - If you switch from OnDemand to Provisioned throughput, then Autoscaling will automatically be applied to all GSIs with: min: 1, max: 10; target: 70%.
7. If you want to clear the settings you've just entered and start over, **Clear form**.
 8. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

9. If you want the operation to be run immediately, choose **Run**.

Results of the operation will appear in the results tab at the bottom of the screen.

10. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

The screenshot shows the 'Operation builder' interface for 'Interface-based operations'. On the left sidebar, there are sections for 'Connection' (set to 'user1' and 'us-east-1'), 'Tables' (listing 'DeleteMe1' and 'NoSQLTest1'), and 'Saved operations' (empty). The main area is titled 'Build operations' and shows the configuration for an 'UpdateTable' operation on the 'NoSQLTest1' table, specifically for creating a Global Secondary Index (GSI) named 'GSI1'. The 'GSI1' configuration includes a 'Partition key' of 'pk2' and a 'Projection type' of 'ALL'. Under 'Capacity settings', there is a checked checkbox for 'Inherit capacity settings from base table', with a note explaining that global secondary indexes inherit the capacity settings of the base table unless the base table is using provisioned capacity mode. At the bottom right are buttons for 'Clear form', 'Run', 'Generate code', and 'Save operation'.

For more information about this operation, see [Update table](#) in the *Amazon DynamoDB API Reference*.

Item operations

Get item

To run or generate code for a Get Item operation, do the following.

1. Specify the partition key value.
2. If you want to add a projection expression , do the following:

- a. Choose **Projection expression**.
- b. Choose the + (plus sign) next to **Projection expression**.
- c. Specify the **Attribute name**.

The screenshot shows the 'GetItem' operation configuration page. At the top, 'Data plane operations' dropdown is set to 'GetItem' and 'Table' dropdown is set to 'Music'. Below these, there are two input fields: 'Partition key' with value 'Artist' and 'Sort key' with value 'SongTitle'. A checked checkbox labeled 'Projection expression' has a '+' button next to it, which is highlighted in blue. Below this, a 'Projected attribute' field contains 'Attribute name' with a delete icon. At the bottom right, there are four buttons: 'Clear form', 'Run' (blue), 'Generate code' (blue), and 'Save operation'.

3. If you want to generate code, choose **Generate code**.

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

4. If you want the operation to be run immediately, choose **Run**.
5. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see [GetItem](#) in the *Amazon DynamoDB API Reference*.

Put item

To run or generate code for a Put Item operation, do the following.

1. Specify the partition key value.

2. Specify the sort key value, if one exists.
3. If you want to add non-key attributes, do the following:
 - a. Choose the + (plus sign) next to **Other attributes**.
 - b. Specify the **Attribute name, Type, and Value**.
4. If a condition expression must be satisfied for the Put Item operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition expressions](#).

Data plane operations Table

PutItem Music

* Partition key Attribute value for partition key: Artist

* Sort key Attribute value for sort key: SongTitle

Other attributes + i

Attribute name	Attribute type	Attribute value
Attribute name	String	<empty>

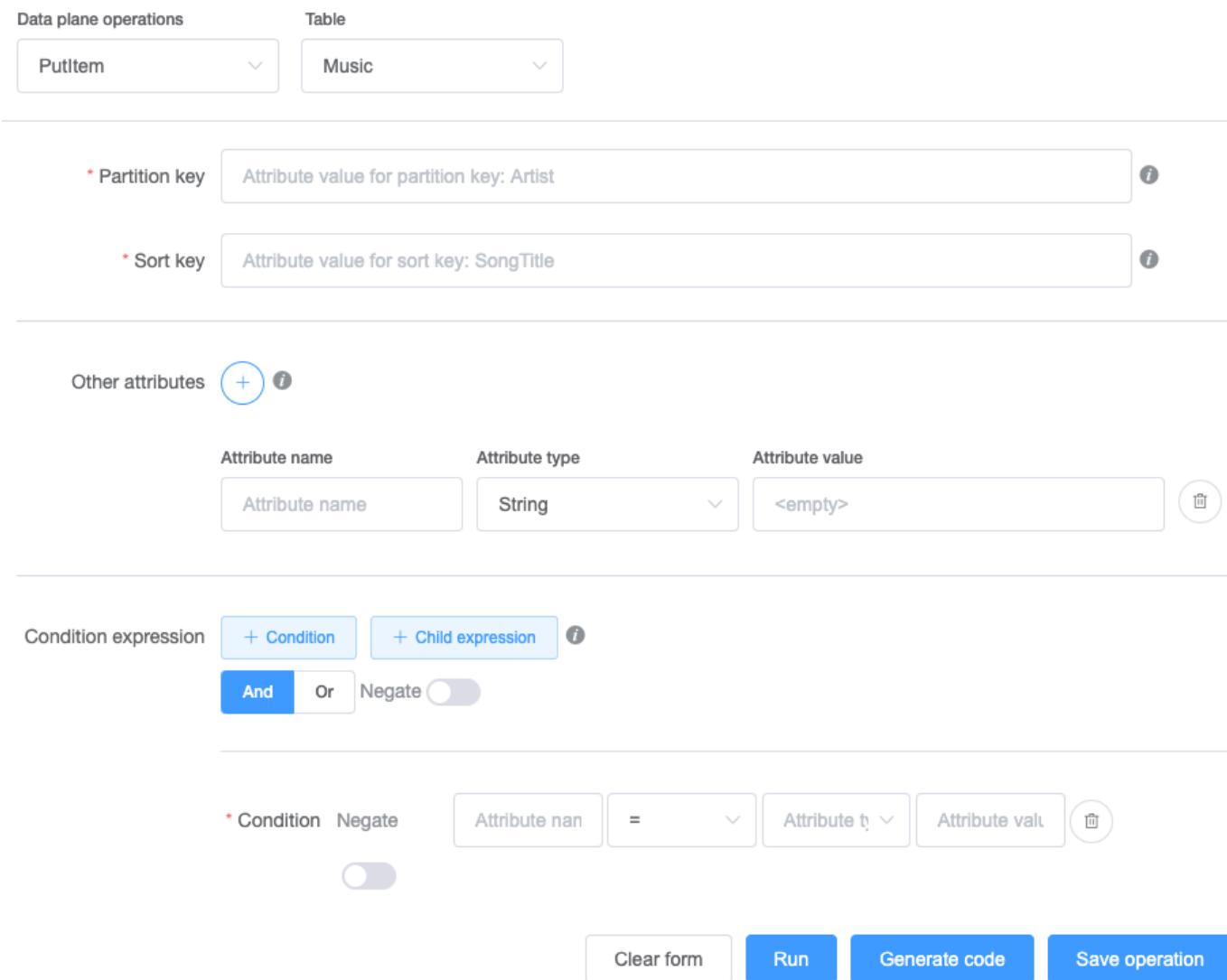
Condition expression + Condition + Child expression i

And Or Negate (checkbox)

* Condition Negate i

Attribute name	=	Attribute type	Attribute value
Attribute name	=	Attribute type	Attribute value

Clear form Run Generate code Save operation



5. If you want to generate code, choose **Generate code**.

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

6. If you want the operation to be run immediately, choose **Run**.
7. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see [PutItem](#) in the *Amazon DynamoDB API Reference*.

Update item

To run or generate code for an Update Item operation, do the following:

1. Enter the partition key value.
2. Enter the sort key value, if one exists.
3. In **Update expression**, choose the expression in the list.
4. Choose the + (plus sign) for the expression.
5. Enter the attribute name and attribute value for the selected expression.
6. If you want to add more expressions, choose another expression in the **Update Expression** drop-down list, and then select the +.
7. If a condition expression must be satisfied for the Update Item operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition expressions](#).

Data plane operations Table

UpdateItem Music

* Partition key Attribute value for partition key: Artist i

* Sort key Attribute value for sort key: SongTitle i

Update expression Select operation + i

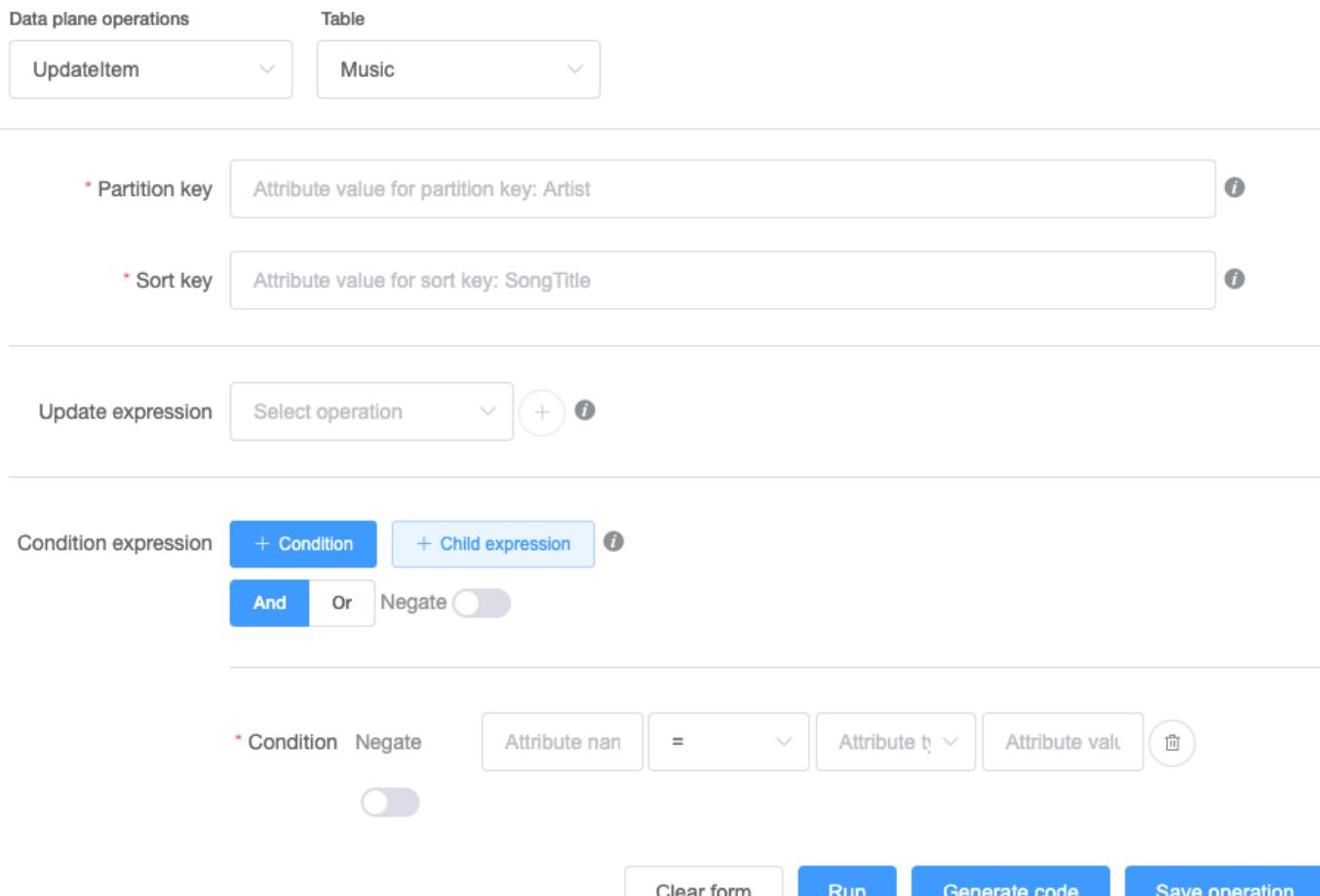
Condition expression + Condition + Child expression i

And Or Negate ()

* Condition Negate ()

Attribute name = Attribute type Attribute value

Clear form Run Generate code Save operation



8. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

9. If you want the operation to be run immediately, choose **Run**.

10. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see [UpdateItem](#) in the *Amazon DynamoDB API Reference*.

Delete item

To run or generate code for a Delete Item operation, do the following.

1. Enter the partition key value.

2. Enter the sort key value, if one exists.
3. If a condition expression must be satisfied for the Delete Item operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition expressions](#).

The screenshot shows the 'DeleteItem' form in the Amazon DynamoDB Data Plane Operations interface. At the top, there are dropdown menus for 'Data plane operations' (set to 'DeleteItem') and 'Table' (set to 'Music'). Below these are fields for specifying the partition key ('Artist') and sort key ('SongTitle'). Under the 'Condition expression' section, there are buttons for '+ Condition' and '+ Child expression'. A 'And' button is selected. There is also a 'Negate' toggle switch. Below this, there are fields for 'Attribute name' (with a dropdown menu), 'Comparison operator' (set to '='), 'Attribute type' (dropdown menu), and 'Attribute value' (input field). At the bottom right are buttons for 'Clear form', 'Run', 'Generate code', and 'Save operation'.

4. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

5. If you want the operation to be run immediately, choose **Run**.
6. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see [DeleteItem](#) in the *Amazon DynamoDB API Reference*.

Query

To run or generate code for a Query operation, do the following.

1. Specify the partition key value.
2. If a sort key is needed for the Query operation:
 - a. Select **Sort key**.
 - b. Specify the comparison operator, attribute type, and attribute value.
3. If not all the attributes should be returned with the operation result, select **Projection expression**.
4. Choose the + (plus sign).
5. Enter the attribute to return with the query result.
6. If more attributes are needed, choose the + .
7. If a condition expression must be satisfied for the Query operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition expressions](#).

Data plane operations Table

Query Music

* Partition key Attribute value for partition key: Artist i

Sort key i

Projection expression i +

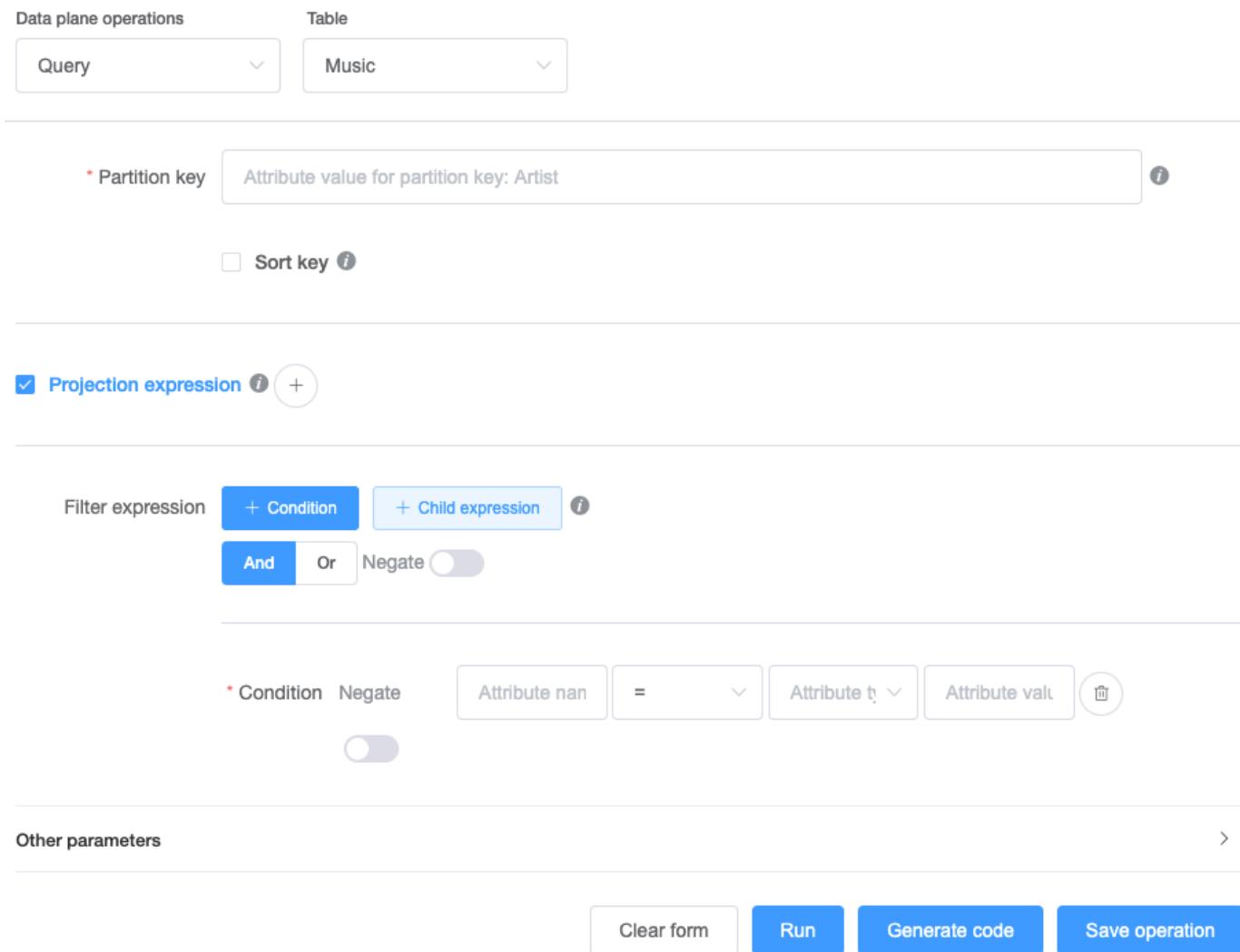
Filter expression + Condition + Child expression i

And Or Negate toggle

* Condition Negate toggle Attribute name = Attribute type i Attribute value trash

Other parameters >

Clear form Run Generate code Save operation



8. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

9. If you want the operation to be run immediately, choose **Run**.

10. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see [Query](#) in the *Amazon DynamoDB API Reference*.

Scan

To run or generate code for a Scan operation, do the following.

1. If not all the attributes should be returned with the operation result, select **Projection expression**.
2. Choose the + (plus sign).
3. Specify the attribute to return with the query result.
4. If more attributes are needed, choose the + again.
5. If a condition expression must be satisfied for the scan operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition expressions](#).

6. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

7. If you want the operation to be run immediately, choose **Run**.
8. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

TransactGetItems

To run or generate code for a TransactGetItems operation, do the following.

1. Choose the + (plus sign).
2. Follow the instructions for the [Get item](#) operation.

When you are done specifying the details of the operation, choose the + (plus sign) if you want to add more operations.

The screenshot shows the Amazon DynamoDB Data plane operations interface. At the top, there's a dropdown menu labeled "TransactGetItems". Below it, a sidebar lists actions: "TransactGetItem" (selected), "1. GetItem", and "1. GetItem" again. The main area is titled "1. GetItem". It has fields for "Table Name" (set to "Music"), "Partition key" (set to "Attribute value for partition key: Artist"), and "Sort key" (set to "Attribute value for sort key: SongTitle"). There's also a checked checkbox for "Projection expression" with a plus sign icon, and a field for "Projected attribute" with a placeholder "Attribute name" and a delete icon. At the bottom, there are three buttons: "Run", "Generate code", and "Save operation".

To change the order of actions, choose an action in the list on the left side, and then choose the up or down arrows to move it up or down in the list.

To delete an action, choose the action in the list, and then choose the **Delete** (trash can) icon.

3. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

4. If you want the operation to be run immediately, choose **Run**.
5. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about transactions, see [Amazon DynamoDB transactions](#).

TransactWriteItems

To run or generate code for a TransactWriteItems operation, do the following.

1. In the **Actions** drop-down list, choose the operation that you want.

- For DeleteItem, follow the instructions for the [Delete item](#) operation.
- For PutItem, follow the instructions for the [Put item](#) operation.
- For UpdateItem, follow the instructions for the [Update item](#) operation.

When you are done specifying the details of the operation, choose the **+** button.

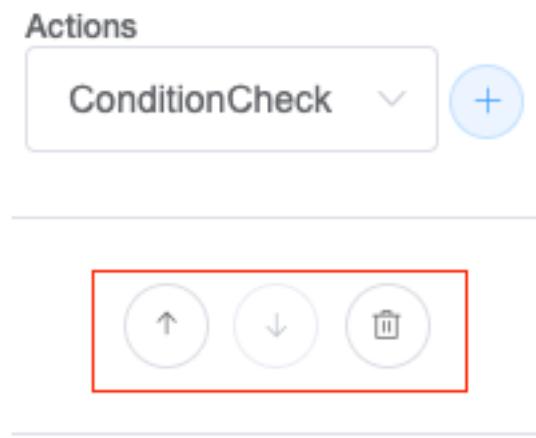
The screenshot shows the AWS Lambda Function Builder interface. On the left, there's a sidebar titled "Data plane operations" with a dropdown menu set to "TransactWriteItems". Below it is a list of actions: "DeleteItem", "PutItem" (which is selected and highlighted in blue), "UpdateItem", and "ConditionCheck". Under "PutItem", there's a sub-section titled "1. PutItem". The main workspace is titled "PutItem" and contains the following fields:

- "Table name": "Music"
- "* Partition key": "Attribute value for partition key: Artist" (with an info icon)
- "* Sort key": "Attribute value for sort key: SongTitle" (with an info icon)
- "Other attributes": A section with a "+" button and an info icon.
- "Condition expression": Buttons for "+ Condition" and "+ Child expression" (both with info icons).

At the bottom right of the workspace are three buttons: "Run", "Generate code", and "Save operation".

To change the order of actions, choose an action in the list on the left side, and then choose the up or down arrows to move it up or down in the list.

To delete an action, choose the action in the list, and then choose the **Delete** (trash can) icon.



1. DeleteItem

2. PutItem

3. ConditionCheck

2. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

3. If you want the operation to be run immediately, choose **Run**.
4. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about transactions, see [Amazon DynamoDB transactions](#).

Cloning tables with NoSQL Workbench

Cloning tables will copy a table's key schema (and optionally GSI schema and items) between your development environments. You can clone a table between DynamoDB local to an Amazon

DynamoDB account, and even clone a table from one account to another in different Regions for faster experimentation.

To clone a table

1. In the **Operation Builder**, select your connection and Region (Region selection is not available for DynamoDB local).
2. Once you are connected to DynamoDB, browse your tables and select the table you want to clone.
3. Select the **Clone** button.

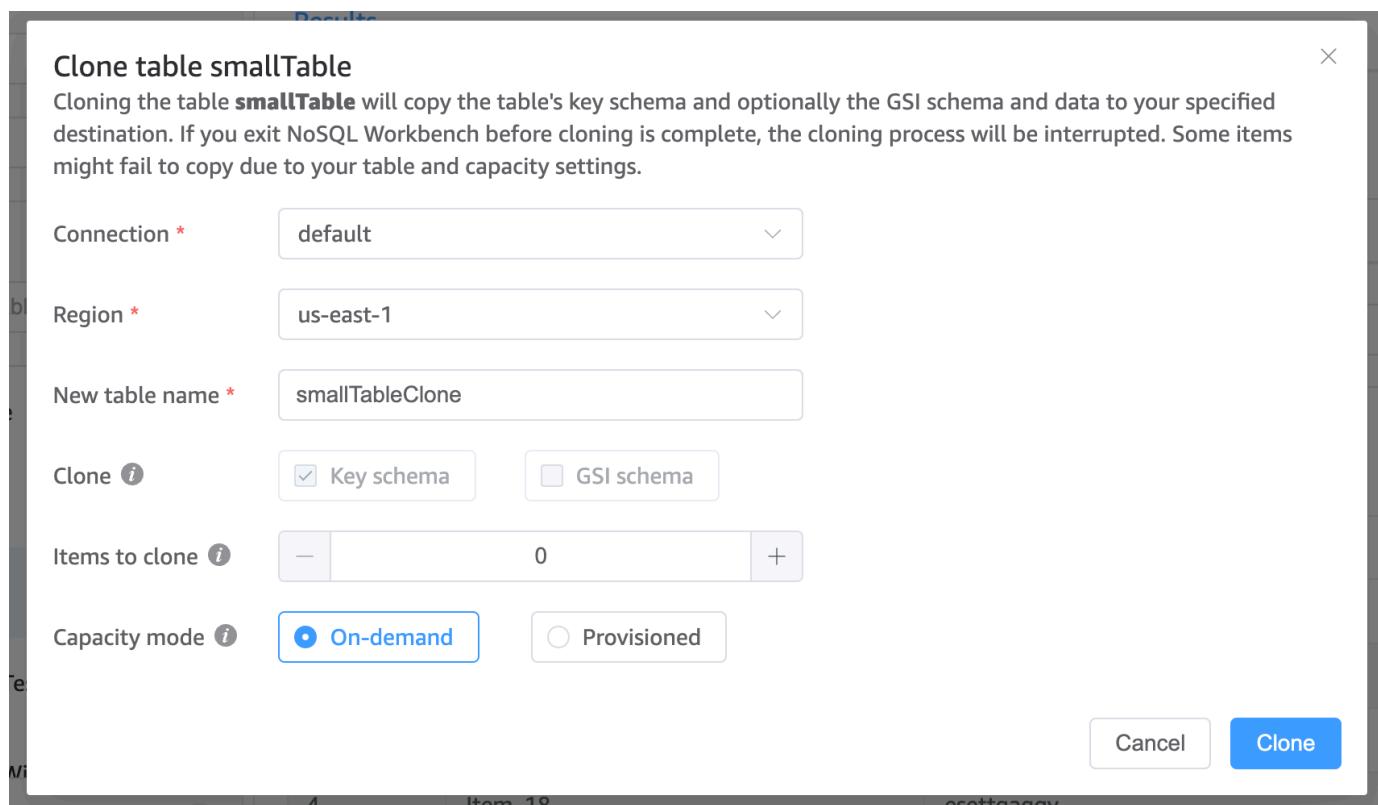
Items returned (15)
Consumed 1 RCU.

[Clone](#) [Export to CSV](#)

#	PrimaryKey ⓘ	Attribute1	Attribute2
1	Item_39	ggdqztcaw	61
2	Item_32	wljiouytoj	42
3	Item_3	aniqhpznsb	86
4	Item_18	esettqagqv	67

4. Input your clone destination details:
 - a. Select a connection.
 - b. Select a Region (Region is not available for DynamoDB local).
 - c. Enter a new table name.
 - d. Choose a clone options:
 - i. **Key schema** is selected by default and cannot be unselected. By default, cloning a table will copy your primary key and sort key if they are available.
 - ii. **GSI schema** is selected by default if your table to be cloned has a GSI. Cloning a table will copy your GSI primary key and sort key if they are available. You have the option to deselect GSI schema to skip cloning the GSI schema. Cloning a table will copy your base table's capacity settings as the GSI's capacity settings. You can use the `UpdateTable` operation in Operation Builder to update the table's GSI capacity setting after cloning is complete.

5. Enter the number of items to clone. To only clone the key schema and optionally the GSI schema, you can keep the **Items to clone** value at 0. The maximum number of items that can be cloned is 5000.
6. Choose a capacity mode:
 - a. **On-demand mode** is selected by default. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use. To learn more, see [DynamoDB On-demand mode](#).
 - b. **Provisioned mode** lets you specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes. To learn more, see [DynamoDB Provisioned mode](#).
7. Select **Clone** to begin cloning.



8. The cloning process will run in the background. The **Operation builder** tab will show a notification when there is a change in the cloning table status. You can access this status by selecting the **Operation builder** tab and then selecting the arrow button. The arrow button is located on the cloning table status widget located near the bottom of the menu sidebar.

The screenshot shows the Amazon DynamoDB Operation Builder interface. At the top, a progress bar indicates a cloning operation is in progress. Below it, a modal window displays the completion of a cloning job from a table named **smallTable** to a new table named **smallTableTest**. The summary shows 50 items read, 50 items cloned successfully, 0 items cloned failed, and 0 items cloned in progress. RCU and WRU consumed were 0.5 RCU / 50 WCU. There are "Close" and "See new table" buttons at the bottom of the modal.

Cloning table

... In progress

Clone table job completed

Table **smallTable** was successfully cloned to **smallTableTest** in the connection **default** and region **us-east-1**

Summary

Number of items read	50
Items cloned successfully	50
Items cloned failed	0
Items clone in progress	0
RCU and WRU consumed	0.5 RCU / 50 WCU

Close See new table

Exporting data to a CSV file

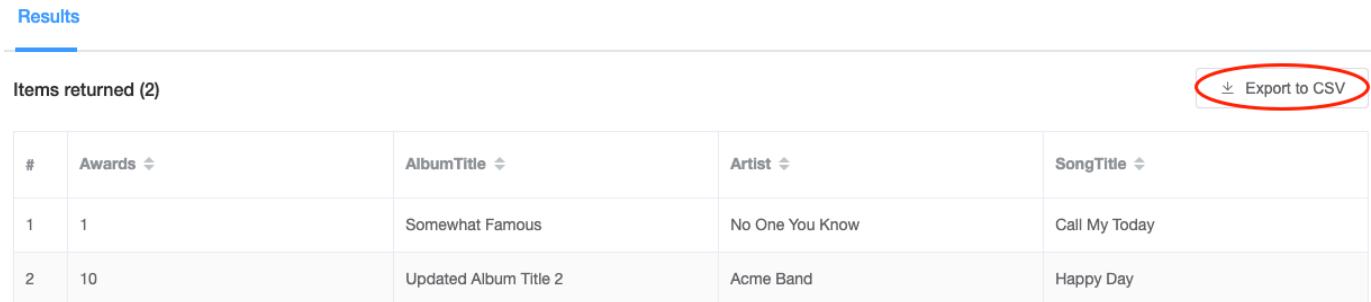
You can export the results of a query from Operation Builder to a CSV file. This enables you to load the data into a spreadsheet or process it using your preferred programming language.

Exporting to CSV

1. In the Operation Builder, run an operation of your choice, such as a Scan or Query.

Note

- You can only export results from read API operations and PartiQL statements to a CSV file. You can't export results from transaction read statements.
- Currently, you can export results one page at a time to a CSV file. If there are multiple pages of results, you must export each page individually.

2. In the Results panel, choose Export to CSV.

Items returned (2)				
#	Awards	AlbumTitle	Artist	SongTitle
1	1	Somewhat Famous	No One You Know	Call My Today
2	10	Updated Album Title 2	Acme Band	Happy Day

3. Choose a filename and location for your CSV file and select Save.

Sample data models for NoSQL Workbench

The home page for the modeler and visualizer display a number of sample models that ship with the NoSQL Workbench. This section describes these models and their potential uses.

Topics

- [Employee data model](#)
- [Discussion forum data model](#)
- [Music library data model](#)
- [Ski resort data model](#)
- [Credit card offers data model](#)
- [Bookmarks data model](#)

Employee data model

This data model is an introductory model. It represents an employee's basic details such as a unique alias, first name, last name, designation, manager, and skills.

This data model depicts a few techniques such as handling complex attribute such as having more than one skill. This model is also an example of one-to-many relationship through the manager and their reporting employees that has been achieved by the secondary index DirectReports.

The access patterns facilitated by this data model are:

- Retrieval of an employee record using the employee's login alias, facilitated by a table called Employee.
- Search for employees by name, facilitated by the Employee table's global secondary index called Name.
- Retrieval of all direct reports of a manager using the manager's login alias, facilitated by the Employee table's global secondary index called DirectReports.

Discussion forum data model

This data model represents a discussion forums. Using this model customers can engage with the developer community, ask questions, and respond to other customers' posts. Each AWS service has a dedicated forum. Anyone can start a new discussion thread by posting a message in a forum, and each thread receives any number of replies.

The access patterns facilitated by this data model are:

- Retrieval of a forum record using the forum's name, facilitated by a table called Forum.
- Retrieval of a specific thread or all threads for a forum, facilitated by a table called Thread.
- Search for replies using the posting user's email address, facilitated by the Reply table's global secondary index called PostedBy-Message-Index.

Music library data model

This data model represents a music library that has a large collection of songs and showcases its most downloaded songs in near-real time.

The access patterns facilitated by this data model are:

- Retrieval of a song record, facilitated by a table called Songs.
- Retrieval of a specific download record or all download records for a song, facilitated by a table called Songs.

- Retrieval of a specific monthly download count record or all monthly download count records for a song, facilitated by a table called Song.
- Retrieval of all records (including song record, download records, and monthly download count records) for a song, facilitated by a table called Songs.
- Search for most downloaded songs, facilitated by the Songs table's global secondary index called DownloadsByMonth.

Ski resort data model

This data model represents a ski resort that has a large collection of data for each ski lift collected daily.

The access patterns facilitated by this data model are:

- Retrieval of all data for a given ski lift or overall resort, dynamic and static, facilitated by a table called SkiLifts.
- Retrieval of all dynamic data (including unique lift riders, snow coverage, avalanche danger, and lift status) for a ski lift or the overall resort on a specific date, facilitated by a table called SkiLifts.
- Retrieval of all static data (including if the lift is for experienced riders only, vertical feet the lift rises, and lift riding time) for a specific ski lift, facilitated by a table called SkiLifts.
- Retrieval of date of data recorded for a specific ski lift or the overall resort sorted by total unique riders, facilitated by the SkiLifts table's global secondary index called SkiLiftsByRiders.

Credit card offers data model

This data model is used by a Credit Card Offers Application.

A credit card provider produces offers over time. These offers include balance transfers without fees, increased credit limits, lower interest rates, cash back, and airline miles. After a customer accepts or declines these offers, the respective offer status is updated accordingly.

The access patterns facilitated by this data model are:

- Retrieval of account records using AccountId, as facilitated by the main table.
- Retrieval of all the accounts with few projected items, as facilitated by the secondary index AccountIndex.

- Retrieval of accounts and all the offer records associated with those accounts by using AccountId, as facilitated by the main table.
- Retrieval of accounts and specific offer records associated with those accounts by using AccountId and OfferId, as facilitated by the main table.
- Retrieval of all ACCEPTED/DECLINED offer records of specific OfferType associated with accounts using AccountId, OfferType, and Status, as facilitated by the secondary index GSI1.
- Retrieval of offers and associated offer item records using OfferId, as facilitated by the main table.

Bookmarks data model

This data model is used store bookmarks for customers.

A customer can have many bookmarks and a bookmark can belong to many customers. This data model represents a many-to-many relationship.

The access patterns facilitated by this data model are:

- A single query by customerId can now return customer data as well as bookmarks.
- A query ByEmail index returns customer data by email address. Note that bookmarks are not retrieved by this index.
- A queryByUrl index gets bookmarks data by URL. Note that we have customerId as the sort key for the index because the same URL can be bookmarked by multiple customers.
- A query ByCustomerFolder index gets bookmarks by folder for each customer.

Release history for NoSQL Workbench

The following table describes the important changes in each release of the *NoSQL Workbench* client tool.

Version	Change	Description	Date
3.12.0	Cloning tables with NoSQL Workbench	You can now clone tables between DynamoDB local and	February 26, 2024

Version	Change	Description	Date
	and returning capacity consumed	a DynamoDB web service account or between DynamoDB web service accounts for faster development iterations. View RCU or WCU consumed after running an operation using the Operations Builder. We fixed the overwrite data issue when importing from a CSV file.	
3.11.0	DynamoDB local improvements	You can now specify port when launching the built-in DynamoDB local instance. NoSQL Workbench can now be installed on Windows without admin rights. We have updated the data model templates.	January 17, 2024

Version	Change	Description	Date
3.10.0	Native support for Apple silicon	NoSQL Workbench now includes native support for Mac with Apple silicon. You can now configure sample data generation format for attributes of type Number.	December 5, 2023
3.9.0	Data modeler improvements	Visualizer now supports committing data models to DynamoDB local with the option to overwrite existing tables.	November 3, 2023
3.8.0	Sample data generation	NoSQL Workbench now supports generating sample data for your DynamoDB data models.	September 25, 2023
3.6.0	Improvements in the Operations builder	Connections management improvements in the Operations builder. Attribute names in Data Modeler can now be changed without deleting data. Other bug fixes.	April 11, 2023

Version	Change	Description	Date
3.5.0	Support for new AWS Regions	NoSQL Workbench now supports the ap-south-2, ap-southeast-3, ap-southeast-4, eu-central-2, eu-south-2, me-central-1, and me-west-1 regions.	February 23, 2023
3.4.0	Support for DynamoDB local	NoSQL Workbench now supports installing DynamoDB local as part of the installation process.	December 6, 2022
3.3.0	Support for control plane operations	Operation Builder now supports control plane operations.	June 1, 2022
3.2.0	CSV import and export	You can now import sample data from a CSV file in the Visualizer tool, and also export the results of an Operation Builder query to a CSV file.	October 11, 2021
3.1.0	Save operations	The Operation Builder in NoSQL Workbench now supports saving operations for later use.	July 12, 2021

Version	Change	Description	Date
3.0.0	Capacity settings and CloudFormation import/export	NoSQL Workbench for Amazon DynamoDB now supports specifying a read/write capacity mode for tables, and can now import and export data models in AWS CloudFormation format.	April 21, 2021
2.2.0	Support for PartiQL	NoSQL Workbench for Amazon DynamoDB adds support for building PartiQL statements for DynamoDB.	December 4, 2020
1.1.0	Support for Linux.	NoSQL Workbench for Amazon DynamoDB is supported on Linux—Ubuntu, Fedora, and Debian.	May 4, 2020
1.0.0	NoSQL Workbench for Amazon DynamoDB – GA.	NoSQL Workbench for Amazon DynamoDB is generally available.	March 2, 2020

Version	Change	Description	Date
0.4.1	Support for IAM roles and temporary security credentials.	NoSQL Workbench for Amazon DynamoDB adds support for AWS Identity and Access Management (IAM) roles and temporary security credentials.	December 19, 2019
0.3.1	Support for DynamoDB local (Downloadable Version) .	The NoSQL Workbench now supports connecting to DynamoDB local (Downloadable Version) to design, create, query, and manage DynamoDB tables.	November 8, 2019
0.2.1	NoSQL Workbench preview released.	This is the initial release of NoSQL Workbench for DynamoDB. Use NoSQL Workbench to design, create, query, and manage DynamoDB tables.	September 16, 2019

Code examples for DynamoDB using AWS SDKs

The following code examples show how to use DynamoDB with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Cross-service examples are sample applications that work across multiple AWS services.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello DynamoDB

The following code examples show how to get started using DynamoDB.

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace DynamoDB_Actions;

public static class HelloDynamoDB
```

```
{  
    static async Task Main(string[] args)  
    {  
        var dynamoDbClient = new AmazonDynamoDBClient();  
  
        Console.WriteLine($"Hello Amazon Dynamo DB! Following are some of your  
tables:");  
        Console.WriteLine();  
  
        // You can use await and any of the async methods to get a response.  
        // Let's get the first five tables.  
        var response = await dynamoDbClient.ListTablesAsync(  
            new ListTablesRequest()  
            {  
                Limit = 5  
            });  
  
        foreach (var table in response.TableNames)  
        {  
            Console.WriteLine($"\\tTable: {table}");  
            Console.WriteLine();  
        }  
    }  
}
```

- For API details, see [ListTables](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Code for the CMakeLists.txt CMake file.

```
# Set the minimum required version of CMake for this project.
```

```
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS dynamodb)

# Set this project's name.
project("hello_dynamodb")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
"${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD)
    # Copy relevant AWS SDK for C++ libraries into the current binary directory
    # for running and debugging.

    # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
    # may need to uncomment this
    # and set the proper subdirectory to the
    # executables' location.

    AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
    hello_dynamodb.cpp)

target_link_libraries(${PROJECT_NAME}
    ${AWSSDK_LINK_LIBRARIES})
```

Code for the hello_dynamodb.cpp source file.

```
#include <aws/core/Aws.h>
#include <aws/dynamodb/DynamoDBClient.h>
#include <aws/dynamodb/model/ListTablesRequest.h>
#include <iostream>

/*
 * A "Hello DynamoDB" starter application which initializes an Amazon DynamoDB
(DynamoDB) client and lists the
 * DynamoDB tables.
 *
 * main function
 *
 * Usage: 'hello_dynamodb'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//    options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.

    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::DynamoDB::DynamoDBClient dynamodbClient(clientConfig);
        Aws::DynamoDB::Model::ListTablesRequest listTablesRequest;
        listTablesRequest.SetLimit(50);
        do {
            const Aws::DynamoDB::Model::ListTablesOutcome &outcome =
dynamodbClient.ListTables(
                listTablesRequest);
            if (!outcome.IsSuccess()) {
                std::cout << "Error: " << outcome.GetError().GetMessage() <<
std::endl;
                result = 1;
            }
        } while (!outcome.IsSuccess());
    }
}
```

```
        break;
    }

    for (const auto &tableName: outcome.GetResult().GetTableNames()) {
        std::cout << tableName << std::endl;
    }

    listTablesRequest.SetExclusiveStartTableName(
        outcome.GetResult().GetLastEvaluatedTableName());
}

} while (!listTablesRequest.GetExclusiveStartTableName().empty());
}

Aws::ShutdownAPI(options); // Should only be called once.
return result;
}
```

- For API details, see [ListTables](#) in *AWS SDK for C++ API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
```

```
* For more information, see the following documentation topic:  
*  
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html  
*/  
public class ListTables {  
    public static void main(String[] args) {  
        System.out.println("Listing your Amazon DynamoDB tables:\n");  
        Region region = Region.US_EAST_1;  
        DynamoDbClient ddb = DynamoDbClient.builder()  
            .region(region)  
            .build();  
        listAllTables(ddb);  
        ddb.close();  
    }  
  
    public static void listAllTables(DynamoDbClient ddb) {  
        boolean moreTables = true;  
        String lastName = null;  
  
        while (moreTables) {  
            try {  
                ListTablesResponse response = null;  
                if (lastName == null) {  
                    ListTablesRequest request =  
ListTablesRequest.builder().build();  
                    response = ddb.listTables(request);  
                } else {  
                    ListTablesRequest request = ListTablesRequest.builder()  
                        .exclusiveStartTableName(lastName).build();  
                    response = ddb.listTables(request);  
                }  
  
                List<String> tableNames = response.tableNames();  
                if (tableNames.size() > 0) {  
                    for (String curName : tableNames) {  
                        System.out.format("* %s\n", curName);  
                    }  
                } else {  
                    System.out.println("No tables found!");  
                    System.exit(0);  
                }  
  
                lastName = response.lastEvaluatedTableName();  
            } catch (Exception e) {  
                System.out.println("An error occurred: " + e.getMessage());  
            }  
        }  
    }  
}
```

```
        if (lastName == null) {
            moreTables = false;
        }

    } catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}

System.out.println("\nDone!");
}
```

- For API details, see [ListTables](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
    const command = new ListTablesCommand({});

    const response = await client.send(command);
    console.log(response.TableNames.join("\n"));
    return response;
};
```

- For API details, see [ListTables](#) in *AWS SDK for JavaScript API Reference*.

Code examples

- [Actions for DynamoDB using AWS SDKs](#)

- [Create a DynamoDB table using an AWS SDK](#)
- [Delete a DynamoDB table using an AWS SDK](#)
- [Delete an item from a DynamoDB table using an AWS SDK](#)
- [Get a batch of DynamoDB items using an AWS SDK](#)
- [Get an item from a DynamoDB table using an AWS SDK](#)
- [Get information about a DynamoDB table](#)
- [List DynamoDB tables using an AWS SDK](#)
- [Put an item in a DynamoDB table using an AWS SDK](#)
- [Query a DynamoDB table using an AWS SDK](#)
- [Run a PartiQL statement on a DynamoDB table using an AWS SDK](#)
- [Run batches of PartiQL statements on a DynamoDB table using an AWS SDK](#)
- [Scan a DynamoDB table using an AWS SDK](#)
- [Update an item in a DynamoDB table using an AWS SDK](#)
- [Write a batch of DynamoDB items using an AWS SDK](#)

- [Scenarios for DynamoDB using AWS SDKs](#)

- [Accelerate DynamoDB reads with DAX using an AWS SDK](#)
- [Get started with DynamoDB tables, items, and queries using an AWS SDK](#)
- [Query a DynamoDB table by using batches of PartiQL statements and an AWS SDK](#)
- [Query a DynamoDB table using PartiQL and an AWS SDK](#)
- [Use a document model for DynamoDB using an AWS SDK](#)
- [Use a high-level object persistence model for DynamoDB using an AWS SDK](#)

- [Cross-service examples for DynamoDB using AWS SDKs](#)

- [Build an application to submit data to a DynamoDB table](#)
- [Create an API Gateway REST API to track COVID-19 data](#)
- [Create a messenger application with Step Functions](#)
- [Create a photo asset management application that lets users manage photos using labels](#)
- [Create a web application to track DynamoDB data](#)
- [Create a websocket chat application with API Gateway](#)

- [Detect PPE in images with Amazon Rekognition using an AWS SDK](#)
- [Invoke a Lambda function from a browser](#)
- [Save EXIF and other image information using an AWS SDK](#)
- [Use API Gateway to invoke a Lambda function](#)
- [Use Step Functions to invoke Lambda functions](#)
- [Use scheduled events to invoke a Lambda function](#)

Actions for DynamoDB using AWS SDKs

The following code examples demonstrate how to perform individual DynamoDB actions with AWS SDKs. These excerpts call the DynamoDB API and are code excerpts from larger programs that must be run in context. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the [Amazon DynamoDB API Reference](#).

Examples

- [Create a DynamoDB table using an AWS SDK](#)
- [Delete a DynamoDB table using an AWS SDK](#)
- [Delete an item from a DynamoDB table using an AWS SDK](#)
- [Get a batch of DynamoDB items using an AWS SDK](#)
- [Get an item from a DynamoDB table using an AWS SDK](#)
- [Get information about a DynamoDB table](#)
- [List DynamoDB tables using an AWS SDK](#)
- [Put an item in a DynamoDB table using an AWS SDK](#)
- [Query a DynamoDB table using an AWS SDK](#)
- [Run a PartiQL statement on a DynamoDB table using an AWS SDK](#)
- [Run batches of PartiQL statements on a DynamoDB table using an AWS SDK](#)
- [Scan a DynamoDB table using an AWS SDK](#)
- [Update an item in a DynamoDB table using an AWS SDK](#)
- [Write a batch of DynamoDB items using an AWS SDK](#)

Create a DynamoDB table using an AWS SDK

The following code examples show how to create a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- [Accelerate reads with DAX](#)
- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Creates a new Amazon DynamoDB table and then waits for the new
/// table to become active.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</
param>
/// <param name="tableName">The name of the table to create.</param>
/// <returns>A Boolean value indicating the success of the operation.</
returns>
public static async Task<bool> CreateMovieTableAsync(AmazonDynamoDBClient
client, string tableName)
{
    var response = await client.CreateTableAsync(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "title",

```

```
        AttributeType = ScalarAttributeType.S,
    },
    new AttributeDefinition
    {
        AttributeName = "year",
        AttributeType = ScalarAttributeType.N,
    },
},
KeySchema = new List<KeySchemaElement>()
{
    new KeySchemaElement
    {
        AttributeName = "year",
        KeyType = KeyType.HASH,
    },
    new KeySchemaElement
    {
        AttributeName = "title",
        KeyType = KeyType.RANGE,
    },
},
ProvisionedThroughput = new ProvisionedThroughput
{
    ReadCapacityUnits = 5,
    WriteCapacityUnits = 5,
},
});

// Wait until the table is ACTIVE and then report success.
Console.WriteLine("Waiting for table to become active...");

var request = new DescribeTableRequest
{
    TableName = response.TableDescription.TableName,
};

TableStatus status;

int sleepDuration = 2000;

do
{
    System.Threading.Thread.Sleep(sleepDuration);
```

```
        var describeTableResponse = await
client.DescribeTableAsync(request);
        status = describeTableResponse.Table.TableStatus;

        Console.Write(".");
    }
    while (status != "ACTIVE");

    return status == TableStatus.ACTIVE;
}
```

- For API details, see [CreateTable](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table to create.
#     -a attribute_definitions -- JSON file path of a list of attributes and
#                               their types.
#     -k key_schema -- JSON file path of a list of attributes and their key
#                     types.
#     -p provisioned_throughput -- Provisioned throughput settings for the
#                                table.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
```

```
#####
function dynamodb_create_table() {
    local table_name attribute_definitions key_schema provisioned_throughput
    response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_create_table"
    echo "Creates an Amazon DynamoDB table."
    echo " -n table_name -- The name of the table to create."
    echo " -a attribute_definitions -- JSON file path of a list of attributes and
their types."
    echo " -k key_schema -- JSON file path of a list of attributes and their key
types."
    echo " -p provisioned_throughput -- Provisioned throughput settings for the
table."
    echo ""
}

# Retrieve the calling parameters.
while getopt "n:a:k:p:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        a) attribute_definitions="${OPTARG}" ;;
        k) key_schema="${OPTARG}" ;;
        p) provisioned_throughput="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) 
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
```

```
usage
return 1
fi

if [[ -z "$attribute_definitions" ]]; then
    errecho "ERROR: You must provide an attribute definitions json file path the
-a parameter."
    usage
    return 1
fi

if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
parameter."
    usage
    return 1
fi

if [[ -z "$provisioned_throughput" ]]; then
    errecho "ERROR: You must provide a provisioned throughput json file path the
-p parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    attribute_definitions: $attribute_definitions"
iecho "    key_schema: $key_schema"
iecho "    provisioned_throughput: $provisioned_throughput"
iecho ""

response=$(aws dynamodb create-table \
--table-name "$table_name" \
--attribute-definitions file://"$attribute_definitions" \
--key-schema file://"$key_schema" \
--provisioned-throughput "$provisioned_throughput")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
```

```
    fi

    return 0
}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#     $1 - The error code returned by the AWS CLI.
#
# Returns:
#     0: - Success.
#
```

```
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho " The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho " 255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [CreateTable](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Create an Amazon DynamoDB table.
/*!
 \sa createTable()
 \param tableName: Name for the DynamoDB table.
 \param primaryKey: Primary key for the DynamoDB table.
 \param clientConfiguration: AWS client configuration.
```

```
\return bool: Function succeeded.  
*/  
bool AwsDoc::DynamoDB::createTable(const Aws::String &tableName,  
                                    const Aws::String &primaryKey,  
                                    const Aws::Client::ClientConfiguration  
&clientConfiguration) {  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
  
    std::cout << "Creating table " << tableName <<  
        " with a simple primary key: \\" " << primaryKey << "\\." <<  
    std::endl;  
  
    Aws::DynamoDB::Model::CreateTableRequest request;  
  
    Aws::DynamoDB::Model::AttributeDefinition hashKey;  
    hashKey.SetAttributeName(primaryKey);  
    hashKey.SetAttributeType(Aws::DynamoDB::Model::ScalarAttributeType::S);  
    request.AddAttributeDefinitions(hashKey);  
  
    Aws::DynamoDB::Model::KeySchemaElement keySchemaElement;  
    keySchemaElement.WithAttributeName(primaryKey).WithKeyType(  
        Aws::DynamoDB::Model::KeyType::HASH);  
    request.AddKeySchema(keySchemaElement);  
  
    Aws::DynamoDB::Model::ProvisionedThroughput throughput;  
    throughput.WithReadCapacityUnits(5).WithWriteCapacityUnits(5);  
    request.SetProvisionedThroughput(throughput);  
    request.SetTableName(tableName);  
  
    const Aws::DynamoDB::Model::CreateTableOutcome &outcome =  
        dynamoClient.CreateTable(  
            request);  
    if (outcome.IsSuccess()) {  
        std::cout << "Table \\""  
            << outcome.GetResult().GetTableDescription().GetTableName() <<  
            " created!" << std::endl;  
    }  
    else {  
        std::cerr << "Failed to create table: " <<  
        outcome.GetError().GetMessage()  
            << std::endl;  
    }  
  
    return outcome.IsSuccess();
```

{

- For API details, see [CreateTable](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To create a table with tags

The following `create-table` example uses the specified attributes and key schema to create a table named `MusicCollection`. This table uses provisioned throughput and is encrypted at rest using the default AWS owned CMK. The command also applies a tag to the table, with a key of `Owner` and a value of `blueTeam`.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
    AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

Output:

```
{  
    "TableDescription": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "Artist",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "AttributeType": "S"  
            }  
        ],  
        "ProvisionedThroughput": {  
            "NumberOfDecreasesToday": 0,  
            "ReadCapacityUnits": 5,  
            "WriteCapacityUnits": 5  
        }  
    }  
}
```

```
        "WriteCapacityUnits": 5,
        "ReadCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "TableName": "MusicCollection",
    "TableStatus": "CREATING",
    "KeySchema": [
        {
            "KeyType": "HASH",
            "AttributeName": "Artist"
        },
        {
            "KeyType": "RANGE",
            "AttributeName": "SongTitle"
        }
    ],
    "ItemCount": 0,
    "CreationDateTime": "2020-05-26T16:04:41.627000-07:00",
    "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",
    "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To create a table in On-Demand Mode

The following example creates a table called MusicCollection using on-demand mode, rather than provisioned throughput mode. This is useful for tables with unpredictable workloads.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

Output:

```
{  
    "TableDescription": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "Artist",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "AttributeType": "S"  
            }  
        ],  
        "TableName": "MusicCollection",  
        "KeySchema": [  
            {  
                "AttributeName": "Artist",  
                "KeyType": "HASH"  
            },  
            {  
                "AttributeName": "SongTitle",  
                "KeyType": "RANGE"  
            }  
        ],  
        "TableStatus": "CREATING",  
        "CreationDateTime": "2020-05-27T11:44:10.807000-07:00",  
        "ProvisionedThroughput": {  
            "NumberOfDecreasesToday": 0,  
            "ReadCapacityUnits": 0,  
            "WriteCapacityUnits": 0  
        },  
        "TableSizeBytes": 0,  
        "ItemCount": 0,  
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/  
MusicCollection",  
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",  
        "BillingModeSummary": {  
            "BillingMode": "PAY_PER_REQUEST"  
        }  
    }  
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To create a table and encrypt it with a Customer Managed CMK

The following example creates a table named MusicCollection and encrypts it using a customer managed CMK.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-\
abcd-1234-a123-ab1234a1b234
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:12:16.431000-07:00",
        "ProvisionedThroughput": {
```

```
        "NumberOfDecreasesToday": 0,
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
    "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "SSEDescription": {
        "Status": "ENABLED",
        "SSEType": "KMS",
        "KMSMasterKeyArn": "arn:aws:kms:us-west-2:123456789012:key/abcd1234-
abcd-1234-a123-ab1234a1b234"
    }
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 4: To create a table with a Local Secondary Index

The following example uses the specified attributes and key schema to create a table named MusicCollection with a Local Secondary Index named AlbumTitleIndex.

```
aws dynamodb create-table \
--table-name MusicCollection \
--attribute-definitions AttributeName=Artist,AttributeType=S
AttributeName=SongTitle,AttributeType=S AttributeName=AlbumTitle,AttributeType=S \
\
--key-schema AttributeName=Artist,KeyType=HASH
AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--local-secondary-indexes \
"[
{
    \"IndexName\": \"AlbumTitleIndex\",
    \"KeySchema\": [
        {\"AttributeName\": \"Artist\", \"KeyType\": \"HASH\"},
        {\"AttributeName\": \"AlbumTitle\", \"KeyType\": \"RANGE\"}
    ],
    \"Projection\": {
```

```
\\"ProjectionType\\": \\"INCLUDE\",
\\"NonKeyAttributes\\": [\\"Genre\\", \\"Year\\"]
}
]
]"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
    }
}
```

```
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Genre",
                        "Year"
                    ]
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ]
    }
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 5: To create a table with a Global Secondary Index

The following example creates a table named GameScores with a Global Secondary Index called GameTitleIndex. The base table has a partition key of UserId and a sort key of GameTitle, allowing you to find an individual user's best score for a specific game efficiently, whereas the GSI has a partition key of GameTitle and a sort key of TopScore, allowing you to quickly find the overall highest score for a particular game.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S
    AttributeName=GameTitle,AttributeType=S AttributeName=TopScore,AttributeType=N \
    --key-schema AttributeName=UserId,KeyType=HASH \
        AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
    "[
        {
            \"IndexName\": \"GameTitleIndex\",
            \"KeySchema\": [
                {\"AttributeName\": \"GameTitle\", \"KeyType\": \"HASH\"},
                {\"AttributeName\": \"TopScore\", \"KeyType\": \"RANGE\"}
            ],
            \"Projection\": {
                \"ProjectionType\": \"INCLUDE\",
                \"NonKeyAttributes\": [\"UserId\"]
            },
            \"ProvisionedThroughput\": {
                \"ReadCapacityUnits\": 10,
                \"WriteCapacityUnits\": 5
            }
        }
    ]"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ]
    }
}
```

```
],
  "TableName": "GameScores",
  "KeySchema": [
    {
      "AttributeName": "UserId",
      "KeyType": "HASH"
    },
    {
      "AttributeName": "GameTitle",
      "KeyType": "RANGE"
    }
  ],
  "TableStatus": "CREATING",
  "CreationDateTime": "2020-05-26T17:28:15.602000-07:00",
  "ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
  },
  "TableSizeBytes": 0,
  "ItemCount": 0,
  "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
  "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "GlobalSecondaryIndexes": [
    {
      "IndexName": "GameTitleIndex",
      "KeySchema": [
        {
          "AttributeName": "GameTitle",
          "KeyType": "HASH"
        },
        {
          "AttributeName": "TopScore",
          "KeyType": "RANGE"
        }
      ],
      "Projection": {
        "ProjectionType": "INCLUDE",
        "NonKeyAttributes": [
          "UserId"
        ]
      },
      "IndexStatus": "CREATING",
      "ProvisionedThroughput": {
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
      }
    }
  ]
}
```

```
        "NumberOfDecreasesToday": 0,
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
    },
    "IndexSizeBytes": 0,
    "ItemCount": 0,
    "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
}
]
}
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 6: To create a table with multiple Global Secondary Indexes at once

The following example creates a table named GameScores with two Global Secondary Indexes. The GSI schemas are passed via a file, rather than on the command line.

```
aws dynamodb create-table \
--table-name GameScores \
--attribute-definitions AttributeName=UserId,AttributeType=S
AttributeName=GameTitle,AttributeType=S AttributeName=TopScore,AttributeType=N
AttributeName=Date,AttributeType=S \
--key-schema AttributeName=UserId,KeyType=HASH
AttributeName=GameTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--global-secondary-indexes file://gsi.json
```

Contents of gsi.json:

```
[
{
    "IndexName": "GameTitleIndex",
    "KeySchema": [
        {
            "AttributeName": "GameTitle",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "UserId",
            "KeyType": "RANGE"
        }
    ],
    "ProvisionedThroughput": {
        "ReadCapacityUnits": 10,
        "WriteCapacityUnits": 5
    }
}
```

```
        "AttributeName": "TopScore",
        "KeyType": "RANGE"
    },
],
"Projection": {
    "ProjectionType": "ALL"
},
"ProvisionedThroughput": {
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
}
},
{
    "IndexName": "GameDateIndex",
    "KeySchema": [
        {
            "AttributeName": "GameTitle",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "Date",
            "KeyType": "RANGE"
        }
    ],
    "Projection": {
        "ProjectionType": "ALL"
    },
    "ProvisionedThroughput": {
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    }
}
]
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Date",
                "AttributeType": "S"
            },

```

```
{  
    "AttributeName": "GameTitle",  
    "AttributeType": "S"  
},  
{  
    "AttributeName": "TopScore",  
    "AttributeType": "N"  
},  
{  
    "AttributeName": "UserId",  
    "AttributeType": "S"  
}  
,  
]  
,  
"TableName": "GameScores",  
"KeySchema": [  
    {  
        "AttributeName": "UserId",  
        "KeyType": "HASH"  
    },  
    {  
        "AttributeName": "GameTitle",  
        "KeyType": "RANGE"  
    }  
,  
    "TableStatus": "CREATING",  
    "CreationDateTime": "2020-08-04T16:40:55.524000-07:00",  
    "ProvisionedThroughput": {  
        "NumberOfDecreasesToday": 0,  
        "ReadCapacityUnits": 10,  
        "WriteCapacityUnits": 5  
    },  
    "TableSizeBytes": 0,  
    "ItemCount": 0,  
    "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",  
    "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",  
    "GlobalSecondaryIndexes": [  
        {  
            "IndexName": "GameTitleIndex",  
            "KeySchema": [  
                {  
                    "AttributeName": "GameTitle",  
                    "KeyType": "HASH"  
                },  
                {  
                    "AttributeName": "TopScore",  
                    "KeyType": "RANGE"  
                }  
            ]  
        }  
    ]  
}
```

```
        "AttributeName": "TopScore",
        "KeyType": "RANGE"
    },
],
"Projection": {
    "ProjectionType": "ALL"
},
"IndexStatus": "CREATING",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"IndexSizeBytes": 0,
"ItemCount": 0,
"IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
},
{
    "IndexName": "GameDateIndex",
    "KeySchema": [
        {
            "AttributeName": "GameTitle",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "Date",
            "KeyType": "RANGE"
        }
    ],
    "Projection": {
        "ProjectionType": "ALL"
    },
    "IndexStatus": "CREATING",
    "ProvisionedThroughput": {
        "NumberOfDecreasesToday": 0,
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    },
    "IndexSizeBytes": 0,
    "ItemCount": 0,
    "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameDateIndex"
}
```

```
        ]  
    }  
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 7: To create a table with Streams enabled

The following example creates a table called GameScores with DynamoDB Streams enabled. Both new and old images of each item will be written to the stream.

```
aws dynamodb create-table \  
  --table-name GameScores \  
  --attribute-definitions AttributeName=UserId,AttributeType=S  
  AttributeName=GameTitle,AttributeType=S \  
  --key-schema AttributeName=UserId,KeyType=HASH  
  AttributeName=GameTitle,KeyType=RANGE \  
  --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \  
  --stream-specification StreamEnabled=TRUE,StreamViewType=NEW_AND_OLD_IMAGES
```

Output:

```
{  
  "TableDescription": {  
    "AttributeDefinitions": [  
      {  
        "AttributeName": "GameTitle",  
        "AttributeType": "S"  
      },  
      {  
        "AttributeName": "UserId",  
        "AttributeType": "S"  
      }  
    ],  
    "TableName": "GameScores",  
    "KeySchema": [  
      {  
        "AttributeName": "UserId",  
        "KeyType": "HASH"  
      },  
      {  
        "AttributeName": "GameTitle",  
        "KeyType": "RANGE"  
      }  
    ]  
  }  
}
```

```
        "AttributeName": "GameTitle",
        "KeyType": "RANGE"
    },
],
"TableStatus": "CREATING",
"CreationDateTime": "2020-05-27T10:49:34.056000-07:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"StreamSpecification": {
    "StreamEnabled": true,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
},
"LatestStreamLabel": "2020-05-27T17:49:34.056",
"LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2020-05-27T17:49:34.056"
}
}
```

For more information, see [Basic Operations for Tables](#) in the *Amazon DynamoDB Developer Guide*.

Example 8: To create a table with Keys-Only Stream enabled

The following example creates a table called GameScores with DynamoDB Streams enabled. Only the key attributes of modified items are written to the stream.

```
aws dynamodb create-table \
--table-name GameScores \
--attribute-definitions AttributeName=UserId,AttributeType=S \
AttributeName=GameTitle,AttributeType=S \
--key-schema AttributeName=UserId,KeyType=HASH \
AttributeName=GameTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
--stream-specification StreamEnabled=TRUE,StreamViewType=KEYS_ONLY
```

Output:

```
{  
    "TableDescription": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "GameTitle",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "UserId",  
                "AttributeType": "S"  
            }  
        ],  
        "TableName": "GameScores",  
        "KeySchema": [  
            {  
                "AttributeName": "UserId",  
                "KeyType": "HASH"  
            },  
            {  
                "AttributeName": "GameTitle",  
                "KeyType": "RANGE"  
            }  
        ],  
        "TableStatus": "CREATING",  
        "CreationDateTime": "2023-05-25T18:45:34.140000+00:00",  
        "ProvisionedThroughput": {  
            "NumberOfDecreasesToday": 0,  
            "ReadCapacityUnits": 10,  
            "WriteCapacityUnits": 5  
        },  
        "TableSizeBytes": 0,  
        "ItemCount": 0,  
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",  
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",  
        "StreamSpecification": {  
            "StreamEnabled": true,  
            "StreamViewType": "KEYS_ONLY"  
        },  
        "LatestStreamLabel": "2023-05-25T18:45:34.140",  
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/  
GameScores/stream/2023-05-25T18:45:34.140",  
        "DeletionProtectionEnabled": false  
    }  
}
```

{

For more information, see [Change data capture for DynamoDB Streams](#) in the *Amazon DynamoDB Developer Guide*.

Example 9: To create a table with the Standard Infrequent Access class

The following example creates a table called GameScores and assigns the Standard-Infrequent Access (DynamoDB Standard-IA) table class. This table class is optimized for storage being the dominant cost.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S
    AttributeName=GameTitle,AttributeType=S \
    --key-schema AttributeName=UserId,KeyType=HASH
    AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

Output:

```
{  
    "TableDescription": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "GameTitle",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "UserId",  
                "AttributeType": "S"  
            }  
        ],  
        "TableName": "GameScores",  
        "KeySchema": [  
            {  
                "AttributeName": "UserId",  
                "KeyType": "HASH"  
            },  
            {  
                "AttributeName": "GameTitle",  
                "KeyType": "RANGE"  
            }  
        ]  
    }  
}
```

```
        "KeyType": "RANGE"
    },
],
"TableStatus": "CREATING",
"CreationDateTime": "2023-05-25T18:33:07.581000+00:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"TableClassSummary": {
    "TableClass": "STANDARD_INFREQUENT_ACCESS"
},
"DeletionProtectionEnabled": false
}
}
```

For more information, see [Table classes](#) in the *Amazon DynamoDB Developer Guide*.

Example 10: To Create a table with Delete Protection enabled

The following example creates a table called GameScores and enables deletion protection.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S
    AttributeName=GameTitle,AttributeType=S \
    --key-schema AttributeName=UserId,KeyType=HASH
    AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --deletion-protection-enabled
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
```

```
        "AttributeName": "GameTitle",
        "AttributeType": "S"
    },
    {
        "AttributeName": "UserId",
        "AttributeType": "S"
    }
],
"TableName": "GameScores",
"KeySchema": [
    {
        "AttributeName": "UserId",
        "KeyType": "HASH"
    },
    {
        "AttributeName": "GameTitle",
        "KeyType": "RANGE"
    }
],
"TableStatus": "CREATING",
"CreationDateTime": "2023-05-25T23:02:17.093000+00:00",
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
"TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
"DeletionProtectionEnabled": true
}
}
```

For more information, see [Using deletion protection](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [CreateTable](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// CreateMovieTable creates a DynamoDB table with a composite primary key defined
// as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable() (*types.TableDescription, error) {
    var tableDesc *types.TableDescription
    table, err := basics.DynamoDbClient.CreateTable(context.TODO(),
        &dynamodb.CreateTableInput{
            AttributeDefinitions: []types.AttributeDefinition{{
               AttributeName: aws.String("year"),
               AttributeType: types.ScalarAttributeTypeN,
            }, {
               AttributeName: aws.String("title"),
               AttributeType: types.ScalarAttributeTypeS,
            }},
            KeySchema: []types.KeySchemaElement{{
               AttributeName: aws.String("year"),
               KeyType:      types.KeyTypeHash,
            }, {
```

```
    AttributeName: aws.String("title"),
    KeyType:       types.KeyTypeRange,
},
TableName: aws.String(basics.TableName),
ProvisionedThroughput: &types.ProvisionedThroughput{
    ReadCapacityUnits: aws.Int64(10),
    WriteCapacityUnits: aws.Int64(10),
},
})
if err != nil {
    log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
} else {
    waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
    err = waiter.Wait(context.TODO(), &dynamodb.DescribeTableInput{
        TableName: aws.String(basics.TableName)}, 5*time.Minute)
    if err != nil {
        log.Printf("Wait for table exists failed. Here's why: %v\n", err)
    }
    tableDesc = table.TableDescription
}
return tableDesc, err
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
```

```
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class CreateTable {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key>

            Where:
            tableName - The Amazon DynamoDB table to create (for example,
            Music3).
            key - The key for the Amazon DynamoDB table (for example,
            Artist).
            """;

        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        System.out.println("Creating an Amazon DynamoDB table " + tableName + "
with a simple primary key: " + key);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
```

```
        .region(region)
        .build();

    String result = createTable(ddb, tableName, key);
    System.out.println("New table is " + result);
    ddb.close();
}

public static String createTable(DynamoDbClient ddb, String tableName, String
key) {
    DynamoDbWaiter dbWaiter = ddb.waiter();
    CreateTableRequest request = CreateTableRequest.builder()
        .attributeDefinitions(AttributeDefinition.builder()
            .attributeName(key)
            .attributeType(ScalarAttributeType.S)
            .build())
        .keySchema(KeySchemaElement.builder()
            .attributeName(key)
            .keyType(KeyType.HASH)
            .build())
        .provisionedThroughput(ProvisionedThroughput.builder()
            .readCapacityUnits(10L)
            .writeCapacityUnits(10L)
            .build())
        .tableName(tableName)
        .build();

    String newTable;
    try {
        CreateTableResponse response = ddb.createTable(request);
        DescribeTableRequest tableRequest = DescribeTableRequest.builder()
            .tableName(tableName)
            .build();

        // Wait until the Amazon DynamoDB table is created.
        WaiterResponse<DescribeTableResponse> waiterResponse =
        dbWaiter.waitUntilTableExists(tableRequest);
        waiterResponse.matched().response().ifPresent(System.out::println);
        newTable = response.tableDescription().tableName();
        return newTable;

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
```

```
        }
        return "";
    }
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
  });
}
```

```
    ProvisionedThroughput: {
      ReadCapacityUnits: 1,
      WriteCapacityUnits: 1,
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTable](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
},
```

```
KeySchema: [
  {
    AttributeName: "CUSTOMER_ID",
    KeyType: "HASH",
  },
  {
    AttributeName: "CUSTOMER_NAME",
    KeyType: "RANGE",
  },
],
ProvisionedThroughput: {
  ReadCapacityUnits: 1,
  WriteCapacityUnits: 1,
},
TableName: "CUSTOMER_LIST",
StreamSpecification: {
  StreamEnabled: false,
},
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTable](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun createNewTable(tableNameVal: String, key: String): String? {
    val attDef = AttributeDefinition {
        attributeName = key
        attributeType = ScalarAttributeType.S
    }

    val keySchemaVal = KeySchemaElement {
        attributeName = key
        keyType = KeyType.Hash
    }

    val provisionedVal = ProvisionedThroughput {
        readCapacityUnits = 10
        writeCapacityUnits = 10
    }

    val request = CreateTableRequest {
        attributeDefinitions = listOf(attDef)
        keySchema = listOf(keySchemaVal)
        provisionedThroughput = provisionedVal
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->

        var tableArn: String
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists { // suspend call
            tableName = tableNameVal
        }
        tableArn = response.tableDescription!!.tableArn.toString()
        println("Table $tableArn is ready")
        return tableArn
    }
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a table.

```
$tableName = "ddb_demo_table_$uuid";
$service->createTable(
    $tableName,
    [
        new DynamoDBAttribute('year', 'N', 'HASH'),
        new DynamoDBAttribute('title', 'S', 'RANGE')
    ]
);

public function createTable(string $tableName, array $attributes)
{
    $keySchema = [];
    $attributeDefinitions = [];
    foreach ($attributes as $attribute) {
        if (is_a($attribute, DynamoDBAttribute::class)) {
            $keySchema[] = ['AttributeName' => $attribute->AttributeName,
'KeyType' => $attribute->KeyType];
            $attributeDefinitions[] =
                ['AttributeName' => $attribute->AttributeName,
'AttributeType' => $attribute->AttributeType];
        }
    }

    $this->dynamoDbClient->createTable([
        'TableName' => $tableName,
        'KeySchema' => $keySchema,
        'AttributeDefinitions' => $attributeDefinitions,
        'ProvisionedThroughput' => ['ReadCapacityUnits' => 10,
'WriteCapacityUnits' => 10],
    ]);
}
```

- For API details, see [CreateTable](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a table for storing movie data.

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
    def create_table(self, table_name):  
        """  
        Creates an Amazon DynamoDB table that can be used to store movie data.  
        The table uses the release year of the movie as the partition key and the  
        title as the sort key.  
  
        :param table_name: The name of the table to create.  
        :return: The newly created table.  
        """  
        try:  
            self.table = self.dyn_resource.create_table(  
                TableName=table_name,  
                KeySchema=[
```

```
        {"AttributeName": "year", "KeyType": "HASH"}, # Partition
key
        {"AttributeName": "title", "KeyType": "RANGE"}, # Sort key
    ],
AttributeDefinitions=[
    {"AttributeName": "year", "AttributeType": "N"},
    {"AttributeName": "title", "AttributeType": "S"},
],
ProvisionedThroughput={
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 10,
},
)
self.table.wait_until_exists()
except ClientError as err:
    logger.error(
        "Couldn't create table %s. Here's why: %s: %s",
        table_name,
        err.response["Error"]["Code"],
        err.response["Error"]["Message"],
    )
    raise
else:
    return self.table
```

- For API details, see [CreateTable in AWS SDK for Python \(Boto3\) API Reference](#).

Ruby

SDK for Ruby

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
    attr_reader :dynamo_resource
```

```
attr_reader :table_name
attr_reader :table

def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
end

# Creates an Amazon DynamoDB table that can be used to store movie data.
# The table uses the release year of the movie as the partition key and the
# title as the sort key.
#
# @param table_name [String] The name of the table to create.
# @return [Aws::DynamoDB::Table] The newly created table.
def create_table(table_name)
    @table = @dynamo_resource.create_table(
        table_name: table_name,
        key_schema: [
            {attribute_name: "year", key_type: "HASH"}, # Partition key
            {attribute_name: "title", key_type: "RANGE"} # Sort key
        ],
        attribute_definitions: [
            {attribute_name: "year", attribute_type: "N"},
            {attribute_name: "title", attribute_type: "S"}
        ],
        provisioned_throughput: {read_capacity_units: 10, write_capacity_units: 10})
    @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
    @table
rescue Aws::DynamoDB::Errors::ServiceError => e
    @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
    raise
end
```

- For API details, see [CreateTable in AWS SDK for Ruby API Reference](#).

Rust

SDK for Rust

 Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn create_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<CreateTableOutput, Error> {
    let a_name: String = key.into();
    let table_name: String = table.into();

    let ad = AttributeDefinition::builder()
        .attribute_name(&a_name)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .map_err(Error::BuildError)?;

    let ks = KeySchemaElement::builder()
        .attribute_name(&a_name)
        .key_type(KeyType::Hash)
        .build()
        .map_err(Error::BuildError)?;

    let pt = ProvisionedThroughput::builder()
        .read_capacity_units(10)
        .write_capacity_units(5)
        .build()
        .map_err(Error::BuildError)?;

    let create_table_response = client
        .create_table()
        .table_name(table_name)
        .key_schema(ks)
        .attribute_definitions(ad)
        .provisioned_throughput(pt)
```

```
.send()
.await;

match create_table_response {
    Ok(out) => {
        println!("Added table {} with key {}", table, key);
        Ok(out)
    }
    Err(e) => {
        eprintln!("Got an error creating table:");
        eprintln!("{}: {}", e);
        Err(Error::unhandled(e))
    }
}
```

- For API details, see [CreateTable](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
    ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                         iv_keytype = 'HASH' ) )
    ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                         iv_keytype = 'RANGE' ) ) ).

DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynamattributedefn=>tt_attributedefinitions(
    ( NEW /aws1/cl_dynamattributedefn( iv_attributename = 'year'
                                         iv_attributetype = 'N' ) )
    ( NEW /aws1/cl_dynamattributedefn( iv_attributename = 'title'
                                         iv_attributetype = 'S' ) ) ).
```

```
" Adjust read/write capacities as desired.  
DATA(lo_dynprovthroughput) = NEW /aws1/cl_dynprovthroughput(  
    iv_readcapacityunits = 5  
    iv_writecapacityunits = 5 ).  
oo_result = lo_dyn->createtable(  
    it_keyschema = lt_keyschema  
    iv_tablename = iv_table_name  
    it_attributedefinitions = lt_attributedefinitions  
    io_provisionedthroughput = lo_dynprovthroughput ).  
" Table creation can take some time. Wait till table exists before  
returning.  
lo_dyn->get_waiter( )->tableexists(  
    iv_max_wait_time = 200  
    iv_tablename      = iv_table_name ).  
MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.  
" This exception can happen if the table already exists.  
CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).  
    DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -  
{ lo_resourceinuseex->av_err_msg }|.  
    MESSAGE lv_error TYPE 'E'.  
ENDTRY.
```

- For API details, see [CreateTable](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

 **Note**

This is prerelease documentation for an SDK in preview release. It is subject to change.

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
///  
/// Create a movie table in the Amazon DynamoDB data store.  
///  
private func createTable() async throws {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = CreateTableInput(  
        attributeDefinitions: [  
            DynamoDBClientTypes.AttributeDefinition(attributeName: "year",  
attributeType: .n),  
            DynamoDBClientTypes.AttributeDefinition(attributeName: "title",  
attributeType: .s),  
        ],  
        keySchema: [  
            DynamoDBClientTypes.KeySchemaElement(attributeName: "year",  
keyType: .hash),  
            DynamoDBClientTypes.KeySchemaElement(attributeName: "title",  
keyType: .range)  
        ],  
        provisionedThroughput: DynamoDBClientTypes.ProvisionedThroughput(  
            readCapacityUnits: 10,  
            writeCapacityUnits: 10  
        ),  
        tableName: self.tableName  
    )  
    let output = try await client.createTable(input: input)  
    if output.tableDescription == nil {  
        throw MoviesError.TableNotFound  
    }  
}
```

- For API details, see [CreateTable in AWS SDK for Swift API reference](#).

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Delete a DynamoDB table using an AWS SDK

The following code examples show how to delete a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- [Accelerate reads with DAX](#)
- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public static async Task<bool> DeleteTableAsync(AmazonDynamoDBClient
client, string tableName)
{
    var request = new DeleteTableRequest
    {
        TableName = tableName,
    };

    var response = await client.DeleteTableAsync(request);
    if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
    {
        Console.WriteLine($"Table {response.TableDescription.TableName} successfully deleted.");
        return true;
    }
    else
    {
        Console.WriteLine("Could not delete table.");
        return false;
    }
}
```

- For API details, see [DeleteTable in AWS SDK for .NET API Reference](#).

Bash

AWS CLI with Bash script

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table to delete.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_delete_table() {
    local table_name response
    local option OPTARG # Required to use getopts command in a function.

    # bashsupport disable=BP5008
    function usage() {
        echo "function dynamodb_delete_table"
        echo "Deletes an Amazon DynamoDB table."
        echo " -n table_name -- The name of the table to delete."
        echo ""
    }

    # Retrieve the calling parameters.
    while getopts "n:h" option; do
        case "${option}" in
```

```
n) table_name="\${OPTARG}" ;;
h)
    usage
    return 0
;;
\?)
    echo "Invalid parameter"
    usage
    return 1
;;
esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho ""

response=$(aws dynamodb delete-table \
    --table-name "$table_name")

local error_code=\$?

if [[ \$error_code -ne 0 ]]; then
    aws_cli_error_log \$error_code
    errecho "ERROR: AWS reports delete-table operation failed.\$response"
    return 1
fi

return 0
}
```

The utility functions used in this example.

```
#####
# function iecho
```

```
#  
# This function enables the script to display the specified text only if  
# the global variable $VERBOSE is set to true.  
#####
function iecho() {  
    if [[ $VERBOSE == true ]]; then  
        echo "$@"  
    fi  
}  
  
#####  
# function errecho  
#  
# This function outputs everything sent to it to STDERR (standard error output).  
#####  
function errecho() {  
    printf "%s\n" "$*" 1>&2  
}  
  
#####  
# function aws_cli_error_log()  
#  
# This function is used to log the error messages from the AWS CLI.  
#  
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.  
#  
# The function expects the following argument:  
#     $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#     0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then
```

```
    errecho " Command syntax invalid."
elif [ "$err_code" == 253 ]; then
    errecho " The system environment or configuration was invalid."
elif [ "$err_code" == 254 ]; then
    errecho " The service returned an error."
elif [ "$err_code" == 255 ]; then
    errecho " 255 is a catch-all error."
fi

return 0
}
```

- For API details, see [DeleteTable](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Delete an Amazon DynamoDB table.
<太后
    \sa deleteTable()
    \param tableName: The DynamoDB table name.
    \param clientConfiguration: AWS client configuration.
    \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteTable(const Aws::String &tableName,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
dynamoClient.DeleteTable(
```

```
        request);
if (result.IsSuccess()) {
    std::cout << "Your table \""
        << result.GetResult().GetTableDescription().GetTableName()
        << " was deleted.\n";
}
else {
    std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
        << std::endl;
}

return result.IsSuccess();
}
```

- For API details, see [DeleteTable](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To delete a table

The following `delete-table` example deletes the `MusicCollection` table.

```
aws dynamodb delete-table \
--table-name MusicCollection
```

Output:

```
{
    "TableDescription": {
        "TableStatus": "DELETING",
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableName": "MusicCollection",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        }
    }
}
```

{}

For more information, see [Deleting a Table](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [DeleteTable](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable() error {
    _, err := basics.DynamoDbClient.DeleteTable(context.TODO(),
        &dynamodb.DeleteTableInput{
            TableName: aws.String(basics.TableName)})
    if err != nil {
        log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
    }
    return err
}
```

- For API details, see [DeleteTable](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */

public class DeleteTable {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName>
            Where:
            tableName - The Amazon DynamoDB table to delete (for example,
            Music3).

            **Warning** This program will delete the table that you specify!
            """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }
    }
}
```

```
}

    String tableName = args[0];
    System.out.format("Deleting the Amazon DynamoDB table %s...\n",
tableName);
    Region region = Region.US_EAST_1;
    DynamoDbClient ddb = DynamoDbClient.builder()
        .region(region)
        .build();

    deleteDynamoDBTable(ddb, tableName);
    ddb.close();
}

public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{
    DeleteTableRequest request = DeleteTableRequest.builder()
        .tableName(tableName)
        .build();

    try {
        ddb.deleteTable(request);

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    System.out.println(tableName + " was successfully deleted!");
}
}
```

- For API details, see [DeleteTable](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [DeleteTable in AWS SDK for JavaScript API Reference](#).

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
```

```
        console.log("Error: Table in use");
    } else {
        console.log("Success", data);
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteTable](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun deleteDynamoDBTable(tableNameVal: String) {
    val request = DeleteTableRequest {
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}
```

- For API details, see [DeleteTable](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function deleteTable(string $TableName)
{
    $this->customWaiter(function () use ($TableName) {
        return $this->dynamoDbClient->deleteTable([
            'TableName' => $TableName,
        ]);
    });
}
```

- For API details, see [DeleteTable](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
```

```
# The table variable is set during the scenario in the call to
# 'exists' if the table exists. Otherwise, it is set by 'create_table'.
self.table = None

def delete_table(self):
    """
    Deletes the table.
    """
    try:
        self.table.delete()
        self.table = None
    except ClientError as err:
        logger.error(
            "Couldn't delete table. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [DeleteTable in AWS SDK for Python \(Boto3\) API Reference](#).

Ruby

SDK for Ruby

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource
  attr_reader :table_name
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
```

```
@dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
@table_name = table_name
@table = nil
@logger = Logger.new($stdout)
@logger.level = Logger::DEBUG
end

# Deletes the table.
def delete_table
  @table.delete
  @table = nil
rescue Aws::DynamoDB::Errors::ServiceError => e
  puts("Couldn't delete table. Here's why:")
  puts("\t#{e.code}: #{e.message}")
  raise
end
```

- For API details, see [DeleteTable](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn delete_table(client: &Client, table: &str) ->
Result<DeleteTableOutput, Error> {
    let resp = client.delete_table().table_name(table).send().await;

    match resp {
        Ok(out) => {
            println!("Deleted table");
            Ok(out)
        }
        Err(e) => Err(Error::Unhandled(e.into())),
    }
}
```

- For API details, see [DeleteTable](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
lo_dyn->deletetable( iv_tablename = iv_table_name ).  
" Wait till the table is actually deleted.  
lo_dyn->get_waiter( )->tablenotexists(  
    iv_max_wait_time = 200  
    iv_tablename      = iv_table_name ).  
MESSAGE 'Table ' && iv_table_name && ' deleted.' TYPE 'I'.  
CATCH /aws1/cx_dynresourcenotfoundex.  
MESSAGE 'The table ' && iv_table_name && ' does not exist' TYPE 'E'.  
CATCH /aws1/cx_dynresourceinuseex.  
MESSAGE 'The table cannot be deleted since it is in use' TYPE 'E'.  
ENDTRY.
```

- For API details, see [DeleteTable](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
///  
/// Deletes the table from Amazon DynamoDB.  
///  
func deleteTable() async throws {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = DeleteTableInput(  
        tableName: self.tableName  
    )  
    _ = try await client.deleteTable(input: input)  
}
```

- For API details, see [DeleteTable in AWS SDK for Swift API reference](#).

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Delete an item from a DynamoDB table using an AWS SDK

The following code examples show how to delete an item from a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Deletes a single item from a DynamoDB table.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="tableName">The name of the table from which the item
/// will be deleted.</param>
/// <param name="movieToDelete">A movie object containing the title and
/// year of the movie to delete.</param>
/// <returns>A Boolean value indicating the success or failure of the
/// delete operation.</returns>
public static async Task<bool> DeleteItemAsync(
    AmazonDynamoDBClient client,
    string tableName,
    Movie movieToDelete)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = movieToDelete.Title },
        ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
    };

    var request = new DeleteItemRequest
    {
        TableName = tableName,
        Key = key,
    };

    var response = await client.DeleteItemAsync(request);
    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_delete_item
#
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys   -- Path to json file containing the keys that identify the item
# to delete.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_delete_item() {
    local table_name keys response
    local option OPTARG # Required to use getopts command in a function.

    # #####
    # Function usage explanation
    #####
    function usage() {
        echo "function dynamodb_delete_item"
        echo "Delete an item from a DynamoDB table."
        echo " -n table_name -- The name of the table."
        echo " -k keys   -- Path to json file containing the keys that identify the
item to delete."
    }
}
```

```
echo ""
}

while getopts "n:k:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        k) keys="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    keys: $keys"
iecho ""

response=$(aws dynamodb delete-item \
    --table-name "$table_name" \
    --key file://"$keys")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
```

```
errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
fi

return 0

}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#     $1 - The error code returned by the AWS CLI.
#
```

```
# Returns:
#         0: - Success.
#
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho " The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho " 255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [DeleteItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Delete an item from an Amazon DynamoDB table.
/*!
 \sa deleteItem()
```

```
\param tableName: The table name.  
\param partitionKey: The partition key.  
\param partitionValue: The value for the partition key.  
\param clientConfiguration: AWS client configuration.  
\return bool: Function succeeded.  
*/  
  
bool AwsDoc::DynamoDB::deleteItem(const Aws::String &tableName,  
                                    const Aws::String &partitionKey,  
                                    const Aws::String &partitionValue,  
                                    const Aws::Client::ClientConfiguration  
&clientConfiguration) {  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
  
    Aws::DynamoDB::Model::DeleteItemRequest request;  
  
    request.AddKey(partitionKey,  
                  Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));  
    request.SetTableName(tableName);  
  
    const Aws::DynamoDB::Model::DeleteItemOutcome &outcome =  
        dynamoClient.DeleteItem(  
            request);  
    if (outcome.IsSuccess()) {  
        std::cout << "Item '" << partitionValue << "' deleted!" << std::endl;  
    }  
    else {  
        std::cerr << "Failed to delete item: " << outcome.GetError().GetMessage()  
              << std::endl;  
    }  
  
    return outcome.IsSuccess();  
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To delete an item

The following `delete-item` example deletes an item from the `MusicCollection` table and requests details about the item that was deleted and the capacity used by the request.

```
aws dynamodb delete-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-values ALL_OLD \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `key.json`:

```
{  
    "Artist": {"S": "No One You Know"},  
    "SongTitle": {"S": "Scared of My Shadow"}  
}
```

Output:

```
{  
    "Attributes": {  
        "AlbumTitle": {  
            "S": "Blue Sky Blues"  
        },  
        "Artist": {  
            "S": "No One You Know"  
        },  
        "SongTitle": {  
            "S": "Scared of My Shadow"  
        }  
    },  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 2.0  
    },  
    "ItemCollectionMetrics": {  
        "ItemCollectionKey": {  
            "Artist": {  
                "S": "No One You Know"  
            }  
        },  
        "SizeEstimateRangeGB": [  
    }
```

```
    0.0,  
    1.0  
]  
}  
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To delete an item conditionally

The following example deletes an item from the ProductCatalog table only if its ProductCategory is either Sporting Goods or Gardening Supplies and its price is between 500 and 600. It returns details about the item that was deleted.

```
aws dynamodb delete-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"456"}}' \  
  --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (#P  
between :lo and :hi)" \  
  --expression-attribute-names file://names.json \  
  --expression-attribute-values file://values.json \  
  --return-values ALL_OLD
```

Contents of names.json:

```
{  
  "#P": "Price"  
}
```

Contents of values.json:

```
{  
  ":cat1": {"S": "Sporting Goods"},  
  ":cat2": {"S": "Gardening Supplies"},  
  ":lo": {"N": "500"},  
  ":hi": {"N": "600"}  
}
```

Output:

```
{  
  "Attributes": {
```

```
    "Id": {
        "N": "456"
    },
    "Price": {
        "N": "550"
    },
    "ProductCategory": {
        "S": "Sporting Goods"
    }
}
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [DeleteItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// DeleteMovie removes a movie from the DynamoDB table.
func (basics TableBasics) DeleteMovie(movie Movie) error {
    _, err := basics.DynamoDbClient.DeleteItem(context.TODO(),
        &dynamodb.DeleteItemInput{
```

```
    TableName: aws.String(basics.TableName), Key: movie.GetKey(),
})
if err != nil {
    log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
err)
}
return err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class DeleteItem {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key> <keyval>

            Where:
            tableName - The Amazon DynamoDB table to delete the item from
            (for example, Music3).
            key - The key used in the Amazon DynamoDB table (for example,
            Artist).\s
        """;
    }
}
```

```
keyval - The key value that represents the item to delete  
(for example, Famous Band).  
""";  
  
if (args.length != 3) {  
    System.out.println(usage);  
    System.exit(1);  
}  
  
String tableName = args[0];  
String key = args[1];  
String keyVal = args[2];  
System.out.format("Deleting item \"%s\" from %s\n", keyVal, tableName);  
Region region = Region.US_EAST_1;  
DynamoDbClient ddb = DynamoDbClient.builder()  
    .region(region)  
    .build();  
  
deleteDynamoDBItem(ddb, tableName, key, keyVal);  
ddb.close();  
}  
  
public static void deleteDynamoDBItem(DynamoDbClient ddb, String tableName,  
String key, String keyVal) {  
    HashMap<String, AttributeValue> keyToGet = new HashMap<>();  
    keyToGet.put(key, AttributeValue.builder()  
        .s(keyVal)  
        .build());  
  
    DeleteItemRequest deleteReq = DeleteItemRequest.builder()  
        .tableName(tableName)  
        .key(keyToGet)  
        .build();  
  
    try {  
        ddb.deleteItem(deleteReq);  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    }  
}
```

- For API details, see [DeleteItem in AWS SDK for Java 2.x API Reference](#).

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [DeleteCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, DeleteCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new DeleteCommand({
    TableName: "Sodas",
    Key: {
      Flavor: "Cola",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteItem in AWS SDK for JavaScript API Reference](#).

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an item from a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

Delete an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

docClient.delete(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteItem](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun deleteDynamoDBItem(tableNameVal: String, keyName: String, keyVal: String) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request = DeleteItemRequest {
        tableName = tableNameVal
        key = keyToGet
    }
```

```
DynamoDbClient { region = "us-east-1" }.use { ddb ->
    ddb.deleteItem(request)
    println("Item with key matching $keyVal was deleted")
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$key = [
    'Item' => [
        'title' => [
            'S' => $movieName,
        ],
        'year' => [
            'N' => $movieYear,
        ],
    ],
];
];

$key;
$service->deleteItemByKey($tableName, $key);
echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";

public function deleteItemByKey(string $tableName, array $key)
{
    $this->dynamoDbClient->deleteItem([
        'Key' => $key['Item'],
        'TableName' => $tableName,
    ]);
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
    def delete_movie(self, title, year):  
        """  
        Deletes a movie from the table.  
  
        :param title: The title of the movie to delete.  
        :param year: The release year of the movie to delete.  
        """  
        try:  
            self.table.delete_item(Key={"year": year, "title": title})  
        except ClientError as err:  
            logger.error(  
                "Couldn't delete movie %s. Here's why: %s: %s",  
                title,  
                err.response["Error"]["Code"],  
                err.response["Error"]["Message"],
```

```
)  
raise
```

You can specify a condition so that an item is deleted only when it meets certain criteria.

```
class UpdateQueryWrapper:  
    def __init__(self, table):  
        self.table = table  
  
    def delete_underrated_movie(self, title, year, rating):  
        """  
        Deletes a movie only if it is rated below a specified value. By using a  
        condition expression in a delete operation, you can specify that an item  
        is  
        deleted only when it meets certain criteria.  
  
        :param title: The title of the movie to delete.  
        :param year: The release year of the movie to delete.  
        :param rating: The rating threshold to check before deleting the movie.  
        """  
        try:  
            self.table.delete_item(  
                Key={"year": year, "title": title},  
                ConditionExpression="info.rating <= :val",  
                ExpressionAttributeValues={":val": Decimal(str(rating))},  
            )  
        except ClientError as err:  
            if err.response["Error"]["Code"] ==  
                "ConditionalCheckFailedException":  
                logger.warning(  
                    "Didn't delete %s because its rating is greater than %s.",  
                    title,  
                    rating,  
                )  
            else:  
                logger.error(  
                    "Couldn't delete movie %s. Here's why: %s: %s",  
                    title,  
                    err.response["Error"]["Code"],  
                    err.response["Error"]["Message"],  
                )
```

```
)  
raise
```

- For API details, see [DeleteItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics  
  attr_reader :dynamo_resource  
  attr_reader :table  
  
  def initialize(table_name)  
    client = Aws::DynamoDB::Client.new(region: "us-east-1")  
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)  
    @table = @dynamo_resource.table(table_name)  
  end  
  
  # Deletes a movie from the table.  
  #  
  # @param title [String] The title of the movie to delete.  
  # @param year [Integer] The release year of the movie to delete.  
  def delete_item(title, year)  
    @table.delete_item(key: {"year" => year, "title" => title})  
  rescue Aws::DynamoDB::Errors::ServiceError => e  
    puts("Couldn't delete movie #{title}. Here's why:")  
    puts("\t#{e.code}: #{e.message}")  
    raise  
  end
```

- For API details, see [DeleteItem](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn delete_item(
    client: &Client,
    table: &str,
    key: &str,
    value: &str,
) -> Result<DeleteItemOutput, Error> {
    match client
        .delete_item()
        .table_name(table)
        .key(key, AttributeValue::S(value.into()))
        .send()
        .await
    {
        Ok(out) => {
            println!("Deleted item from table");
            Ok(out)
        }
        Err(e) => Err(Error::unhandled(e)),
    }
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.  
  DATA(lo_resp) = lo_dyn->deleteitem(  
    iv_tablename          = iv_table_name  
    it_key                = it_key_input ).  
  MESSAGE 'Deleted one item.' TYPE 'I'.  
  CATCH /aws1/cx_dyncondalcheckfaile00.  
    MESSAGE 'A condition specified in the operation could not be evaluated.'  
    TYPE 'E'.  
    CATCH /aws1/cx_dynresourcenotfoundex.  
      MESSAGE 'The table or index does not exist' TYPE 'E'.  
    CATCH /aws1/cx_dyntransactconflictex.  
      MESSAGE 'Another transaction is using the item' TYPE 'E'.  
  ENDTRY.
```

- For API details, see [DeleteItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Delete a movie, given its title and release year.  
///  
/// - Parameters:  
///   - title: The movie's title.  
///   - year: The movie's release year.  
  
func delete(title: String, year: Int) async throws {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = DeleteItemInput(  
        key: [  
            "year": .n(String(year)),  
            "title": .s(title)  
        ],  
        tableName: self.tableName  
    )  
    _ = try await client.deleteItem(input: input)  
}
```

- For API details, see [DeleteItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Get a batch of DynamoDB items using an AWS SDK

The following code examples show how to get a batch of DynamoDB items.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace LowLevelBatchGet
{
    public class LowLevelBatchGet
    {
        private static readonly string _table1Name = "Forum";
        private static readonly string _table2Name = "Thread";

        public static async void
RetrieveMultipleItemsBatchGet(AmazonDynamoDBClient client)
        {
            var request = new BatchGetItemRequest
            {
                RequestItems = new Dictionary<string, KeysAndAttributes>()
            {
                { _table1Name,
                    new KeysAndAttributes
                    {
                        Keys = new List<Dictionary<string, AttributeValue> >()
                        {
                            new Dictionary<string, AttributeValue>()
                            {
                                { "Name", new AttributeValue {
                                    S = "Amazon DynamoDB"
                                } }
                            },
                            new Dictionary<string, AttributeValue>()
                            {

```

```
        { "Name", new AttributeValue {
            S = "Amazon S3"
        } }
    }
}
},
{
    _table2Name,
    new KeysAndAttributes
    {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue {
                    S = "Amazon DynamoDB"
                } },
                { "Subject", new AttributeValue {
                    S = "DynamoDB Thread 1"
                } }
            },
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue {
                    S = "Amazon DynamoDB"
                } },
                { "Subject", new AttributeValue {
                    S = "DynamoDB Thread 2"
                } }
            },
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue {
                    S = "Amazon S3"
                } },
                { "Subject", new AttributeValue {
                    S = "S3 Thread 1"
                } }
            }
        }
    }
}
};
```

```
BatchGetItemResponse response;
do
{
    Console.WriteLine("Making request");
    response = await client.BatchGetItemAsync(request);

    // Check the response.
    var responses = response.Responses; // Attribute list in the
response.

    foreach (var tableResponse in responses)
    {
        var tableResults = tableResponse.Value;
        Console.WriteLine("Items retrieved from table {0}",
tableResponse.Key);
        foreach (var item1 in tableResults)
        {
            PrintItem(item1);
        }
    }

    // Any unprocessed keys? Could happen if you exceed
ProvisionedThroughput or some other error.
    Dictionary<string, KeysAndAttributes> unprocessedKeys =
response.UnprocessedKeys;
    foreach (var unprocessedTableKeys in unprocessedKeys)
    {
        // Print table name.
        Console.WriteLine(unprocessedTableKeys.Key);
        // Print unprocessed primary keys.
        foreach (var key in unprocessedTableKeys.Value.Keys)
        {
            PrintItem(key);
        }
    }

    request.RequestItems = unprocessedKeys;
} while (response.UnprocessedKeys.Count > 0);
}

private static void PrintItem(Dictionary<string, AttributeValue>
attributeList)
{
```

```
foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
{
    string attributeName = kvp.Key;
    AttributeValue value = kvp.Value;

    Console.WriteLine(
        attributeName + " " +
        (value.S == null ? "" : "S=[" + value.S + "]") +
        (value.N == null ? "" : "N=[" + value.N + "]") +
        (value.SS == null ? "" : "SS=[" + string.Join(",", value.SS.ToArray()) + "]") +
        (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToArray()) + "]")
    );
}

Console.WriteLine("*****");
}

static void Main()
{
    var client = new AmazonDynamoDBClient();

    RetrieveMultipleItemsBatchGet(client);
}
}
```

- For API details, see [BatchGetItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
```

```
# function dynamodb_batch_get_item
#
# This function gets a batch of items from a DynamoDB table.
#
# Parameters:
#     -i item -- Path to json file containing the keys of the items to get.
#
# Returns:
#     The items as json output.
# And:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_batch_get_item() {
    local item response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_batch_get_item"
    echo "Get a batch of items from a DynamoDB table."
    echo " -i item -- Path to json file containing the keys of the items to"
    echo "get."
    echo ""
}

while getopt "i:h" option; do
    case "${option}" in
        i) item="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1
```

```
if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
fi

response=$(aws dynamodb batch-get-item \
    --request-items file://"$item")
local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-get-item operation failed.$response"
    return 1
fi

echo "$response"

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#      $1 - The error code returned by the AWS CLI.
```

```
#  
# Returns:  
#         0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}  
#####
```

- For API details, see [BatchGetItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Batch get items from different Amazon DynamoDB tables.  
/*!
```

```
\sa batchGetItem()
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::batchGetItem(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

Aws::DynamoDB::Model::BatchGetItemRequest request;

// Table1: Forum.
Aws::String table1Name = "Forum";
Aws::DynamoDB::Model::KeysAndAttributes table1KeysAndAttributes;

// Table1: Projection expression.
table1KeysAndAttributes.SetProjectionExpression("#n, Category, Messages,
#v");

// Table1: Expression attribute names.
Aws::Http::HeaderValueCollection headerValueCollection;
headerValueCollection.emplace("#n", "Name");
headerValueCollection.emplace("#v", "Views");
table1KeysAndAttributes.SetExpressionAttributeNames(headerValueCollection);

// Table1: Set key name, type, and value to search.
std::vector<Aws::String> nameValues = {"Amazon DynamoDB", "Amazon S3"};
for (const Aws::String &name: nameValues) {
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> keys;
    Aws::DynamoDB::Model::AttributeValue key;
    key.SetS(name);
    keys.emplace("Name", key);
    table1KeysAndAttributes.AddKeys(keys);
}

Aws::Map<Aws::String, Aws::DynamoDB::Model::KeysAndAttributes> requestItems;
requestItems.emplace(table1Name, table1KeysAndAttributes);

// Table2: ProductCatalog.
Aws::String table2Name = "ProductCatalog";
Aws::DynamoDB::Model::KeysAndAttributes table2KeysAndAttributes;
table2KeysAndAttributes.SetProjectionExpression("Title, Price, Color");

// Table2: Set key name, type, and value to search.
std::vector<Aws::String> idValues = {"102", "103", "201"};
```

```
for (const Aws::String &id: idValues) {
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> keys;
    Aws::DynamoDB::Model::AttributeValue key;
    key.SetN(id);
    keys.emplace("Id", key);
    table2KeysAndAttributes.AddKeys(keys);
}

requestItems.emplace(table2Name, table2KeysAndAttributes);

bool result = true;
do { // Use a do loop to handle pagination.
    request.SetRequestItems(requestItems);
    const Aws::DynamoDB::Model::BatchGetItemOutcome &outcome =
dynamoClient.BatchGetItem(
        request);

    if (outcome.IsSuccess()) {
        for (const auto &responsesMapEntry:
outcome.GetResult().GetResponses()) {
            Aws::String tableName = responsesMapEntry.first;
            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &tableResults = responsesMapEntry.second;
            std::cout << "Retrieved " << tableResults.size()
                << " responses for table '" << tableName << "'.\n"
                << std::endl;
            if (tableName == "Forum") {

                std::cout << "Name | Category | Message | Views" <<
std::endl;
                for (const Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue> &item: tableResults) {
                    std::cout << item.at("Name").GetS() << " | ";
                    std::cout << item.at("Category").GetS() << " | ";
                    std::cout << (item.count("Message") == 0 ? "" : item.at(
                        "Messages").GetN()) << " | ";
                    std::cout << (item.count("Views") == 0 ? "" : item.at(
                        "Views").GetN()) << std::endl;
                }
            }
        else {
            std::cout << "Title | Price | Color" << std::endl;
            for (const Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue> &item: tableResults) {
```

```
        std::cout << item.at("Title").GetS() << " | ";
        std::cout << (item.count("Price") == 0 ? "" : item.at(
            "Price").GetN());
        if (item.count("Color")) {
            std::cout << " | ";
            for (const
std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> &listItem: item.at(
                "Color").GetL())
                std::cout << listItem->GetS() << " ";
        }
        std::cout << std::endl;
    }
}

// If necessary, repeat request for remaining items.
requestItems = outcome.GetResult().GetUnprocessedKeys();
}
else {
    std::cerr << "Batch get item failed: " <<
outcome.GetError().GetMessage()
        << std::endl;
    result = false;
    break;
}
} while (!requestItems.empty());

return result;
}
```

- For API details, see [BatchGetItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To retrieve multiple items from a table

The following batch-get-items example reads multiple items from the MusicCollection table using a batch of three GetItem requests, and requests the

number of read capacity units consumed by the operation. The command returns only the `AlbumTitle` attribute.

```
aws dynamodb batch-get-item \  
  --request-items file://request-items.json \  
  --return-consumed-capacity TOTAL
```

Contents of `request-items.json`:

```
{  
    "MusicCollection": {  
        "Keys": [  
            {  
                "Artist": {"S": "No One You Know"},  
                "SongTitle": {"S": "Call Me Today"}  
            },  
            {  
                "Artist": {"S": "Acme Band"},  
                "SongTitle": {"S": "Happy Day"}  
            },  
            {  
                "Artist": {"S": "No One You Know"},  
                "SongTitle": {"S": "Scared of My Shadow"}  
            }  
        ],  
        "ProjectionExpression": "AlbumTitle"  
    }  
}
```

Output:

```
{  
    "Responses": {  
        "MusicCollection": [  
            {  
                "AlbumTitle": {  
                    "S": "Somewhat Famous"  
                }  
            },  
            {  
                "AlbumTitle": {  
                    "S": "Blue Sky Blues"  
                }  
            }  
        ]  
    }  
}
```

```
        }
    },
    {
        "AlbumTitle": {
            "S": "Louder Than Ever"
        }
    }
],
{
    "UnprocessedKeys": {},
    "ConsumedCapacity": [
        {
            "TableName": "MusicCollection",
            "CapacityUnits": 1.5
        }
    ]
}
```

For more information, see [Batch Operations](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [BatchGetItem](#) in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

shows how to get batch items using the service client.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.KeysAndAttributes;
import java.util.HashMap;
import java.util.List;
```

```
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development environment,
 * including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class BatchReadItems {
    public static void main(String[] args){
        final String usage = """
            Usage:
            <tableName>
            Where:
            tableName - The Amazon DynamoDB table (for example, Music).\s"""
        ;

        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        getBatchItems(dynamoDbClient, tableName);
    }

    public static void getBatchItems(DynamoDbClient dynamoDbClient, String tableName) {
        // Define the primary key values for the items you want to retrieve.
        Map<String, AttributeValue> key1 = new HashMap<>();
        key1.put("Artist", AttributeValue.builder().s("Artist1").build());

        Map<String, AttributeValue> key2 = new HashMap<>();
        key2.put("Artist", AttributeValue.builder().s("Artist2").build());

        // Construct the batchGetItem request.
        Map<String, KeysAndAttributes> requestItems = new HashMap<>();
        requestItems.put(tableName, KeysAndAttributes.builder()
            .keys(List.of(key1, key2))
        );
    }
}
```

```
.projectionExpression("Artist, SongTitle")
.build());  
  
BatchGetItemRequest batchGetItemRequest = BatchGetItemRequest.builder()
.requestItems(requestItems)
.build();  
  
// Make the batchGetItem request.
BatchGetItemResponse batchGetItemResponse =
dynamoDbClient.batchGetItem(batchGetItemRequest);  
  
// Extract and print the retrieved items.
Map<String, List<Map<String, AttributeValue>>> responses =
batchGetItemResponse.responses();
if (responses.containsKey(tableName)) {
    List<Map<String, AttributeValue>> musicItems =
responses.get(tableName);
    for (Map<String, AttributeValue> item : musicItems) {
        System.out.println("Artist: " + item.get("Artist").s() +
", SongTitle: " + item.get("SongTitle").s());
    }
} else {
    System.out.println("No items retrieved.");
}
}  
}
```

shows how to get batch items using the service client and a paginator.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.KeysAndAttributes;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;  
  
public class BatchGetItemsPaginator {  
  
    public static void main(String[] args){
```

```
final String usage = """""

    Usage:
        <tableName>

    Where:
        tableName - The Amazon DynamoDB table (for example, Music).\s
        """";

String tableName = "Music";
Region region = Region.US_EAST_1;
DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
    .region(region)
    .build();

getBatchItemsPaginator(dynamoDbClient, tableName) ;
}

public static void getBatchItemsPaginator(DynamoDbClient dynamoDbClient,
String tableName) {
    // Define the primary key values for the items you want to retrieve.
    Map<String, AttributeValue> key1 = new HashMap<>();
    key1.put("Artist", AttributeValue.builder().s("Artist1").build());

    Map<String, AttributeValue> key2 = new HashMap<>();
    key2.put("Artist", AttributeValue.builder().s("Artist2").build());

    // Construct the batchGetItem request.
    Map<String, KeysAndAttributes> requestItems = new HashMap<>();
    requestItems.put(tableName, KeysAndAttributes.builder()
        .keys(List.of(key1, key2))
        .projectionExpression("Artist, SongTitle")
        .build());

    BatchGetItemRequest batchGetItemRequest = BatchGetItemRequest.builder()
        .requestItems(requestItems)
        .build();

    // Use batchGetItemPaginator for paginated requests.
    dynamoDbClient.batchGetItemPaginator(batchGetItemRequest).stream()
        .flatMap(response -> response.responses().getOrDefault(tableName,
Collections.emptyList()).stream())
        .forEach(item -> {
            System.out.println("Artist: " + item.get("Artist").s() +
                "SongTitle: " + item.get("SongTitle").s());
        });
}
```

```
        ", SongTitle: " + item.get("SongTitle").s());
    });
}
}
```

- For API details, see [BatchGetItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [BatchGet](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { BatchGetCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchGetCommand({
    // Each key in this object is the name of a table. This example refers
    // to a Books table.
    RequestItems: {
      Books: [
        // Each entry in Keys is an object that specifies a primary key.
        Keys: [
          {
            Title: "How to AWS",
          },
          {
            Title: "DynamoDB for DBAs",
          },
        ],
      ],
    },
  });
}
```

```
// Only return the "Title" and "PageCount" attributes.  
ProjectionExpression: "Title, PageCount",  
},  
},  
});  
  
const response = await docClient.send(command);  
console.log(response.Responses["Books"]);  
return response;  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [BatchGetItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create DynamoDB service object  
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });  
  
var params = {  
    RequestItems: {  
        TABLE_NAME: {  
            Keys: [  
                { KEY_NAME: { N: "KEY_VALUE_1" } },  
                { KEY_NAME: { N: "KEY_VALUE_2" } },  
                { KEY_NAME: { N: "KEY_VALUE_3" } },  
            ],  
            ProjectionExpression: "KEY_NAME, ATTRIBUTE",  
        },  
    },
```

```
};

ddb.batchGetItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    data.Responses.TABLE_NAME.forEach(function (element, index, array) {
      console.log(element);
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [BatchGetItem](#) in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import decimal
import json
import logging
import os
import pprint
import time
import boto3
from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
dynamodb = boto3.resource("dynamodb")

MAX_GET_SIZE = 100 # Amazon DynamoDB rejects a get batch larger than 100 items.
```

```
def do_batch_get(batch_keys):
    """
    Gets a batch of items from Amazon DynamoDB. Batches can contain keys from
    more than one table.

    When Amazon DynamoDB cannot process all items in a batch, a set of
    unprocessed
        keys is returned. This function uses an exponential backoff algorithm to
        retry
        getting the unprocessed keys until all are retrieved or the specified
        number of tries is reached.

    :param batch_keys: The set of keys to retrieve. A batch can contain at most
        100
                    keys. Otherwise, Amazon DynamoDB returns an error.
    :return: The dictionary of retrieved items grouped under their respective
        table names.
    """

    tries = 0
    max_tries = 5
    sleepy_time = 1 # Start with 1 second of sleep, then exponentially increase.
    retrieved = {key: [] for key in batch_keys}
    while tries < max_tries:
        response = dynamodb.batch_get_item(RequestItems=batch_keys)
        # Collect any retrieved items and retry unprocessed keys.
        for key in response.get("Responses", []):
            retrieved[key] += response["Responses"][key]
        unprocessed = response["UnprocessedKeys"]
        if len(unprocessed) > 0:
            batch_keys = unprocessed
            unprocessed_count = sum(
                [len(batch_key["Keys"]) for batch_key in batch_keys.values()])
        )
        logger.info(
            "%s unprocessed keys returned. Sleep, then retry.", unprocessed_count
        )
        tries += 1
        if tries < max_tries:
            logger.info("Sleeping for %s seconds.", sleepy_time)
            time.sleep(sleepy_time)
            sleepy_time = min(sleepy_time * 2, 32)
        else:
            break
```

```
    return retrieved
```

- For API details, see [BatchGetItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Swift

SDK for Swift

 **Note**

This is prerelease documentation for an SDK in preview release. It is subject to change.

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Gets an array of `Movie` objects describing all the movies in the
/// specified list. Any movies that aren't found in the list have no
/// corresponding entry in the resulting array.
///
/// - Parameters
///   - keys: An array of tuples, each of which specifies the title and
///           release year of a movie to fetch from the table.
///
/// - Returns:
///   - An array of `Movie` objects describing each match found in the
///     table.
///
/// - Throws:
///   - `MovieError.ClientUninitialized` if the DynamoDB client has not
///     been initialized.
///   - DynamoDB errors are thrown without change.
func batchGet(keys: [(title: String, year: Int)]) async throws -> [Movie] {
```

```
guard let client = self.ddbClient else {
    throw MovieError.ClientUninitialized
}

var movieList: [Movie] = []
var keyItems: [[Swift.String:DynamoDBClientTypes.AttributeValue]] = []

// Convert the list of keys into the form used by DynamoDB.

for key in keys {
    let item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
        "title": .s(key.title),
        "year": .n(String(key.year))
    ]
    keyItems.append(item)
}

// Create the input record for `batchGetItem()`. The list of requested
// items is in the `requestItems` property. This array contains one
// entry for each table from which items are to be fetched. In this
// example, there's only one table containing the movie data.
//
// If we wanted this program to also support searching for matches
// in a table of book data, we could add a second `requestItem`
// mapping the name of the book table to the list of items we want to
// find in it.
let input = BatchGetItemInput(
    requestItems: [
        self.tableName: .init(
            consistentRead: true,
            keys: keyItems
        )
    ]
)

// Fetch the matching movies from the table.

let output = try await client.batchGetItem(input: input)

// Get the set of responses. If there aren't any, return the empty
// movie list.

guard let responses = output.responses else {
    return movieList
}
```

```
}

// Get the list of matching items for the table with the name
// `tableName`.

guard let responseList = responses[self.tableName] else {
    return movieList
}

// Create `Movie` items for each of the matching movies in the table
// and add them to the `MovieList` array.

for response in responseList {
    movieList.append(try Movie(withItem: response))
}

return movieList
}
```

- For API details, see [BatchGetItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Get an item from a DynamoDB table using an AWS SDK

The following code examples show how to get an item from a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- [Accelerate reads with DAX](#)
- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Gets information about an existing movie from the table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="newMovie">A Movie object containing information about the movie to retrieve.</param>
/// <param name="tableName">The name of the table containing the movie.</param>
/// <returns>A Dictionary object containing information about the item retrieved.</returns>
public static async Task<Dictionary<string, AttributeValue>>
GetItemAsync(AmazonDynamoDBClient client, Movie newMovie, string tableName)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };

    var request = new GetItemRequest
    {
        Key = key,
        TableName = tableName,
    };

    var response = await client.GetItemAsync(request);
    return response.Item;
}
```

- For API details, see [GetItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys   -- Path to json file containing the keys that identify the item
#                  to get.
#     [-q query] -- Optional JMESPath query expression.
#
# Returns:
#     The item as text output.
# And:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_get_item() {
    local table_name keys query response
    local option OPTARG # Required to use getopt command in a function.

    # #####
    # Function usage explanation
    #####
    function usage() {
        echo "function dynamodb_get_item"
        echo "Get an item from a DynamoDB table."
        echo " -n table_name -- The name of the table."
        echo " -k keys   -- Path to json file containing the keys that identify the
#                  item to get."
    }
}
```

```
echo "[ -q query ] -- Optional JMESPath query expression."
echo ""
}
query=""
while getopts "n:k:q:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    k) keys="${OPTARG}" ;;
    q) query="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?) 
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi

if [[ -n "$query" ]]; then
  response=$(aws dynamodb get-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --output text \
    --query "$query")
else
  response=$((
    aws dynamodb get-item \
    --table-name "$table_name" \
```

```
--key file://"${keys}" \
--output text
)
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
fi

if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\\t/s/\\t//1" # Remove initial tab that the JMSEPath
query inserts on some strings.
else
    echo "$response"
fi

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
```

```
# The function expects the following argument:  
#           $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#           0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}  
}
```

- For API details, see [GetItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Get an item from an Amazon DynamoDB table.
<太后!
\sa getItem()
\param tableName: The table name.
\param partitionKey: The partition key.
\param partitionValue: The value for the partition key.
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/



bool AwsDoc::DynamoDB::getItem(const Aws::String &tableName,
                                const Aws::String &partitionKey,
                                const Aws::String &partitionValue,
                                const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::GetItemRequest request;

    // Set up the request.
    request.SetTableName(tableName);
    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));

    // Retrieve the item's fields and values.
    const Aws::DynamoDB::Model::GetItemOutcome &outcome =
dynamoClient.GetItem(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved fields/values.
        const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> &item =
outcome.GetResult().GetItem();
        if (!item.empty()) {
            // Output each retrieved field and its value.
            for (const auto &i: item)
                std::cout << "Values: " << i.first << ":" << i.second.GetS()
                << std::endl;
        }
        else {
            std::cout << "No item found with the key " << partitionKey <<
std::endl;
        }
    }
    else {
        std::cerr << "Failed to get item: " << outcome.GetError().GetMessage();
```

```
    }

    return outcome.IsSuccess();
}
```

- For API details, see [GetItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To read an item in a table

The following get-item example retrieves an item from the MusicCollection table. The table has a hash-and-range primary key (Artist and SongTitle), so you must specify both of these attributes. The command also requests information about the read capacity consumed by the operation.

```
aws dynamodb get-item \
  --table-name MusicCollection \
  --key file://key.json \
  --return-consumed-capacity TOTAL
```

Contents of key.json:

```
{
  "Artist": {"S": "Acme Band"},
  "SongTitle": {"S": "Happy Day"}
}
```

Output:

```
{
  "Item": {
    "AlbumTitle": {
      "S": "Songs About Life"
    },
    "SongTitle": {
      "S": "Happy Day"
    },
  }
}
```

```
        "Artist": {  
            "S": "Acme Band"  
        }  
    },  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 0.5  
    }  
}
```

For more information, see [Reading an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To read an item using a consistent read

The following example retrieves an item from the MusicCollection table using strongly consistent reads.

```
aws dynamodb get-item \  
  --table-name MusicCollection \  
  --key file://key.json \  
  --consistent-read \  
  --return-consumed-capacity TOTAL
```

Contents of key.json:

```
{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"}  
}
```

Output:

```
{  
    "Item": {  
        "AlbumTitle": {  
            "S": "Songs About Life"  
        },  
        "SongTitle": {  
            "S": "Happy Day"  
        },  
        "Artist": {  
            "S": "Acme Band"  
        }  
    }  
}
```

```
        },
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

For more information, see [Reading an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To retrieve specific attributes of an item

The following example uses a projection expression to retrieve only three attributes of the desired item.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id": {"N": "102"}}' \
--projection-expression "#T, #C, #P" \
--expression-attribute-names file://names.json
```

Contents of names.json:

```
{
    "#T": "Title",
    "#C": "ProductCategory",
    "#P": "Price"
}
```

Output:

```
{
    "Item": {
        "Price": {
            "N": "20"
        },
        "Title": {
            "S": "Book 102 Title"
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
}
```

```
    }  
}
```

For more information, see [Reading an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [GetItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the  
// examples.  
// It contains a DynamoDB service client that is used to act on the specified  
// table.  
type TableBasics struct {  
    DynamoDbClient *dynamodb.Client  
    TableName      string  
}  
  
  
// GetMovie gets movie data from the DynamoDB table by using the primary  
// composite key  
// made of title and year.  
func (basics TableBasics) GetMovie(title string, year int) (Movie, error) {  
    movie := Movie{Title: title, Year: year}  
    response, err := basics.DynamoDbClient.GetItem(context.TODO(),  
        &dynamodb.GetItemInput{  
            Key: movie.GetKey(), TableName: aws.String(basics.TableName),  
        })  
    if err != nil {  
        log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)  
    } else {  
        err = attributevalue.UnmarshalMap(response.Item, &movie)  
    }
```

```
if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
}
}
return movie, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [GetItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Gets an item from a table by using the DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """
Usage:
        <tableName> <key> <keyVal>
Where:
```

```
        tableName - The Amazon DynamoDB table from which an item is
retrieved (for example, Music3).\s
        key - The key used in the Amazon DynamoDB table (for example,
Artist).\s
        keyval - The key value that represents the item to get (for
example, Famous Band).
        """;

    if (args.length != 3) {
        System.out.println(usage);
        System.exit(1);
    }

    String tableName = args[0];
    String key = args[1];
    String keyVal = args[2];
    System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal,
tableName);
    Region region = Region.US_EAST_1;
    DynamoDbClient ddb = DynamoDbClient.builder()
        .region(region)
        .build();

    getDynamoDBItem(ddb, tableName, key, keyVal);
    ddb.close();
}

public static void getDynamoDBItem(DynamoDbClient ddb, String tableName,
String key, String keyVal) {
    HashMap<String, AttributeValue> keyToGet = new HashMap<>();
    keyToGet.put(key, AttributeValue.builder()
        .s(keyVal)
        .build());

    GetItemRequest request = GetItemRequest.builder()
        .key(keyToGet)
        .tableName(tableName)
        .build();

    try {
        // If there is no matching item, GetItem does not return any data.
        Map<String, AttributeValue> returnedItem =
ddb.getItem(request).item();
        if (returnedItem.isEmpty())
            return;
    } catch (DynamoDBException e) {
        System.out.println("An error occurred while performing the GetItem operation: " + e.getMessage());
    }
}
```

```
        System.out.format("No item found with the key %s!\n", key);
    else {
        Set<String> keys = returnedItem.keySet();
        System.out.println("Amazon DynamoDB table attributes: \n");
        for (String key1 : keys) {
            System.out.format("%s: %s\n", key1,
returnedItem.get(key1).toString());
        }
    }

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
}
```

- For API details, see [GetItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [GetCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
    const command = new GetCommand({
        TableName: "AngryAnimals",
        Key: {
            id: "1"
        }
    });
    const data = await docClient.send(command);
    console.log(data.Item);
}
```

```
Key: {  
    CommonName: "Shoebill",  
},  
});  
  
const response = await docClient.send(command);  
console.log(response);  
return response;  
};
```

- For API details, see [GetItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get an item from a table.

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create the DynamoDB service object  
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });  
  
var params = {  
    TableName: "TABLE",  
    Key: {  
        KEY_NAME: { N: "001" },  
    },  
    ProjectionExpression: "ATTRIBUTE_NAME",  
};  
  
// Call DynamoDB to read the item from the table  
ddb.getItem(params, function (err, data) {  
    if (err) {  
        console.log("Error", err);  
    } else {  
        console.log(data.Item);  
    }  
});
```

```
    } else {
      console.log("Success", data.Item);
    }
});
```

Get an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetItem](#) in [AWS SDK for JavaScript API Reference](#).

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun getSpecificItem(tableNameVal: String, keyName: String, keyVal: String) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request = GetItemRequest {
        key = keyToGet
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

- For API details, see [GetItem](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$movie = $service->getItemByKey($tableName, $key);
echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";

public function getItemByKey(string $tableName, array $key)
{
    return $this->dynamoDbClient->getItem([

```

```
        'Key' => $key['Item'],
        'TableName' => $tableName,
    ]);
}
```

- For API details, see [GetItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None

    def get_movie(self, title, year):
        """
        Gets movie data from the table for a specific movie.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :return: The data about the requested movie.
        """
        try:
            response = self.table.get_item(Key={"year": year, "title": title})
        except ClientError as err:
```

```
        logger.error(
            "Couldn't get movie %s from table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Item"]
```

- For API details, see [GetItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Gets movie data from the table for a specific movie.
    #
    # @param title [String] The title of the movie.
    # @param year [Integer] The release year of the movie.
    # @return [Hash] The data about the requested movie.
    def get_item(title, year)
        @table.get_item(key: {"year" => year, "title" => title})
```

```
rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't get movie #{title} (#{year}) from table #{@table.name}:\n")
    puts("\t#{$<e.code>}: #{$<e.message>}")
    raise
end
```

- For API details, see [GetItem](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.
    oo_item = lo_dyn->getitem(
        iv_tablename           = iv_table_name
        it_key                 = it_key ).
    DATA(lt_attr) = oo_item->get_item( ).
    DATA(lo_title) = lt_attr[ key = 'title' ]-value.
    DATA(lo_year) = lt_attr[ key = 'year' ]-value.
    DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
    MESSAGE 'Movie name is: ' && lo_title->get_s( )
        && 'Movie year is: ' && lo_year->get_n( )
        && 'Moving rating is: ' && lo_rating->get_n( ) TYPE 'I'.
    CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
ENDTRY.
```

- For API details, see [GetItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Return a `Movie` record describing the specified movie from the Amazon
/// DynamoDB table.
///
/// - Parameters:
///   - title: The movie's title (`String`).
///   - year: The movie's release year (`Int`).
///
/// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
///
/// - Returns: A `Movie` record with the movie's details.
func get(title: String, year: Int) async throws -> Movie {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = GetItemInput(
        key: [
            "year": .n(String(year)),
            "title": .s(title)
        ],
        tableName: self.tableName
    )
    let output = try await client.getItem(input: input)
    guard let item = output.item else {
        throw MoviesError.ItemNotFound
    }
}
```

```
}

let movie = try Movie(withItem: item)
return movie
}
```

- For API details, see [GetItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Get information about a DynamoDB table

The following code examples show how to get information about a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
private static async Task GetTableInformation()
{
    Console.WriteLine("\n*** Retrieving table information ***");

    var response = await Client.DescribeTableAsync(new DescribeTableRequest
    {
        TableName = ExampleTableName
    });
}
```

```
var table = response.Table;
Console.WriteLine($"Name: {table.TableName}");
Console.WriteLine($"# of items: {table.ItemCount}");
Console.WriteLine($"Provision Throughput (reads/sec): " +
    $"{table.ProvisionedThroughput.ReadCapacityUnits}");
Console.WriteLine($"Provision Throughput (writes/sec): " +
    $"{table.ProvisionedThroughput.WriteCapacityUnits}");
}
```

- For API details, see [DescribeTable](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_describe_table
#
# This function returns the status of a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#
# Response:
#     - TableStatus:
#         And:
#             0 - Table is active.
#             1 - If it fails.
#####
function dynamodb_describe_table {
    local table_name
    local option OPTARG # Required to use getopt command in a function.

#####

```

```
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_describe_table"
    echo "Describe the status of a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo ""
}

# Retrieve the calling parameters.
while getopts "n:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

local table_status
table_status=$(
    aws dynamodb describe-table \
        --table-name "$table_name" \
        --output text \
        --query 'Table.TableStatus'
)

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log "$error_code"
```

```
    errecho "ERROR: AWS reports describe-table operation failed.$table_status"
    return 1
}

echo "$table_status"

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#     $1 - The error code returned by the AWS CLI.
#
# Returns:
#     0: - Success.
#
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    fi
}
```

```
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho " The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho " 255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [DescribeTable](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/usr/bin/env sh
#!/bin/bash
#!/bin/sh

//! Describe an Amazon DynamoDB table.
/*!
 * \sa describeTable()
 * \param tableName: The DynamoDB table name.
 * \param clientConfiguration: AWS client configuration.
 * \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::describeTable(const Aws::String &tableName,
                                      const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);
```

```
const Aws::DynamoDB::Model::DescribeTableOutcome &outcome =
dynamoClient.DescribeTable(
    request);

if (outcome.IsSuccess()) {
    const Aws::DynamoDB::Model::TableDescription &td =
outcome.GetResult().GetTable();
    std::cout << "Table name : " << td.GetTableName() << std::endl;
    std::cout << "Table ARN : " << td.GetTableArn() << std::endl;
    std::cout << "Status : "
        <<
    Aws::DynamoDB::Model::TableStatusMapper::GetNameForTableStatus(
        td.GetTableStatus()) << std::endl;
    std::cout << "Item count : " << td.GetItemCount() << std::endl;
    std::cout << "Size (bytes): " << td.GetTableSizeBytes() << std::endl;

    const Aws::DynamoDB::Model::ProvisionedThroughputDescription &ptd =
td.GetProvisionedThroughput();
    std::cout << "Throughput" << std::endl;
    std::cout << " Read Capacity : " << ptd.GetReadCapacityUnits() <<
std::endl;
    std::cout << " Write Capacity: " << ptd.GetWriteCapacityUnits() <<
std::endl;

    const Aws::Vector<Aws::DynamoDB::Model::AttributeDefinition> &ad =
td.GetAttributeDefinitions();
    std::cout << "Attributes" << std::endl;
    for (const auto &a: ad)
        std::cout << " " << a.getAttributeName() << " (" <<

Aws::DynamoDB::Model::ScalarAttributeTypeMapper::GetNameForScalarAttributeType(
    a.getAttributeType()) <<
    ")" << std::endl;
}

else {
    std::cerr << "Failed to describe table: " <<
outcome.GetError().GetMessage();
}

return outcome.IsSuccess();
}
```

- For API details, see [DescribeTable](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To describe a table

The following `describe-table` example describes the `MusicCollection` table.

```
aws dynamodb describe-table \
--table-name MusicCollection
```

Output:

```
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "Artist",
        "AttributeType": "S"
      },
      {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
      }
    ],
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "WriteCapacityUnits": 5,
      "ReadCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "TableName": "MusicCollection",
    "TableStatus": "ACTIVE",
    "KeySchema": [
      {
        "KeyType": "HASH",
        "AttributeName": "Artist"
      },
      {
        "KeyType": "RANGE",
        "AttributeName": "SongTitle"
      }
    ]
  }
}
```

```
        },
    ],
    "ItemCount": 0,
    "CreationDateTime": 1421866952.062
}
}
```

For more information, see [Describing a Table](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [DescribeTable](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// TableExists determines whether a DynamoDB table exists.
func (basics TableBasics) TableExists() (bool, error) {
    exists := true
    _, err := basics.DynamoDbClient.DescribeTable(
        context.TODO(), &dynamodb.DescribeTableInput{TableName:
            aws.String(basics.TableName)},
    )
    if err != nil {
        var notFoundEx *types.ResourceNotFoundException
    }
}
```

```
if errors.As(err, &notFoundEx) {
    log.Printf("Table %v does not exist.\n", basics.TableName)
    err = nil
} else {
    log.Printf("Couldn't determine existence of table %v. Here's why: %v\n",
    basics.TableName, err)
}
exists = false
}
return exists, err
}
```

- For API details, see [DescribeTable](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import
software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughputDescription;
import software.amazon.awssdk.services.dynamodb.model.TableDescription;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
*/
public class DescribeTable {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName>

            Where:
            tableName - The Amazon DynamoDB table to get information
            about (for example, Music3).
            """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        System.out.format("Getting description for %s\n", tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        describeDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void describeDynamoDBTable(DynamoDbClient ddb, String
tableName) {
        DescribeTableRequest request = DescribeTableRequest.builder()
            .tableName(tableName)
            .build();

        try {
            TableDescription tableInfo = ddb.describeTable(request).table();
            if (tableInfo != null) {
                System.out.format("Table name : %s\n", tableInfo.tableName());
                System.out.format("Table ARN : %s\n", tableInfo.tableArn());
                System.out.format("Status : %s\n", tableInfo.tableStatus());
                System.out.format("Item count : %d\n", tableInfo.itemCount());
            }
        }
    }
}
```

```
        System.out.format("Size (bytes): %d\n",
tableInfo.tableSizeBytes());

        ProvisionedThroughputDescription throughputInfo =
tableInfo.provisionedThroughput();
        System.out.println("Throughput");
        System.out.format("  Read Capacity : %d\n",
throughputInfo.readCapacityUnits());
        System.out.format("  Write Capacity: %d\n",
throughputInfo.writeCapacityUnits());

        List<AttributeDefinition> attributes =
tableInfo.attributeDefinitions();
        System.out.println("Attributes");
        for (AttributeDefinition a : attributes) {
            System.out.format("  %s (%s)\n", a.attributeName(),
a.attributeType());
        }
    }

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
System.out.println("\nDone!");
}
```

- For API details, see [DescribeTable](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DescribeTableCommand({
    TableName: "Pastries",
  });

  const response = await client.send(command);
  console.log(`TABLE NAME: ${response.Table.TableName}`);
  console.log(`TABLE ITEM COUNT: ${response.Table.ItemCount}`);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeTable](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to retrieve the selected table descriptions
ddb.describeTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
    } else {
      console.log("Success", data.Table.KeySchema);
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeTable](#) in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None

    def exists(self, table_name):
        """
        Determines whether a table exists. As a side effect, stores the table in
        a member variable.

        :param table_name: The name of the table to check.
        :return: True when the table exists; otherwise, False.
        """
        try:
```

```
        table = self.dyn_resource.Table(table_name)
        table.load()
        exists = True
    except ClientError as err:
        if err.response["Error"]["Code"] == "ResourceNotFoundException":
            exists = False
        else:
            logger.error(
                "Couldn't check for existence of %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
    else:
        self.table = table
    return exists
```

- For API details, see [DescribeTable](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource
  attr_reader :table_name
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
```

```
@table = nil
@logger = Logger.new($stdout)
@logger.level = Logger::DEBUG
end

# Determines whether a table exists. As a side effect, stores the table in
# a member variable.
#
# @param table_name [String] The name of the table to check.
# @return [Boolean] True when the table exists; otherwise, False.
def exists?(table_name)
  @dynamo_resource.client.describe_table(table_name: table_name)
  @logger.debug("Table #{table_name} exists")
rescue Aws::DynamoDB::Errors::ResourceNotFoundException
  @logger.debug("Table #{table_name} doesn't exist")
  false
rescue Aws::DynamoDB::Errors::ServiceError => e
  puts("Couldn't check for existence of #{table_name}:\n")
  puts("\t#{e.code}: #{e.message}")
  raise
end
```

- For API details, see [DescribeTable](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
oo_result = lo_dyn->describetable( iv_tablename = iv_table_name ).
DATA(lv_tablename) = oo_result->get_table( )->ask_tablename( ).
DATA(lv_tablelearn) = oo_result->get_table( )->ask_tablelearn( ).
DATA(lv_tablestatus) = oo_result->get_table( )->ask_tablestatus( ).
DATA(lv_itemcount) = oo_result->get_table( )->ask_itemcount( ).
MESSAGE 'The table name is ' && lv_tablename
```

```
&& '. The table ARN is ' && lv_tablearn  
&& '. The tablestatus is ' && lv_tablestatus  
&& '. Item count is ' && lv_itemcount TYPE 'I'.  
CATCH /aws1/cx_dynresourcenotfoundex.  
MESSAGE 'The table ' && lv_tablename && ' does not exist' TYPE 'E'.  
ENDTRY.
```

- For API details, see [DescribeTable](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

List DynamoDB tables using an AWS SDK

The following code examples show how to list DynamoDB tables.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
private static async Task ListMyTables()  
{  
    Console.WriteLine("\n*** Listing tables ***");  
  
    string lastTableNameEvaluated = null;  
    do  
    {  
        var response = await Client.ListTablesAsync(new ListTablesRequest  
        {  
            Limit = 2,  
            ExclusiveStartTableName = lastTableNameEvaluated  
        });  
    }  
}
```

```
        foreach (var name in response.TableNames)
        {
            Console.WriteLine(name);
        }

        lastTableNameEvaluated = response.LastEvaluatedTableName;
    } while (lastTableNameEvaluated != null);
}
```

- For API details, see [ListTables](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_list_tables
#
# This function lists all the tables in a DynamoDB.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_list_tables() {
    response=$(aws dynamodb list-tables \
        --output text \
        --query "TableNames")

    local error_code=${?}

    if [[ $error_code -ne 0 ]]; then
        aws_cli_error_log $error_code
        errecho "ERROR: AWS reports batch-write-item operation failed.$response"
        return 1
    fi
}
```

```
fi

echo "$response" | tr -s "[[:space:]]" "\n"

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#     $1 - The error code returned by the AWS CLI.
#
# Returns:
#     0: - Success.
#
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
```

```
        elif [ "$err_code" == 252 ]; then
            errecho " Command syntax invalid."
        elif [ "$err_code" == 253 ]; then
            errecho " The system environment or configuration was invalid."
        elif [ "$err_code" == 254 ]; then
            errecho " The service returned an error."
        elif [ "$err_code" == 255 ]; then
            errecho " 255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [ListTables](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ List the Amazon DynamoDB tables for the current AWS account.
/*
\sa listTables()
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/

bool AwsDoc::DynamoDB::listTables(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::ListTablesRequest listTablesRequest;
    listTablesRequest.SetLimit(50);
    do {
        const Aws::DynamoDB::Model::ListTablesOutcome &outcome =
            dynamoClient.ListTables(
```

```
        listTablesRequest);
    if (!outcome.IsSuccess()) {
        std::cout << "Error: " << outcome.GetError().GetMessage() <<
std::endl;
        return false;
    }

    for (const auto &tableName: outcome.GetResult().GetTableNames())
        std::cout << tableName << std::endl;
    listTablesRequest.SetExclusiveStartTableName(
        outcome.GetResult().GetLastEvaluatedTableName());

} while (!listTablesRequest.GetExclusiveStartTableName().empty());

return true;
}
```

- For API details, see [ListTables](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To list tables

The following `list-tables` example lists all of the tables associated with the current AWS account and Region.

```
aws dynamodb list-tables
```

Output:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog",
        "Reply",
        "Thread"
    ]
}
```

For more information, see [Listing Table Names](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To limit page size

The following example returns a list of all existing tables, but retrieves only one item in each call, performing multiple calls if necessary to get the entire list. Limiting the page size is useful when running list commands on a large number of resources, which can result in a "timed out" error when using the default page size of 1000.

```
aws dynamodb list-tables \
--page-size 1
```

Output:

```
{
  "TableNames": [
    "Forum",
    "ProductCatalog",
    "Reply",
    "Thread"
  ]
}
```

For more information, see [Listing Table Names](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To limit the number of items returned

The following example limits the number of items returned to 2. The response includes a NextToken value with which to retrieve the next page of results.

```
aws dynamodb list-tables \
--max-items 2
```

Output:

```
{
  "TableNames": [
    "Forum",
    "ProductCatalog"
  ],
  "NextToken":
    "abCDeFGhiJKlmnOPqrSTuvwxyzYZ1aBCdEFghijK7LM51n0pqRSTuv3WxY3ZabC5dEFGhI2Jk3LmnoPQ6RST9"
```

{

For more information, see [Listing Table Names](#) in the *Amazon DynamoDB Developer Guide*.

Example 4: To retrieve the next page of results

The following command uses the NextToken value from a previous call to the list-tables command to retrieve another page of results. Since the response in this case does not include a NextToken value, we know that we have reached the end of the results.

```
aws dynamodb list-tables \
    --starting-token
    aBCDeFGhiJKLMNOPqrstuvwxyz1aBCdEFghijK7LM51n0pqRSTuv3WxY3ZabC5dEFGhI2Jk3LmnoPQ6RST9
```

Output:

```
{
  "TableNames": [
    "Reply",
    "Thread"
  ]
}
```

For more information, see [Listing Table Names](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [ListTables](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the examples.
```

```
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// ListTables lists the DynamoDB table names for the current account.
func (basics TableBasics) ListTables() ([]string, error) {
    var tableNames []string
    var output *dynamodb.ListTablesOutput
    var err error
    tablePaginator := dynamodb.NewListTablesPaginator(basics.DynamoDbClient,
        &dynamodb.ListTablesInput{})
    for tablePaginator.HasMorePages() {
        output, err = tablePaginator.NextPage(context.TODO())
        if err != nil {
            log.Printf("Couldn't list tables. Here's why: %v\n", err)
            break
        } else {
            tableNames = append(tableNames, output.TableNames...)
        }
    }
    return tableNames, err
}
```

- For API details, see [ListTables](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class ListTables {
    public static void main(String[] args) {
        System.out.println("Listing your Amazon DynamoDB tables:\n");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
        listAllTables(ddb);
        ddb.close();
    }

    public static void listAllTables(DynamoDbClient ddb) {
        boolean moreTables = true;
        String lastName = null;

        while (moreTables) {
            try {
                ListTablesResponse response = null;
                if (lastName == null) {
                    ListTablesRequest request =
ListTablesRequest.builder().build();
                    response = ddb.listTables(request);
                } else {
                    ListTablesRequest request = ListTablesRequest.builder()
                        .exclusiveStartTableName(lastName).build();
                    response = ddb.listTables(request);
                }
            }
        }
    }
}
```

```
        List<String> tableNames = response.tableNames();
        if (tableNames.size() > 0) {
            for (String curName : tableNames) {
                System.out.format("* %s\n", curName);
            }
        } else {
            System.out.println("No tables found!");
            System.exit(0);
        }

        lastName = response.lastEvaluatedTableName();
        if (lastName == null) {
            moreTables = false;
        }
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
System.out.println("\nDone!");
```

- For API details, see [ListTables](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});
```

```
export const main = async () => {
    const command = new ListTablesCommand({});

    const response = await client.send(command);
    console.log(response);
    return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListTables](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

// Call DynamoDB to retrieve the list of tables
ddb.listTables({ Limit: 10 }, function (err, data) {
    if (err) {
        console.log("Error", err.code);
    } else {
        console.log("Table names are ", data.TableNames);
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListTables](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun listAllTables() {  
    DynamoDbClient { region = "us-east-1" }.use { ddb ->  
        val response = ddb.listTables(ListTablesRequest {})  
        response.tableNames?.forEach { tableName ->  
            println("Table name is $tableName")  
        }  
    }  
}
```

- For API details, see [ListTables](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function listTables($exclusiveStartTableName = "", $limit = 100)  
{  
    $this->dynamoDbClient->listTables([  
        'ExclusiveStartTableName' => $exclusiveStartTableName,  
        'Limit' => $limit,  
    ]);  
}
```

- For API details, see [ListTables](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
    def list_tables(self):  
        """  
        Lists the Amazon DynamoDB tables for the current account.  
  
        :return: The list of tables.  
        """  
        try:  
            tables = []  
            for table in self.dyn_resource.tables.all():  
                print(table.name)  
                tables.append(table)  
        except ClientError as err:  
            logger.error(  
                "Couldn't list tables. Here's why: %s: %s",  
                err.response["Error"]["Code"],  
                err.response["Error"]["Message"],  
            )
```

```
        raise
    else:
        return tables
```

- For API details, see [ListTables](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Determine whether a table exists.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
    attr_reader :dynamo_resource
    attr_reader :table_name
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table_name = table_name
        @table = nil
        @logger = Logger.new($stdout)
        @logger.level = Logger::DEBUG
    end

    # Determines whether a table exists. As a side effect, stores the table in
    # a member variable.
    #
    # @param table_name [String] The name of the table to check.
    # @return [Boolean] True when the table exists; otherwise, False.
    def exists?(table_name)
        @dynamo_resource.client.describe_table(table_name: table_name)
```

```
@logger.debug("Table #{table_name} exists")
rescue Aws::DynamoDB::Errors::ResourceNotFoundException
  @logger.debug("Table #{table_name} doesn't exist")
  false
rescue Aws::DynamoDB::Errors::ServiceError => e
  puts("Couldn't check for existence of #{table_name}:\n")
  puts("\t#{e.code}: #{e.message}")
  raise
end
```

- For API details, see [ListTables](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn list_tables(client: &Client) -> Result<Vec<String>, Error> {
    let paginator = client.list_tables().intoPaginator().items().send();
    let table_names = paginator.collect::<Result<Vec<_>, _>>().await?;

    println!("Tables:");

    for name in &table_names {
        println!("  {}", name);
    }

    println!("Found {} tables", table_names.len());
    Ok(table_names)
}
```

Determine whether table exists.

```
pub async fn table_exists(client: &Client, table: &str) -> Result<bool, Error> {
```

```
    debug!("Checking for table: {table}");
    let table_list = client.list_tables().send().await;

    match table_list {
        Ok(list) => Ok(list.table_names().contains(&table.into())),
        Err(e) => Err(e.into()),
    }
}
```

- For API details, see [ListTables](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
oo_result = lo_dyn->listtables( ).
" You can loop over the oo_result to get table properties like this.
LOOP AT oo_result->get_tablenames( ) INTO DATA(lo_table_name).
  DATA(lv_tablename) = lo_table_name->get_value( ).
ENDLOOP.
DATA(lv_tablecount) = lines( oo_result->get_tablenames( ) ).
MESSAGE 'Found ' && lv_tablecount && ' tables' TYPE 'I'.
CATCH /aws1/cx_rt_service_generic INTO DATA(lo_exception).
  DATA(lv_error) = |"{ lo_exception->av_err_code }" - { lo_exception-
>av_err_msg }|.
  MESSAGE lv_error TYPE 'E'.
ENDTRY.
```

- For API details, see [ListTables](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Get a list of the DynamoDB tables available in the specified Region.  
///  
/// - Returns: An array of strings listing all of the tables available  
///   in the Region specified when the session was created.  
public func getTableList() async throws -> [String] {  
    var tableList: [String] = []  
    var lastEvaluated: String? = nil  
  
    // Iterate over the list of tables, 25 at a time, until we have the  
    // names of every table. Add each group to the `tableList` array.  
    // Iteration is complete when `output.lastEvaluatedTableName` is `nil`.  
  
    repeat {  
        let input = ListTablesInput(  
            exclusiveStartTableName: lastEvaluated,  
            limit: 25  
        )  
        let output = try await self.session.listTables(input: input)  
        guard let tableNames = output.tableNames else {  
            return tableList  
        }  
        tableList.append(contentsOf: tableNames)  
        lastEvaluated = output.lastEvaluatedTableName  
    } while lastEvaluated != nil
```

```
    return tableList
}
```

- For API details, see [ListTables](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Put an item in a DynamoDB table using an AWS SDK

The following code examples show how to put an item in a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- [Accelerate reads with DAX](#)
- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Adds a new item to the table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</
param>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
added.</param>
```

```
    /// <returns>A Boolean value that indicates the results of adding the item.</returns>
    public static async Task<bool> PutItemAsync(AmazonDynamoDBClient client,
        Movie newMovie, string tableName)
    {
        var item = new Dictionary<string, AttributeValue>
        {
            ["title"] = new AttributeValue { S = newMovie.Title },
            ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
        };

        var request = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
        };

        var response = await client.PutItemAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}
```

- For API details, see [PutItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
```

```
#      -i item  -- Path to json file containing the item values.
#
# Returns:
#      0 - If successful.
#      1 - If it fails.
#####
function dynamodb_put_item() {
    local table_name item response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
}

while getopt "n:i:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        i) item="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) 
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi
```

```
if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    item: $item"
iecho ""
iecho ""

response=$(aws dynamodb put-item \
--table-name "$table_name" \
--item file://"$item")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
fi

return 0

}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}
```

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#       $1 - The error code returned by the AWS CLI.
#
# Returns:
#       0: - Success.
#
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho " The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho " 255 is a catch-all error."
    fi

    return 0
}
```

```
}
```

- For API details, see [PutItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
///! Put an item in an Amazon DynamoDB table.
/*!
 \sa putItem()
 \param tableName: The table name.
 \param artistKey: The artist key. This is the partition key for the table.
 \param artistValue: The artist value.
 \param albumTitleKey: The album title key.
 \param albumTitleValue: The album title value.
 \param awardsKey: The awards key.
 \param awardsValue: The awards value.
 \param songTitleKey: The song title key.
 \param songTitleValue: The song title value.
 \param clientConfiguration: AWS client configuration.
 \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::putItem(const Aws::String &tableName,
                                const Aws::String &artistKey,
                                const Aws::String &artistValue,
                                const Aws::String &albumTitleKey,
                                const Aws::String &albumTitleValue,
                                const Aws::String &awardsKey,
                                const Aws::String &awardsValue,
                                const Aws::String &songTitleKey,
                                const Aws::String &songTitleValue,
                                const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
```

```
Aws::DynamoDB::Model::PutItemRequest putItemRequest;
putItemRequest.SetTableName(tableName);

putItemRequest.AddItem(artistKey,
Aws::DynamoDB::Model::AttributeValue().Sets(
    artistValue)); // This is the hash key.
putItemRequest.AddItem(albumTitleKey,
Aws::DynamoDB::Model::AttributeValue().Sets(
    albumTitleValue));
putItemRequest.AddItem(awardsKey,
Aws::DynamoDB::Model::AttributeValue().Sets(awardsValue));
putItemRequest.AddItem(songTitleKey,
Aws::DynamoDB::Model::AttributeValue().Sets(songTitleValue));

const Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
    putItemRequest);
if (outcome.IsSuccess()) {
    std::cout << "Successfully added Item!" << std::endl;
}
else {
    std::cerr << outcome.GetError().GetMessage() << std::endl;
}

return outcome.IsSuccess();
}
```

- For API details, see [PutItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To add an item to a table

The following `put-item` example adds a new item to the *MusicCollection* table.

```
aws dynamodb put-item \
--table-name MusicCollection \
```

```
--item file://item.json \
--return-consumed-capacity TOTAL \
--return-item-collection-metrics SIZE
```

Contents of item.json:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Greatest Hits"}
}
```

Output:

```
{
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To conditionally overwrite an item in a table

The following put-item example overwrites an existing item in the MusicCollection table only if that existing item has an AlbumTitle attribute with a value of Greatest Hits. The command returns the previous value of the item.

```
aws dynamodb put-item \
```

```
--table-name MusicCollection \
--item file://item.json \
--condition-expression "#A = :A" \
--expression-attribute-names file://names.json \
--expression-attribute-values file://values.json \
--return-values ALL_OLD
```

Contents of item.json:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Somewhat Famous"}
}
```

Contents of names.json:

```
{
    "#A": "AlbumTitle"
}
```

Contents of values.json:

```
{
    ":A": {"S": "Greatest Hits"}
}
```

Output:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Greatest Hits"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
}
```

{}

If the key already exists, you should see the following output:

A client error (ConditionalCheckFailedException) occurred when calling the PutItem operation: The conditional request failed.

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [PutItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the examples.  
// It contains a DynamoDB service client that is used to act on the specified table.  
type TableBasics struct {  
    DynamoDbClient *dynamodb.Client  
    TableName      string  
}  
  
  
// AddMovie adds a movie to the DynamoDB table.  
func (basics TableBasics) AddMovie(movie Movie) error {  
    item, err := attributevalue.MarshalMap(movie)  
    if err != nil {  
        panic(err)  
    }  
    _, err = basics.DynamoDbClient.PutItem(context.TODO(), &dynamodb.PutItemInput{  
        TableName: aws.String(basics.TableName), Item: item,  
    })  
    return err  
}
```

```
)  
if err != nil {  
    log.Printf("Couldn't add item to table. Here's why: %v\n", err)  
}  
return err  
}  
  
  
// Movie encapsulates data about a movie. Title and Year are the composite  
primary key  
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition  
key,  
// and Info is additional data.  
type Movie struct {  
    Title string           `dynamodbav:"title"  
    Year  int               `dynamodbav:"year"  
    Info   map[string]interface{} `dynamodbav:"info"  
}  
  
// GetKey returns the composite primary key of the movie in a format that can be  
// sent to DynamoDB.  
func (movie Movie) GetKey() map[string]types.AttributeValue {  
    title, err := attributevalue.Marshal(movie.Title)  
    if err != nil {  
        panic(err)  
    }  
    year, err := attributevalue.Marshal(movie.Year)  
    if err != nil {  
        panic(err)  
    }  
    return map[string]types.AttributeValue{"title": title, "year": year}  
}  
  
// String returns the title, year, rating, and plot of a movie, formatted for the  
example.  
func (movie Movie) String() string {  
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
    movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])  
}
```

- For API details, see [PutItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Puts an item into a table using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key> <keyVal> <albumtitle> <albumtitleval>
            <awards> <awardsval> <Songtitle> <songtitleval>
        Where:
        """;
    }
}
```

```
        tableName - The Amazon DynamoDB table in which an item is
placed (for example, Music3).
        key - The key used in the Amazon DynamoDB table (for example,
Artist).
        keyval - The key value that represents the item to get (for
example, Famous Band).
        albumTitle - The Album title (for example, AlbumTitle).
        AlbumTitleValue - The name of the album (for example, Songs
About Life).
        Awards - The awards column (for example, Awards).
        AwardVal - The value of the awards (for example, 10).
        SongTitle - The song title (for example, SongTitle).
        SongTitleVal - The value of the song title (for example,
Happy Day).

**Warning** This program will place an item that you specify
into a table!
""";
```

```
if (args.length != 9) {
    System.out.println(usage);
    System.exit(1);
}

String tableName = args[0];
String key = args[1];
String keyVal = args[2];
String albumTitle = args[3];
String albumTitleValue = args[4];
String awards = args[5];
String awardVal = args[6];
String songTitle = args[7];
String songTitleVal = args[8];

Region region = Region.US_EAST_1;
DynamoDbClient ddb = DynamoDbClient.builder()
    .region(region)
    .build();

putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
    songTitleVal);
System.out.println("Done!");
ddb.close();
}
```

```
public static void putItemInTable(DynamoDbClient ddb,
        String tableName,
        String key,
        String keyVal,
        String albumTitle,
        String albumTitleValue,
        String awards,
        String awardVal,
        String songTitle,
        String songTitleVal) {

    HashMap<String,AttributeValue> itemValues = new HashMap<>();
    itemValues.put(key, AttributeValue.builder().s(keyVal).build());
    itemValues.put(songTitle,
        AttributeValue.builder().s(songTitleVal).build());
    itemValues.put(albumTitle,
        AttributeValue.builder().s(albumTitleValue).build());
    itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

    PutItemRequest request = PutItemRequest.builder()
        .tableName(tableName)
        .item(itemValues)
        .build();

    try {
        PutItemResponse response = ddb.putItem(request);
        System.out.println(tableName + " was successfully updated. The
request id is "
            + response.responseMetadata().requestId());

    } catch (ResourceNotFoundException e) {
        System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
        System.err.println("Be sure that it exists and that you've typed its
name correctly!");
        System.exit(1);
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
```

- For API details, see [PutItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [PutCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
      CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [PutItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Put an item in a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

Put an item in a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutItem](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun putItemInTable(
  tableNameVal: String,
  key: String,
  keyVal: String,
  albumTitle: String,
  albumTitleValue: String,
```

```
    awards: String,
    awardVal: String,
    songTitle: String,
    songTitleVal: String
) {
    val itemValues = mutableMapOf<String, AttributeValue>()

    // Add all content to the table.
    itemValues[key] = AttributeValue.S(keyVal)
    itemValues[songTitle] = AttributeValue.S(songTitleVal)
    itemValues[albumTitle] = AttributeValue.S(albumTitleValue)
    itemValues[awards] = AttributeValue.S(awardVal)

    val request = PutItemRequest {
        tableName = tableNameVal
        item = itemValues
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println(" A new item was placed into $tableNameVal.")
    }
}
```

- For API details, see [PutItem](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
echo "What's the name of the last movie you watched?\n";
while (empty($movieName)) {
    $movieName = testable_readline("Movie name: ");
}
echo "And what year was it released?\n";
```

```
$movieYear = "year";
while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
    $movieYear = testable_readline("Year released: ");
}

$service->putItem([
    'Item' => [
        'year' => [
            'N' => "$movieYear",
        ],
        'title' => [
            'S' => $movieName,
        ],
    ],
    'TableName' => $tableName,
]);
}

public function putItem(array $array)
{
    $this->dynamoDbClient->putItem($array);
}
```

- For API details, see [PutItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
```

```
self.dyn_resource = dyn_resource
# The table variable is set during the scenario in the call to
# 'exists' if the table exists. Otherwise, it is set by 'create_table'.
self.table = None

def add_movie(self, title, year, plot, rating):
    """
    Adds a movie to the table.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :param plot: The plot summary of the movie.
    :param rating: The quality rating of the movie.
    """
    try:
        self.table.put_item(
            Item={
                "year": year,
                "title": title,
                "info": {"plot": plot, "rating": Decimal(str(rating))},
            }
        )
    except ClientError as err:
        logger.error(
            "Couldn't add movie %s to table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [PutItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Adds a movie to the table.
    #
    # @param movie [Hash] The title, year, plot, and rating of the movie.
    def add_item(movie)
        @table.put_item(
            item: {
                "year" => movie[:year],
                "title" => movie[:title],
                "info" => {"plot" => movie[:plot], "rating" => movie[:rating]}))
    rescue Aws::DynamoDB::Errors::ServiceError => e
        puts("Couldn't add movie #{title} to table #{@table.name}. Here's why:")
        puts("\t#{e.code}: #{e.message}")
        raise
    end
end
```

- For API details, see [PutItem](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn add_item(client: &Client, item: Item, table: &String) ->
Result<ItemOut, Error> {
    let user_av = AttributeValue::S(item.username);
    let type_av = AttributeValue::S(item.p_type);
    let age_av = AttributeValue::S(item.age);
    let first_av = AttributeValue::S(item.first);
    let last_av = AttributeValue::S(item.last);

    let request = client
        .put_item()
        .table_name(table)
        .item("username", user_av)
        .item("account_type", type_av)
        .item("age", age_av)
        .item("first_name", first_av)
        .item("last_name", last_av);

    println!("Executing request [{request:?}] to add item...");

    let resp = request.send().await?;

    let attributes = resp.attributes().unwrap();

    let username = attributes.get("username").cloned();
    let first_name = attributes.get("first_name").cloned();
    let last_name = attributes.get("last_name").cloned();
    let age = attributes.get("age").cloned();
    let p_type = attributes.get("p_type").cloned();

    println!(
        "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
        username, first_name, last_name, age, p_type
    )
}
```

```
);

Ok(ItemOut {
    p_type,
    age,
    username,
    first_name,
    last_name,
})
}
```

- For API details, see [PutItem](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.
  DATA(lo_resp) = lo_dyn->putitem(
    iv_tablename = iv_table_name
    it_item      = it_item ).
  MESSAGE '1 row inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
  CATCH /aws1/cx_dyncondalcheckfaile00.
  MESSAGE 'A condition specified in the operation could not be evaluated.' TYPE 'E'.
  CATCH /aws1/cx_dyncodenotfoundex.
  MESSAGE 'The table or index does not exist' TYPE 'E'.
  CATCH /aws1/cx_dynamtransactconflictex.
  MESSAGE 'Another transaction is using the item' TYPE 'E'.
ENDTRY.
```

- For API details, see [PutItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
/// table.
///
/// - Parameter movie: The `Movie` to add to the table.
///
func add(movie: Movie) async throws {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    // Get a DynamoDB item containing the movie data.
    let item = try await movie.getAsItem()

    // Send the `PutItem` request to Amazon DynamoDB.

    let input = PutItemInput(
        item: item,
        tableName: self.tableName
    )
    _ = try await client.putItem(input: input)
}

///
/// Return an array mapping attribute names to Amazon DynamoDB attribute
/// values, representing the contents of the `Movie` record as a DynamoDB
```

```
/// item.  
///  
/// - Returns: The movie item as an array of type  
///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.  
///  
func getAsItem() async throws ->  
[Swift.String:DynamoDBClientTypes.AttributeValue] {  
    // Build the item record, starting with the year and title, which are  
    // always present.  
  
    var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [  
        "year": .n(String(self.year)),  
        "title": .s(self.title)  
    ]  
  
    // Add the `info` field with the rating and/or plot if they're  
    // available.  
  
    var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]  
    if (self.info.rating != nil || self.info.plot != nil) {  
        if self.info.rating != nil {  
            details["rating"] = .n(String(self.info.rating!))  
        }  
        if self.info.plot != nil {  
            details["plot"] = .s(self.info.plot!)  
        }  
    }  
    item["info"] = .m(details)  
  
    return item  
}
```

- For API details, see [PutItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table using an AWS SDK

The following code examples show how to query a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- [Accelerate reads with DAX](#)
- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public static async Task<int> QueryMoviesAsync(AmazonDynamoDBClient
client, string tableName, int year)
{
    var movieTable = Table.LoadTable(client, tableName);
    var filter = new QueryFilter("year", QueryOperator.Equal, year);

    Console.WriteLine("\nFind movies released in: {year}:");

    var config = new QueryOperationConfig()
    {
        Limit = 10, // 10 items per page.
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string>
        {
            "title",
```

```
        "year",
    },
    ConsistentRead = true,
    Filter = filter,
};

// Value used to track how many movies match the
// supplied criteria.
var moviesFound = 0;

Search search = movieTable.Query(config);
do
{
    var movieList = await search.GetNextSetAsync();
    moviesFound += movieList.Count;

    foreach (var movie in movieList)
    {
        DisplayDocument(movie);
    }
}
while (!search.IsDone);

return moviesFound;
}
```

- For API details, see [Query](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_query
```

```
#  
# This function queries a DynamoDB table.  
#  
# Parameters:  
#     -n table_name -- The name of the table.  
#     -k key_condition_expression -- The key condition expression.  
#     -a attribute_names -- Path to JSON file containing the attribute names.  
#     -v attribute_values -- Path to JSON file containing the attribute values.  
#     [-p projection_expression] -- Optional projection expression.  
#  
# Returns:  
#     The items as json output.  
# And:  
#     0 - If successful.  
#     1 - If it fails.  
#####  
function dynamodb_query() {  
    local table_name key_condition_expression attribute_names attribute_values  
    projection_expression response  
    local optionOPTARG # Required to use getopt command in a function.  
  
    # #####  
    # Function usage explanation  
    #####  
    function usage() {  
        echo "function dynamodb_query"  
        echo "Query a DynamoDB table."  
        echo " -n table_name -- The name of the table."  
        echo " -k key_condition_expression -- The key condition expression."  
        echo " -a attribute_names -- Path to JSON file containing the attribute  
        names."  
        echo " -v attribute_values -- Path to JSON file containing the attribute  
        values."  
        echo " [-p projection_expression] -- Optional projection expression."  
        echo ""  
    }  
  
    while getopt "n:k:a:v:p:h" option; do  
        case "${option}" in  
            n) table_name="${OPTARG}" ;;  
            k) key_condition_expression="${OPTARG}" ;;  
            a) attribute_names="${OPTARG}" ;;  
            v) attribute_values="${OPTARG}" ;;  
            p) projection_expression="${OPTARG}" ;;  
        esac  
    done  
}
```

```
h)
    usage
    return 0
    ;;
\?) 
    echo "Invalid parameter"
    usage
    return 1
    ;;
esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$key_condition_expression" ]]; then
    errecho "ERROR: You must provide a key condition expression with the -k parameter."
    usage
    return 1
fi

if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
fi

if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
fi

if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
        --table-name "$table_name" \
        --key-condition-expression "$key_condition_expression" \
        --expression-attribute-names file://"$attribute_names" \
        --expression-attribute-values file://"$attribute_values")
```

```
else
    response=$(aws dynamodb query \
        --table-name "$table_name" \
        --key-condition-expression "$key_condition_expression" \
        --expression-attribute-names file://"$attribute_names" \
        --expression-attribute-values file://"$attribute_values" \
        --projection-expression "$projection_expression")
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
fi

echo "$response"

return 0
}
```

The utility functions used in this example.

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
```

```
#           $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#           0: - Success.  
  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}  
}
```

- For API details, see [Query](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Perform a query on an Amazon DynamoDB Table and retrieve items.
```

```
/*!
 \sa queryItem()
 \param tableName: The table name.
 \param partitionKey: The partition key.
 \param partitionValue: The value for the partition key.
 \param projectionExpression: The projections expression, which is ignored if
 empty.
 \param clientConfiguration: AWS client configuration.
 \return bool: Function succeeded.
 */

/*
 * The partition key attribute is searched with the specified value. By default,
 all fields and values
 * contained in the item are returned. If an optional projection expression is
 * specified on the command line, only the specified fields and values are
 * returned.
 */
bool AwsDoc::DynamoDB::queryItems(const Aws::String &tableName,
                                    const Aws::String &partitionKey,
                                    const Aws::String &partitionValue,
                                    const Aws::String &projectionExpression,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::QueryRequest request;

    request.SetTableName(tableName);

    if (!projectionExpression.empty()) {
        request.SetProjectionExpression(projectionExpression);
    }

    // Set query key condition expression.
    request.SetKeyConditionExpression(partitionKey + "= :valueToMatch");

    // Set Expression AttributeValues.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> attributeValues;
    attributeValues.emplace(":valueToMatch", partitionValue);

    request.SetExpressionAttributeValues(attributeValues);

    bool result = true;
```

```
// "exclusiveStartKey" is used for pagination.
Aws::Map< Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
do {
    if (!exclusiveStartKey.empty()) {
        request.SetExclusiveStartKey(exclusiveStartKey);
        exclusiveStartKey.clear();
    }
    // Perform Query operation.
    const Aws::DynamoDB::Model::QueryOutcome &outcome =
dynamoClient.Query(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved items.
        const Aws::Vector< Aws::Map< Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
        if (!items.empty()) {
            std::cout << "Number of items retrieved from Query: " <<
items.size()
                << std::endl;
            // Iterate each item and print.
            for (const auto &item: items) {
                std::cout
                    <<
"*****"
                    << std::endl;
                // Output each retrieved field and its value.
                for (const auto &i: item)
                    std::cout << i.first << ":" << i.second.GetS() <<
std::endl;
            }
        }
        else {
            std::cout << "No item found in table: " << tableName <<
std::endl;
        }
    }

    exclusiveStartKey = outcome.GetResult().GetLastEvaluatedKey();
}
else {
    std::cerr << "Failed to Query items: " <<
outcome.GetError().GetMessage();
    result = false;
    break;
}
```

```
        }
    } while (!exclusiveStartKey.empty());

    return result;
}
```

- For API details, see [Query](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To query a table

The following query example queries items in the MusicCollection table. The table has a hash-and-range primary key (Artist and SongTitle), but this query only specifies the hash key value. It returns song titles by the artist named "No One You Know".

```
aws dynamodb query \
--table-name MusicCollection \
--projection-expression "SongTitle" \
--key-condition-expression "Artist = :v1" \
--expression-attribute-values file://expression-attributes.json \
--return-consumed-capacity TOTAL
```

Contents of expression-attributes.json:

```
{
  ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
  "Items": [
    {
      "SongTitle": {
        "S": "Call Me Today"
      },
      "SongTitle": {
        "S": "Scared of My Shadow"
      }
    }
  ]
}
```

```
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To query a table using strongly consistent reads and traverse the index in descending order

The following example performs the same query as the first example, but returns results in reverse order and uses strongly consistent reads.

```
aws dynamodb query \
--table-name MusicCollection \
--projection-expression "SongTitle" \
--key-condition-expression "Artist = :v1" \
--expression-attribute-values file://expression-attributes.json \
--consistent-read \
--no-scan-index-forward \
--return-consumed-capacity TOTAL
```

Contents of expression-attributes.json:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        }
    ]
}
```

```
        }
    },
    {
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
],
"Count": 2,
"ScannedCount": 2,
"ConsumedCapacity": {
    "TableName": "MusicCollection",
    "CapacityUnits": 1.0
}
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 3: To filter out specific results

The following example queries the MusicCollection but excludes results with specific values in the AlbumTitle attribute. Note that this does not affect the ScannedCount or ConsumedCapacity, because the filter is applied after the items have been read.

```
aws dynamodb query \
--table-name MusicCollection \
--key-condition-expression "#n1 = :v1" \
--filter-expression "NOT (#n2 IN (:v2, :v3))" \
--expression-attribute-names file://names.json \
--expression-attribute-values file://values.json \
--return-consumed-capacity TOTAL
```

Contents of values.json:

```
{
    ":v1": {"S": "No One You Know"},
    ":v2": {"S": "Blue Sky Blues"},
    ":v3": {"S": "Greatest Hits"}
}
```

Contents of names.json:

```
{  
    "#n1": "Artist",  
    "#n2": "AlbumTitle"  
}
```

Output:

```
{  
    "Items": [  
        {  
            "AlbumTitle": {  
                "S": "Somewhat Famous"  
            },  
            "Artist": {  
                "S": "No One You Know"  
            },  
            "SongTitle": {  
                "S": "Call Me Today"  
            }  
        }  
    ],  
    "Count": 1,  
    "ScannedCount": 2,  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 0.5  
    }  
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 4: To retrieve only an item count

The following example retrieves a count of items matching the query, but does not retrieve any of the items themselves.

```
aws dynamodb query \  
    --table-name MusicCollection \  
    --select COUNT \  
    --key-condition-expression "Artist = :v1" \  
    --expression-attribute-values file://expression-attributes.json
```

Contents of expression-attributes.json:

```
{  
    ":v1": {"S": "No One You Know"}  
}
```

Output:

```
{  
    "Count": 2,  
    "ScannedCount": 2,  
    "ConsumedCapacity": null  
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

Example 5: To query an index

The following example queries the local secondary index `AlbumTitleIndex`. The query returns all attributes from the base table that have been projected into the local secondary index. Note that when querying a local secondary index or global secondary index, you must also provide the name of the base table using the `table-name` parameter.

```
aws dynamodb query \  
    --table-name MusicCollection \  
    --index-name AlbumTitleIndex \  
    --key-condition-expression "Artist = :v1" \  
    --expression-attribute-values file://expression-attributes.json \  
    --select ALL_PROJECTED_ATTRIBUTES \  
    --return-consumed-capacity INDEXES
```

Contents of expression-attributes.json:

```
{  
    ":v1": {"S": "No One You Know"}  
}
```

Output:

```
{
```

```
"Items": [
    {
        "AlbumTitle": {
            "S": "Blue Sky Blues"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Scared of My Shadow"
        }
    },
    {
        "AlbumTitle": {
            "S": "Somewhat Famous"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
],
"Count": 2,
"ScannedCount": 2,
"ConsumedCapacity": {
    "TableName": "MusicCollection",
    "CapacityUnits": 0.5,
    "Table": {
        "CapacityUnits": 0.0
    },
    "LocalSecondaryIndexes": {
        "AlbumTitleIndex": {
            "CapacityUnits": 0.5
        }
    }
}
}
```

For more information, see [Working with Queries in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [Query](#) in *AWS CLI Command Reference*.

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// Query gets all movies in the DynamoDB table that were released in the
// specified year.
// The function uses the `expression` package to build the key condition
// expression
// that is used in the query.
func (basics TableBasics) Query(releaseYear int) ([]Movie, error) {
    var err error
    var response *dynamodb.QueryOutput
    var movies []Movie
    keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
    expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
    if err != nil {
        log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
    } else {
        queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
&dynamodb.QueryInput{
    TableName:           aws.String(basics.TableName),
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    KeyConditionExpression:   expr.KeyCondition(),
```

```
        })
    for queryPaginator.HasMorePages() {
        response, err = queryPaginator.NextPage(context.TODO())
        if err != nil {
            log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
releasYear, err)
            break
        } else {
            var moviePage []Movie
            err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
            if err != nil {
                log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
                break
            } else {
                movies = append(movies, moviePage...)
            }
        }
    }
    return movies, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
```

```
    panic(err)
}
return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [Query](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Queries a table by using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
*
* To query items from an Amazon DynamoDB table using the AWS SDK for Java V2,
* its better practice to use the
* Enhanced Client. See the EnhancedQueryRecords example.
*/
public class Query {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <partitionKeyName> <partitionKeyVal>
            Where:
            tableName - The Amazon DynamoDB table to put the item in (for
            example, Music3).
            partitionKeyName - The partition key name of the Amazon
            DynamoDB table (for example, Artist).
            partitionKeyVal - The value of the partition key that should
            match (for example, Famous Band).
            """;
        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }
        String tableName = args[0];
        String partitionKeyName = args[1];
        String partitionKeyVal = args[2];
        // For more information about an alias, see:
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
        Expressions.ExpressionAttributeNames.html
        String partitionAlias = "#a";
        System.out.format("Querying %s", tableName);
        System.out.println("");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
    }
}
```

```
        int count = queryTable(ddb, tableName, partitionKeyName, partitionKeyVal,
partitionAlias);
        System.out.println("There were " + count + " record(s) returned");
        ddb.close();
    }

    public static int queryTable(DynamoDbClient ddb, String tableName, String
partitionKeyName, String partitionKeyVal,
        String partitionAlias) {
        // Set up an alias for the partition key name in case it's a reserved
word.
        HashMap<String, String> attrNameAlias = new HashMap<String, String>();
        attrNameAlias.put(partitionAlias, partitionKeyName);

        // Set up mapping of the partition name with the value.
        HashMap<String, AttributeValue> attrValues = new HashMap<>();
        attrValues.put ":" + partitionKeyName, AttributeValue.builder()
            .s(partitionKeyVal)
            .build());

        QueryRequest queryReq = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(partitionAlias + " = :" +
partitionKeyName)
            .expressionAttributeNames(attrNameAlias)
            .expressionAttributeValues(attrValues)
            .build();

        try {
            QueryResponse response = ddb.query(queryReq);
            return response.count();

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return -1;
    }
}
```

Queries a table by using `DynamoDbClient` and a secondary index.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * Create the Movies table by running the Scenario example and loading the Movie
 * data from the JSON file. Next create a secondary
 * index for the Movies table that uses only the year column. Name the index
 * **year-index**. For more information, see:
 *
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
 */
public class QueryItemsUsingIndex {
    public static void main(String[] args) {
        String tableName = "Movies";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        queryIndex(ddb, tableName);
        ddb.close();
    }

    public static void queryIndex(DynamoDbClient ddb, String tableName) {
        try {
            Map<String, String> expressionAttributeNames = new HashMap<>();
            expressionAttributeNames.put("#year", "year");
            Map<String, AttributeValue> expressionAttributeValues = new
            HashMap<>();
        
```

```
        expressionAttributeValues.put(":yearValue",
AttributeValue.builder().n("2013").build());

QueryRequest request = QueryRequest.builder()
        .tableName(tableName)
        .indexName("year-index")
        .keyConditionExpression("#year = :yearValue")
        .expressionAttributeNames(expressionAttributesNames)
        .expressionAttributeValues(expressionAttributeValues)
        .build();

System.out.println("== Movie Titles ==");
QueryResponse response = ddb.query(request);
response.items()
        .forEach(movie ->
System.out.println(movie.get("title").s()));

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

- For API details, see [Query](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [QueryCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
    const command = new QueryCommand({  
        TableName: "CoffeeCrop",  
        KeyConditionExpression:  
            "OriginCountry = :originCountry AND RoastDate > :roastDate",  
        ExpressionAttributeValues: {  
            ":originCountry": "Ethiopia",  
            ":roastDate": "2023-05-01",  
        },  
        ConsistentRead: true,  
    });  
  
    const response = await docClient.send(command);  
    console.log(response);  
    return response;  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Query](#) in [AWS SDK for JavaScript API Reference](#).

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create DynamoDB document client  
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });  
  
var params = {
```

```
        ExpressionAttributeValues: {
          ":s": 2,
          ":e": 9,
          ":topic": "PHRASE",
        },
        KeyConditionExpression: "Season = :s and Episode > :e",
        FilterExpression: "contains (Subtitle, :topic)",
        TableName: "EPISODES_TABLE",
      };

      docClient.query(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else {
          console.log("Success", data.Items);
        }
      });
    });
  });
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Query](#) in [AWS SDK for JavaScript API Reference](#).

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun queryDynTable(
  tableNameVal: String,
  partitionKeyName: String,
  partitionKeyVal: String,
  partitionAlias: String
): Int {
  val attrNameAlias = mutableMapOf<String, String>()
  attrNameAlias[partitionAlias] = partitionKeyName
}
```

```
// Set up mapping of the partition name with the value.  
val attrValues = mutableMapOf<String, AttributeValue>()  
attrValues[":$partitionKeyName"] = AttributeValue.S(partitionKeyVal)  
  
val request = QueryRequest {  
    tableName = tableNameVal  
    keyConditionExpression = "$partitionAlias = :$partitionKeyName"  
    expressionAttributeNames = attrNameAlias  
    this.expressionAttributeValues = attrValues  
}  
  
DynamoDbClient { region = "us-east-1" }.use { ddb ->  
    val response = ddb.query(request)  
    return response.count  
}  
}
```

- For API details, see [Query](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$birthKey = [  
    'Key' => [  
        'year' => [  
            'N' => "$birthYear",  
        ],  
    ],  
];  
$result = $service->query($tableName, $birthKey);  
  
public function query(string $tableName, $key)  
{  
    $expressionAttributeValues = [];
```

```
$expressionAttributeNames = [];
$keyConditionExpression = "";
$index = 1;
foreach ($key as $name => $value) {
    $keyConditionExpression .= "#" . array_key_first($value) . " = :v" .
    $index ",";
    $expressionAttributeNames["#" . array_key_first($value)] =
    array_key_first($value);
    $hold = array_pop($value);
    $expressionAttributeValues[":v$index"] = [
        array_key_first($hold) => array_pop($hold),
    ];
}
$keyConditionExpression = substr($keyConditionExpression, 0, -1);
$query = [
    'ExpressionAttributeValues' => $expressionAttributeValues,
    'ExpressionAttributeNames' => $expressionAttributeNames,
    'KeyConditionExpression' => $keyConditionExpression,
    'TableName' => $tableName,
];
return $this->dynamoDbClient->query($query);
}
```

- For API details, see [Query](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Query items by using a key condition expression.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
```

```
"""
:param dyn_resource: A Boto3 DynamoDB resource.
"""

self.dyn_resource = dyn_resource
# The table variable is set during the scenario in the call to
# 'exists' if the table exists. Otherwise, it is set by 'create_table'.
self.table = None


def query_movies(self, year):
    """
    Queries for movies that were released in the specified year.

    :param year: The year to query.
    :return: The list of movies that were released in the specified year.
    """

    try:
        response =
self.table.query(KeyConditionExpression=Key("year").eq(year))
    except ClientError as err:
        logger.error(
            "Couldn't query for movies released in %s. Here's why: %s: %s",
            year,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Items"]
```

Query items and project them to return a subset of data.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def query_and_project_movies(self, year, title_bounds):
        """
        Query for movies that were released in a specified year and that have
titles
```

that start within a range of letters. A projection expression is used to return a subset of data for each movie.

```
:param year: The release year to query.
:param title_bounds: The range of starting letters to query.
:return: The list of movies.
"""
try:
    response = self.table.query(
        ProjectionExpression="#yr, title, info.genres, info.actors[0]",
        ExpressionAttributeNames={"#yr": "year"},
        KeyConditionExpression=(
            Key("year").eq(year)
            & Key("title").between(
                title_bounds["first"], title_bounds["second"]
            )
        ),
    )
except ClientError as err:
    if err.response["Error"]["Code"] == "ValidationException":
        logger.warning(
            "There's a validation error. Here's the message: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
    else:
        logger.error(
            "Couldn't query for movies. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
else:
    return response["Items"]
```

- For API details, see [Query](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Queries for movies that were released in the specified year.
    #
    # @param year [Integer] The year to query.
    # @return [Array] The list of movies that were released in the specified year.
    def query_items(year)
        response = @table.query(
            key_condition_expression: "#yr = :year",
            expression_attribute_names: {"#yr" => "year"},
            expression_attribute_values: {":year" => year})
        rescue Aws::DynamoDB::Errors::ServiceError => e
            puts("Couldn't query for movies released in #{year}. Here's why:")
            puts("\t#{e.code}: #{e.message}")
            raise
        else
            response.items
        end
    end
```

- For API details, see [Query](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Find the movies made in the specified year.

```
pub async fn movies_in_year(
    client: &Client,
    table_name: &str,
    year: u16,
) -> Result<Vec<Movie>, MovieError> {
    let results = client
        .query()
        .table_name(table_name)
        .key_condition_expression("#yr = :yyyy")
        .expression_attribute_names("#yr", "year")
        .expression_attribute_values(":yyyy",
AttributeValue::N(year.to_string())))
        .send()
        .await?;

    if let Some(items) = results.items {
        let movies = items.iter().map(|v| v.into()).collect();
        Ok(movies)
    } else {
        Ok(vec![])
    }
}
```

- For API details, see [Query](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
" Query movies for a given year .
DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
    ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_year }| ) ) ).
DATA(lt_key_conditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
    ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
        key = 'year'
        value = NEW /aws1/cl_dyncondition(
            it_attributevaluelist = lt_attributelist
            iv_comparisonoperator = |EQ|
        ) ) ) .
oo_result = lo_dyn->query(
    iv_tablename = iv_table_name
    it_keyconditions = lt_key_conditions ).
DATA(lt_items) = oo_result->get_items( ).
"You can loop over the results to get item attributes.
LOOP AT lt_items INTO DATA(lt_item).
    DATA(lo_title) = lt_item[ key = 'title' ]-value.
    DATA(lo_year) = lt_item[ key = 'year' ]-value.
ENDLOOP.
DATA(lv_count) = oo_result->get_count( ).
MESSAGE 'Item count is: ' && lv_count TYPE 'I'.
CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table or index does not exist' TYPE 'E'.
ENDTRY.
```

- For API details, see [Query](#) in AWS SDK for SAP ABAP API reference.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Get all the movies released in the specified year.  
///  
/// - Parameter year: The release year of the movies to return.  
///  
/// - Returns: An array of `Movie` objects describing each matching movie.  
  
func getMovies(fromYear year: Int) async throws -> [Movie] {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = QueryInput(  
        expressionAttributeNames: [  
            "#y": "year"  
        ],  
        expressionAttributeValues: [  
            ":y": .n(String(year))  
        ],  
        keyConditionExpression: "#y = :y",  
        tableName: self.tableName  
    )  
    let output = try await client.query(input: input)  
  
    guard let items = output.items else {  
        throw MoviesError.ItemNotFound  
    }  
    return items  
}
```

```
}

// Convert the found movies into `Movie` objects and return an array
// of them.

var movieList: [Movie] = []
for item in items {
    let movie = try Movie(withItem: item)
    movieList.append(movie)
}
return movieList
}
```

- For API details, see [Query](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Run a PartiQL statement on a DynamoDB table using an AWS SDK

The following code examples show how to run a PartiQL statement on a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Query a table using PartiQL](#)

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use an INSERT statement to add an item.

```
/// <summary>
/// Inserts a single movie into the movies table.
/// </summary>
/// <param name="tableName">The name of the table.</param>
/// <param name="movieTitle">The title of the movie to insert.</param>
/// <param name="year">The year that the movie was released.</param>
/// <returns>A Boolean value that indicates the success or failure of
/// the INSERT operation.</returns>
public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
{
    string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?, 'year': ?}}";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = insertBatch,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});
}

return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

Use a SELECT statement to get an item.

```
/// <summary>
/// Uses a PartiQL SELECT statement to retrieve a single movie from the
/// movie database.
/// </summary>
/// <param name="tableName">The name of the movie table.</param>
/// <param name="movieTitle">The title of the movie to retrieve.</param>
/// <returns>A list of movie data. If no movie matches the supplied
/// title, the list is empty.</returns>
```

```
public static async Task<List<Dictionary<string, AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
{
    string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
    var parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
    };

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
    {
        Statement = selectSingle,
        Parameters = parameters,
    });

    return response.Items;
}
```

Use a SELECT statement to get a list of items.

```
/// <summary>
/// Retrieve multiple movies by year using a SELECT statement.
/// </summary>
/// <param name="tableName">The name of the movie table.</param>
/// <param name="year">The year the movies were released.</param>
/// <returns></returns>
public static async Task<List<Dictionary<string, AttributeValue>>>
GetMovies(string tableName, int year)
{
    string selectSingle = $"SELECT * FROM {tableName} WHERE year = ?";
    var parameters = new List<AttributeValue>
    {
        new AttributeValue { N = year.ToString() },
    };

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
    {
        Statement = selectSingle,
```

```
        Parameters = parameters,
    });

    return response.Items;
}
```

Use an UPDATE statement to update an item.

```
/// <summary>
/// Updates a single movie in the table, adding information for the
/// producer.
/// </summary>
/// <param name="tableName">the name of the table.</param>
/// <param name="producer">The name of the producer.</param>
/// <param name="movieTitle">The movie title.</param>
/// <param name="year">The year the movie was released.</param>
/// <returns>A Boolean value that indicates the success of the
/// UPDATE operation.</returns>
public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
{
    string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = insertSingle,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = producer },
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});

    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

Use a DELETE statement to delete a single movie.

```
/// <summary>
/// Deletes a single movie from the table.
/// </summary>
/// <param name="tableName">The name of the table.</param>
/// <param name="movieTitle">The title of the movie to delete.</param>
/// <param name="year">The year that the movie was released.</param>
/// <returns>A Boolean value that indicates the success of the
/// DELETE operation.</returns>
public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
{
    var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = deleteSingle,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});
    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use an INSERT statement to add an item.

```
Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

// 2. Add a new movie using an "Insert" statement. (ExecuteStatement)
Aws::String title;
float rating;
int year;
Aws::String plot;
{
    title = askQuestion(
        "Enter the title of a movie you want to add to the table: ");
    year = askQuestionForInt("What year was it released? ");
    rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                    1, 10);
    plot = askQuestion("Summarize the plot for me: ");

    Aws::DynamoDB::Model::ExecuteStatementRequest request;
    std::stringstream sqlStream;
    sqlStream << "INSERT INTO \""
    << MOVIE_TABLE_NAME << "\" VALUE {\""
    << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
    << INFO_KEY << "': ?}";

    request.SetStatement(sqlStream.str());

    // Create the parameter attributes.
    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

    Aws::DynamoDB::Model::AttributeValue infoMapAttribute;
```

```
    std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
        Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
            ALLOCATION_TAG.c_str());
    ratingAttribute->SetN(rating);
    infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

    std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
        Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
            ALLOCATION_TAG.c_str());
    plotAttribute->SetS(plot);
    infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
    attributes.push_back(infoMapAttribute);
    request.SetParameters(attributes);

    Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
        dynamoClient.ExecuteStatement(
            request);

    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to add a movie: " <<
        outcome.GetError().GetMessage()
        << std::endl;
        return false;
    }
}
```

Use a SELECT statement to get an item.

```
// 3. Get the data for the movie using a "Select" statement.
(ExecuteStatement)
{
    Aws::DynamoDB::Model::ExecuteStatementRequest request;
    std::stringstream sqlStream;
    sqlStream << "SELECT * FROM \\" " << MOVIE_TABLE_NAME << "\\ WHERE "
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";

    request.SetStatement(sqlStream.str());

    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
    request.SetParameters(attributes);
```

```
Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
    request);

if (!outcome.IsSuccess()) {
    std::cerr << "Failed to retrieve movie information: "
        << outcome.GetError().GetMessage() << std::endl;
    return false;
}
else {
    // Print the retrieved movie information.
    const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

    const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

    if (items.size() == 1) {
        printMovieInfo(items[0]);
    }
    else {
        std::cerr << "Error: " << items.size() << " movies were
retrieved. "
            << " There should be only one movie." << std::endl;
    }
}
}
```

Use an UPDATE statement to update an item.

```
// 4. Update the data for the movie using an "Update" statement.
(ExecuteStatement)
{
    rating = askQuestionForFloatRange(
        Aws::String("\nLet's update your movie.\nYou rated it  ") +
        std::to_string(rating)
        + ", what new rating would you give it? ", 1, 10);

    Aws::DynamoDB::Model::ExecuteStatementRequest request;
    std::stringstream sqlStream;
    sqlStream << "UPDATE \""
        << MOVIE_TABLE_NAME << "\" SET "
```

```
<< INFO_KEY << "." << RATING_KEY << "=? WHERE "
<< TITLE_KEY << "=? AND " << YEAR_KEY << "=?";

request.SetStatement(sqlStream.str());

Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(rating));
attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

request.SetParameters(attributes);

Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
    request);

if (!outcome.IsSuccess()) {
    std::cerr << "Failed to update a movie: "
        << outcome.GetError().GetMessage();
    return false;
}
}
```

Use a DELETE statement to delete an item.

```
// 6. Delete the movie using a "Delete" statement. (ExecuteStatement)
{
    Aws::DynamoDB::Model::ExecuteStatementRequest request;
    std::stringstream sqlStream;
    sqlStream << "DELETE FROM  \"\" << MOVIE_TABLE_NAME << "\" WHERE "
        << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

    request.SetStatement(sqlStream.str());

    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
    request.SetParameters(attributes);

    Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
```

```
        request);
    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to delete the movie: "
            << outcome.GetError().GetMessage() << std::endl;
        return false;
    }
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use an INSERT statement to add an item.

```
// AddMovie runs a PartiQL INSERT statement to add a movie to the DynamoDB table.
func (runner PartiQLRunner) AddMovie(movie Movie) error {
    params, err := attributevalue.MarshalList([]interface{}{movie.Title, movie.Year,
        movie.Info})
    if err != nil {
        panic(err)
    }
    _, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
        &dynamodb.ExecuteStatementInput{
            Statement: aws.String(
                fmt.Sprintf("INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
                    runner.TableName)),
            Parameters: params,
        })
    if err != nil {
        log.Printf("Couldn't insert an item with PartiQL. Here's why: %v\n", err)
    }
    return err
}
```

```
}
```

Use a SELECT statement to get an item.

```
// GetMovie runs a PartiQL SELECT statement to get a movie from the DynamoDB
// table by
// title and year.
func (runner PartiQLRunner) GetMovie(title string, year int) (Movie, error) {
    var movie Movie
    params, err := attributevalue.MarshalList([]interface{}{title, year})
    if err != nil {
        panic(err)
    }
    response, err := runner.DynamoDbClient.ExecuteStatement(context.TODO(),
        &dynamodb.ExecuteStatementInput{
            Statement: aws.String(
                fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
                    runner.TableName)),
            Parameters: params,
        })
    if err != nil {
        log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
    } else {
        err = attributevalue.UnmarshalMap(response.Items[0], &movie)
        if err != nil {
            log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
        }
    }
    return movie, err
}
```

Use a SELECT statement to get a list of items and project the results.

```
// GetAllMovies runs a PartiQL SELECT statement to get all movies from the
// DynamoDB table.
// pageSize is not typically required and is used to show how to paginate the
// results.
```

```
// The results are projected to return only the title and rating of each movie.
func (runner PartiQLRunner) GetAllMovies(pageSize int32)
    ([]map[string]interface{}, error) {
    var output []map[string]interface{}
    var response *dynamodb.ExecuteStatementOutput
    var err error
    var nextToken *string
    for moreData := true; moreData; {
        response, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
        &dynamodb.ExecuteStatementInput{
            Statement: aws.String(
                fmt.Sprintf("SELECT title, info.rating FROM \"%v\"", runner.TableName)),
            Limit:     aws.Int32(pageSize),
            NextToken: nextToken,
        })
        if err != nil {
            log.Printf("Couldn't get movies. Here's why: %v\n", err)
            moreData = false
        } else {
            var pageOutput []map[string]interface{}
            err = attributevalue.UnmarshalListOfMaps(response.Items, &pageOutput)
            if err != nil {
                log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
            } else {
                log.Printf("Got a page of length %v.\n", len(response.Items))
                output = append(output, pageOutput...)
            }
            nextToken = response.NextToken
            moreData = nextToken != nil
        }
    }
    return output, err
}
```

Use an UPDATE statement to update an item.

```
// UpdateMovie runs a PartiQL UPDATE statement to update the rating of a movie
// that
// already exists in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovie(movie Movie, rating float64) error {
```

```
params, err := attributevalue.MarshalList([]interface{}{rating, movie.Title,
    movie.Year})
if err != nil {
    panic(err)
}
_, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
&dynamodb.ExecuteStatementInput{
    Statement: aws.String(
        fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
            runner.TableName)),
    Parameters: params,
})
if err != nil {
    log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
}
return err
}
```

Use a DELETE statement to delete an item.

```
// DeleteMovie runs a PartiQL DELETE statement to remove a movie from the
// DynamoDB table.
func (runner PartiQLRunner) DeleteMovie(movie Movie) error {
    params, err := attributevalue.MarshalList([]interface{}{movie.Title,
        movie.Year})
    if err != nil {
        panic(err)
    }
    _, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
        &dynamodb.ExecuteStatementInput{
            Statement: aws.String(
                fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
                    runner.TableName)),
            Parameters: params,
        })
    if err != nil {
        log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
            err)
    }
    return err
}
```

```
}
```

Define a Movie struct that is used in this example.

```
// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: `INSERT INTO Flowers value {'Name':?}`,
    Parameters: ["Rose"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Get an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
    const command = new ExecuteStatementCommand({  
        Statement: "SELECT * FROM CloudTypes WHERE IsStorm=?",  
        Parameters: [false],  
        ConsistentRead: true,  
    });  
  
    const response = await docClient.send(command);  
    console.log(response);  
    return response;  
};
```

Update an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";  
  
import {  
    ExecuteStatementCommand,  
    DynamoDBDocumentClient,  
} from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
    const command = new ExecuteStatementCommand({  
        Statement: "UPDATE EyeColors SET IsRecessive=? where Color=?",  
        Parameters: [true, "blue"],  
    });  
  
    const response = await docClient.send(command);  
    console.log(response);  
    return response;  
};
```

Delete an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "DELETE FROM PaintColors where Name=?",
    Parameters: ["Purple"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function insertItemByPartiQL(string $statement, array $parameters)
{
    $this->dynamoDbClient->executeStatement([
        'Statement' => "$statement",
        'Parameters' => $parameters,
    ]);
}
```

```
public function getItemByPartiQL(string $tableName, array $key): Result
{
    list($statement, $parameters) = $this->buildStatementAndParameters("SELECT", $tableName, $key['Item']);

    return $this->dynamoDbClient->executeStatement([
        'Parameters' => $parameters,
        'Statement' => $statement,
    ]);
}

public function updateItemByPartiQL(string $statement, array $parameters)
{
    $this->dynamoDbClient->executeStatement([
        'Statement' => $statement,
        'Parameters' => $parameters,
    ]);
}

public function deleteItemByPartiQL(string $statement, array $parameters)
{
    $this->dynamoDbClient->executeStatement([
        'Statement' => $statement,
        'Parameters' => $parameters,
    ]);
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class PartiQLWrapper:
```

```
"""
Encapsulates a DynamoDB resource to run PartiQL statements.
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource


def run_partiql(self, statement, params):
    """
    Runs a PartiQL statement. A Boto3 resource is used even though
    `execute_statement` is called on the underlying `client` object because
    the
        resource transforms input and output from plain old Python objects
    (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
    transforms yourself.

    :param statement: The PartiQL statement.
    :param params: The list of PartiQL parameters. These are applied to the
        statement in the order they are listed.
    :return: The items returned from the statement, if any.
    """

    try:
        output = self.dyn_resource.meta.client.execute_statement(
            Statement=statement, Parameters=params
        )
    except ClientError as err:
        if err.response["Error"]["Code"] == "ResourceNotFoundException":
            logger.error(
                "Couldn't execute PartiQL '%s' because the table does not
exist.",
                statement,
            )
        else:
            logger.error(
                "Couldn't execute PartiQL '%s'. Here's why: %s: %s",
                statement,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
    
```

```
        raise
    else:
        return output
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Select a single item using PartiQL.

```
class DynamoDBPartiQLSingle

    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamodb = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamodb.table(table_name)
    end

    # Gets a single record from a table using PartiQL.
    # Note: To perform more fine-grained selects,
    # use the Client.query instance method instead.
    #
    # @param title [String] The title of the movie to search.
    # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
    def select_item_by_title(title)
        request = {
            statement: "SELECT * FROM #{@table.name} WHERE title=?",
            parameters: [title]
```

```
    }
    @dynamodb.client.execute_statement(request)
end
```

Update a single item using PartiQL.

```
class DynamoDBPartiQLSingle

  attr_reader :dynamo_resource
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Updates a single record from a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @param rating [Float] The new rating to assign the title.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def update_rating_by_title(title, year, rating)
    request = {
      statement: "UPDATE \"#{@table.name}\" SET info.rating=? WHERE title=? and
year=?",
      parameters: [{ "N": rating }, title, year]
    }
    @dynamodb.client.execute_statement(request)
  end
```

Add a single item using PartiQL.

```
class DynamoDBPartiQLSingle

  attr_reader :dynamo_resource
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
```

```
@dynamodb = Aws::DynamoDB::Resource.new(client: client)
@table = @dynamodb.table(table_name)
end

# Adds a single record to a table using PartiQL.
#
# @param title [String] The title of the movie to update.
# @param year [Integer] The year the movie was released.
# @param plot [String] The plot of the movie.
# @param rating [Float] The new rating to assign the title.
# @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
def insert_item(title, year, plot, rating)
  request = {
    statement: "INSERT INTO \"#{@table.name}\" VALUE {'title': ?, 'year': ?, 'info': ?}",
    parameters: [title, year, {'plot': plot, 'rating': rating}]
  }
  @dynamodb.client.execute_statement(request)
end
```

Delete a single item using PartiQL.

```
class DynamoDBPartiQLSingle

attr_reader :dynamo_resource
attr_reader :table

def initialize(table_name)
  client = Aws::DynamoDB::Client.new(region: "us-east-1")
  @dynamodb = Aws::DynamoDB::Resource.new(client: client)
  @table = @dynamodb.table(table_name)
end

# Deletes a single record from a table using PartiQL.
#
# @param title [String] The title of the movie to update.
# @param year [Integer] The year the movie was released.
# @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
def delete_item_by_title(title, year)
  request = {
    statement: "DELETE FROM \"#{@table.name}\" WHERE title=? and year=?",
    parameters: [title, year]
```

```
    }
    @dynamodb.client.execute_statement(request)
end
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Run batches of PartiQL statements on a DynamoDB table using an AWS SDK

The following code examples show how to run batches of PartiQL statements on a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Query a table by using batches of PartiQL statements](#)

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use batches of INSERT statements to add items.

```
/// <summary>
/// Inserts movies imported from a JSON file into the movie table by
/// using an Amazon DynamoDB PartiQL INSERT statement.
/// </summary>
/// <param name="tableName">The name of the table into which the movie
/// information will be inserted.</param>
```

```
    /// <param name="movieFileName">The name of the JSON file that contains
    /// movie information.</param>
    /// <returns>A Boolean value that indicates the success or failure of
    /// the insert operation.</returns>
    public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
    {
        // Get the list of movies from the JSON file.
        var movies = ImportMovies(movieFileName);

        var success = false;

        if (movies is not null)
        {
            // Insert the movies in a batch using PartiQL. Because the
            // batch can contain a maximum of 25 items, insert 25 movies
            // at a time.
            string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
            var statements = new List<BatchStatementRequest>();

            try
            {
                for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                {
                    for (var i = indexOffset; i < indexOffset + 25; i++)
                    {
                        statements.Add(new BatchStatementRequest
                        {
                            Statement = insertBatch,
                            Parameters = new List<AttributeValue>
                            {
                                new AttributeValue { S = movies[i].Title },
                                new AttributeValue { N =
movies[i].Year.ToString() },
                            },
                        });
                    }
                }

                var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
{
    Statements = statements,

```

```
        });

        // Wait between batches for movies to be successfully
        added.
        System.Threading.Thread.Sleep(3000);

        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

        // Clear the list of statements for the next batch.
        statements.Clear();
    }
}

catch (AmazonDynamoDBException ex)
{
    Console.WriteLine(ex.Message);
}

return success;
}

/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The full path to the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public static List<Movie> ImportMovies(string movieFileName)
{
    if (!File.Exists(movieFileName))
    {
        return null!;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

    if (allMovies is not null)
    {
        // Return the first 250 entries.
        return allMovies.GetRange(0, 250);
    }
}
```

```
        else
        {
            return null!;
        }
    }
```

Use batches of SELECT statements to get items.

```
/// <summary>
/// Gets movies from the movie table by
/// using an Amazon DynamoDB PartiQL SELECT statement.
/// </summary>
/// <param name="tableName">The name of the table.</param>
/// <param name="title1">The title of the first movie.</param>
/// <param name="title2">The title of the second movie.</param>
/// <param name="year1">The year of the first movie.</param>
/// <param name="year2">The year of the second movie.</param>
/// <returns>True if successful.</returns>
public static async Task<bool> GetBatch(
    string tableName,
    string title1,
    string title2,
    int year1,
    int year2)
{
    var getBatch = $"SELECT FROM {tableName} WHERE title = ? AND year
= ?";
    var statements = new List<BatchStatementRequest>
    {
        new BatchStatementRequest
        {
            Statement = getBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = title1 },
                new AttributeValue { N = year1.ToString() },
            },
        },
        new BatchStatementRequest
        {
            Statement = getBatch,
```

```
        Parameters = new List<AttributeValue>
        {
            new AttributeValue { S = title2 },
            new AttributeValue { N = year2.ToString() },
        },
    }
};

var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
{
    Statements = statements,
});

if (response.Responses.Count > 0)
{
    response.Responses.ForEach(r =>
{
    Console.WriteLine($"{r.Item["title"]}\t{r.Item["year"]}");
});
    return true;
}
else
{
    Console.WriteLine($"Couldn't find either {title1} or {title2}.");
    return false;
}

}
```

Use batches of UPDATE statements to update items.

```
/// <summary>
/// Updates information for multiple movies.
/// </summary>
/// <param name="tableName">The name of the table containing the
/// movies to be updated.</param>
/// <param name="producer1">The producer name for the first movie
/// to update.</param>
/// <param name="title1">The title of the first movie.</param>
/// <param name="year1">The year that the first movie was released.</
param>
```

```
    ///>The producer name for the second movie to update.</param>
    ///>The title of the second movie.</param>
    ///>The year that the second movie was released.</param>
///>A Boolean value that indicates the success of the update.</returns>
public static async Task<bool> UpdateBatch(
    string tableName,
    string producer1,
    string title1,
    int year1,
    string producer2,
    string title2,
    int year2)
{
    string updateBatch = $"UPDATE {tableName} SET Producer=? WHERE title = ? AND year = ?";
    var statements = new List<BatchStatementRequest>
    {
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = producer1 },
                new AttributeValue { S = title1 },
                new AttributeValue { N = year1.ToString() },
            },
        },
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = producer2 },
                new AttributeValue { S = title2 },
                new AttributeValue { N = year2.ToString() },
            },
        }
    };
}
```

```
        var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
{
    Statements = statements,
});

return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

Use batches of DELETE statements to delete items.

```
/// <summary>
/// Deletes multiple movies using a PartiQL BatchExecuteAsync
/// statement.
/// </summary>
/// <param name="tableName">The name of the table containing the
/// moves that will be deleted.</param>
/// <param name="title1">The title of the first movie.</param>
/// <param name="year1">The year the first movie was released.</param>
/// <param name="title2">The title of the second movie.</param>
/// <param name="year2">The year the second movie was released.</param>
/// <returns>A Boolean value indicating the success of the operation.</
returns>
public static async Task<bool> DeleteBatch(
    string tableName,
    string title1,
    int year1,
    string title2,
    int year2)
{

    string updateBatch = $"DELETE FROM {tableName} WHERE title = ? AND
year = ?";
    var statements = new List<BatchStatementRequest>
    {
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = title1 },
                new AttributeValue { N = year1.ToString() },
            }
        }
    };
}
```

```
        },
    },

    new BatchStatementRequest
    {
        Statement = updateBatch,
        Parameters = new List<AttributeValue>
        {
            new AttributeValue { S = title2 },
            new AttributeValue { N = year2.ToString() },
        },
    }
};

var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
{
    Statements = statements,
});

return response.StatusCode == System.Net.HttpStatusCode.OK;
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use batches of INSERT statements to add items.

```
// 2. Add multiple movies using "Insert" statements. (BatchExecuteStatement)
Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

std::vector<Aws::String> titles;
```

```
    std::vector<float> ratings;
    std::vector<int> years;
    std::vector<Aws::String> plots;
    Aws::String doAgain = "n";
    do {
        Aws::String aTitle = askQuestion(
            "Enter the title of a movie you want to add to the table: ");
        titles.push_back(aTitle);
        int aYear = askQuestionForInt("What year was it released? ");
        years.push_back(aYear);
        float aRating = askQuestionForFloatRange(
            "On a scale of 1 - 10, how do you rate it? ",
            1, 10);
        ratings.push_back(aRating);
        Aws::String aPlot = askQuestion("Summarize the plot for me: ");
        plots.push_back(aPlot);

        doAgain = askQuestion(Aws::String("Would you like to add more movies? (y/n) "));
    } while (doAgain == "y");

    std::cout << "Adding " << titles.size()
        << (titles.size() == 1 ? " movie " : " movies ")
        << "to the table using a batch \"INSERT\" statement." << std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
            titles.size());

        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \""
            << MOVIE_TABLE_NAME << "\" VALUE {\""
            << TITLE_KEY << "': ?, '\" << YEAR_KEY << "': ?, '\""
            << INFO_KEY << "': ?\"}";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);

            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

            attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
        }
    }
}
```

```
// Create attribute for the info map.
Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute
= Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
    ALLOCATION_TAG.c_str());
ratingAttribute->SetN(ratings[i]);
infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
    ALLOCATION_TAG.c_str());
plotAttribute->SetS(plots[i]);
infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
attributes.push_back(infoMapAttribute);
statements[i].SetParameters(attributes);
}

Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

request.SetStatements(statements);

Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
    request);
if (!outcome.IsSuccess()) {
    std::cerr << "Failed to add the movies: " <<
outcome.GetError().GetMessage()
    << std::endl;
    return false;
}
}
```

Use batches of SELECT statements to get items.

```
// 3. Get the data for multiple movies using "Select" statements.
(BatchExecuteStatement)
{
    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
        titles.size());
    std::stringstream sqlStream;
```

```
sqlStream << "SELECT * FROM \\" " << MOVIE_TABLE_NAME << "\\" WHERE "
    << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

std::string sql(sqlStream.str());

for (size_t i = 0; i < statements.size(); ++i) {
    statements[i].SetStatement(sql);
    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
    attributes.push_back(
        Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
    statements[i].SetParameters(attributes);
}

Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

request.SetStatements(statements);

Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
    request);
if (outcome.IsSuccess()) {
    const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();

    const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

    for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
        const Aws::Map< Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

        printMovieInfo(item);
    }
}
else {
    std::cerr << "Failed to retrieve the movie information: "
        << outcome.GetError().GetMessage() << std::endl;
    return false;
}
}
```

Use batches of UPDATE statements to update items.

```
// 4. Update the data for multiple movies using "Update" statements.  
(BatchExecuteStatement)  
  
for (size_t i = 0; i < titles.size(); ++i) {  
    ratings[i] = askQuestionForFloatRange(  
        Aws::String("\nLet's update your the movie, \\"") + titles[i] +  
        ".\nYou rated it " + std::to_string(ratings[i])  
        + ", what new rating would you give it? ", 1, 10);  
}  
  
std::cout << "Updating the movie with a batch \"UPDATE\" statement." <<  
std::endl;  
  
{  
    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(  
        titles.size());  
  
    std::stringstream sqlStream;  
    sqlStream << "UPDATE \" " << MOVIE_TABLE_NAME << "\" SET "  
        << INFO_KEY << "." << RATING_KEY << "=? WHERE "  
        << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";  
  
    std::string sql(sqlStream.str());  
  
    for (size_t i = 0; i < statements.size(); ++i) {  
        statements[i].SetStatement(sql);  
  
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;  
        attributes.push_back(  
            Aws::DynamoDB::Model::AttributeValue().SetN(ratings[i]));  
        attributes.push_back(  
            Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));  
  
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));  
        statements[i].SetParameters(attributes);  
    }  
  
    Aws::DynamoDB::Model::BatchExecuteStatementRequest request;
```

```
    request.SetStatements(statements);
    Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
    request);
if (!outcome.IsSuccess()) {
    std::cerr << "Failed to update movie information: "
        << outcome.GetError().GetMessage() << std::endl;
    return false;
}
}
```

Use batches of DELETE statements to delete items.

```
// 6. Delete multiple movies using "Delete" statements.
(BatchExecuteStatement)
{
    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
        titles.size());
    std::stringstream sqlStream;
    sqlStream << "DELETE FROM  \\" " << MOVIE_TABLE_NAME << "\\" WHERE "
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";

    std::string sql(sqlStream.str());

    for (size_t i = 0; i < statements.size(); ++i) {
        statements[i].SetStatement(sql);
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(
            Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
        statements[i].SetParameters(attributes);
    }

    Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

    request.SetStatements(statements);

    Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
    request);
```

```
    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to delete the movies: "
        << outcome.GetError().GetMessage() << std::endl;
        return false;
    }
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use batches of INSERT statements to add items.

```
// AddMovieBatch runs a batch of PartiQL INSERT statements to add multiple movies
// to the
// DynamoDB table.
func (runner PartiQLRunner) AddMovieBatch(movies []Movie) error {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{movie.Title,
            movie.Year, movie.Info})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(fmt.Sprintf(
                "INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
                runner.TableName)),
            Parameters: params,
        }
    }
}
```

```
_ , err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
&dynamodb.BatchExecuteStatementInput{
    Statements: statementRequests,
})
if err != nil {
    log.Printf("Couldn't insert a batch of items with PartiQL. Here's why: %v\n",
err)
}
return err
}
```

Use batches of SELECT statements to get items.

```
// GetMovieBatch runs a batch of PartiQL SELECT statements to get multiple movies
// from
// the DynamoDB table by title and year.
func (runner PartiQLRunner) GetMovieBatch(movies []Movie) ([]Movie, error) {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{movie.Title,
        movie.Year})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(
                fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
runner.TableName)),
            Parameters: params,
        }
    }

    output, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
&dynamodb.BatchExecuteStatementInput{
    Statements: statementRequests,
})
    var outMovies []Movie
    if err != nil {
        log.Printf("Couldn't get a batch of items with PartiQL. Here's why: %v\n", err)
    }
}
```

```
    } else {
        for _, response := range output.Responses {
            var movie Movie
            err = attributevalue.UnmarshalMap(response.Item, &movie)
            if err != nil {
                log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
            } else {
                outMovies = append(outMovies, movie)
            }
        }
    }
    return outMovies, err
}
```

Use batches of UPDATE statements to update items.

```
// UpdateMovieBatch runs a batch of PartiQL UPDATE statements to update the
// rating of
// multiple movies that already exist in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovieBatch(movies []Movie, ratings []float64)
    error {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{ratings[index],
            movie.Title, movie.Year})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(
                fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
                    runner.TableName)),
            Parameters: params,
        }
    }

    _, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
        &dynamodb.BatchExecuteStatementInput{
            Statements: statementRequests,
        })
}
```

```
if err != nil {
    log.Printf("Couldn't update the batch of movies. Here's why: %v\n", err)
}
return err
}
```

Use batches of DELETE statements to delete items.

```
// DeleteMovieBatch runs a batch of PartiQL DELETE statements to remove multiple
// movies
// from the DynamoDB table.
func (runner PartiQLRunner) DeleteMovieBatch(movies []Movie) error {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{movie.Title,
            movie.Year})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(
                fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?", runner.TableName)),
            Parameters: params,
        }
    }

    _, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
        &dynamodb.BatchExecuteStatementInput{
            Statements: statementRequests,
        })
    if err != nil {
        log.Printf("Couldn't delete the batch of movies. Here's why: %v\n", err)
    }
    return err
}
```

Define a Movie struct that is used in this example.

```
// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const breakfastFoods = ["Eggs", "Bacon", "Sausage"];
  const command = new BatchExecuteStatementCommand({
    Statements: breakfastFoods.map((food) => ({
      Statement: `INSERT INTO BreakfastFoods value {'Name':?}`,
      Parameters: [food],
    })),
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Get a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
```

```
DynamoDBDocumentClient,  
BatchExecuteStatementCommand,  
} from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
  const command = new BatchExecuteStatementCommand({  
    Statements: [  
      {  
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",  
        Parameters: ["Teaspoons"],  
        ConsistentRead: true,  
      },  
      {  
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",  
        Parameters: ["Grams"],  
        ConsistentRead: true,  
      },  
    ],  
  });  
  
  const response = await docClient.send(command);  
  console.log(response);  
  return response;  
};
```

Update a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";  
  
import {  
  DynamoDBDocumentClient,  
  BatchExecuteStatementCommand,  
} from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
  const eggUpdates = [
```

```
        ["duck", "fried"],
        ["chicken", "omelette"],
    ];
const command = new BatchExecuteStatementCommand({
    Statements: eggUpdates.map((change) => ({
        Statement: "UPDATE Eggs SET Style=? where Variety=?",
        Parameters: [change[1], change[0]],
    })),
});
};

const response = await docClient.send(command);
console.log(response);
return response;
};
```

Delete a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
    DynamoDBDocumentClient,
    BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
    const command = new BatchExecuteStatementCommand({
        Statements: [
            {
                Statement: "DELETE FROM Flavors where Name=?",
                Parameters: ["Grape"],
            },
            {
                Statement: "DELETE FROM Flavors where Name=?",
                Parameters: ["Strawberry"],
            },
        ],
    });
};

const response = await docClient.send(command);
```

```
    console.log(response);
    return response;
};
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function getItemByPartiQLBatch(string $tableName, array $keys): Result
{
    $statements = [];
    foreach ($keys as $key) {
        list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);
        $statements[] = [
            'Statement' => "$statement",
            'Parameters' => $parameters,
        ];
    }

    return $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => $statements,
    ]);
}

public function insertItemByPartiQLBatch(string $statement, array
$parameters)
{
    $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => [
            [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ]
        ]
    ]);
}
```

```
        ],
    ],
]);
}

public function updateItemByPartiQLBatch(string $statement, array
$parameters)
{
    $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => [
            [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ],
        ],
    ]);
}

public function deleteItemByPartiQLBatch(string $statement, array
$parameters)
{
    $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => [
            [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ],
        ],
    ]);
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class PartiQLBatchWrapper:  
    """  
    Encapsulates a DynamoDB resource to run PartiQL statements.  
    """  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
  
  
    def run_partiql(self, statements, param_list):  
        """  
        Runs a PartiQL statement. A Boto3 resource is used even though  
        `execute_statement` is called on the underlying `client` object because  
        the  
            resource transforms input and output from plain old Python objects  
        (POPOs) to  
            the DynamoDB format. If you create the client directly, you must do these  
        transforms yourself.  
  
        :param statements: The batch of PartiQL statements.  
        :param param_list: The batch of PartiQL parameters that are associated  
        with  
            each statement. This list must be in the same order as  
        the  
            statements.  
        :return: The responses returned from running the statements, if any.  
        """  
        try:  
            output = self.dyn_resource.meta.client.batch_execute_statement(  
                Statements=[  
                    {"Statement": statement, "Parameters": params}  
                    for statement, params in zip(statements, param_list)  
                ]  
            )  
        except ClientError as err:  
            if err.response["Error"]["Code"] == "ResourceNotFoundException":  
                logger.error(  
                    "Couldn't execute batch of PartiQL statements because the  
                    table "  
                    "does not exist."  
                )
```

```
        )
    else:
        logger.error(
            "Couldn't execute batch of PartiQL statements. Here's why:
%s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
else:
    return output
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Read a batch of items using PartiQL.

```
class DynamoDBPartiQLBatch

    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamodb = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamodb.table(table_name)
    end

    # Selects a batch of items from a table using PartiQL
    #
    # @param batch_titles [Array] Collection of movie titles
```

```
# @return [Aws::DynamoDB::Types::BatchExecuteStatementOutput]
def batch_execute_select(batch_titles)
    request_items = batch_titles.map do |title, year|
        {
            statement: "SELECT * FROM #{@table.name} WHERE title=? and year=?",
            parameters: [title, year]
        }
    end
    @dynamodb.client.batch_execute_statement({statements: request_items})
end
```

Delete a batch of items using PartiQL.

```
class DynamoDBPartiQLBatch

    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamodb = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamodb.table(table_name)
    end

    # Deletes a batch of items from a table using PartiQL
    #
    # @param batch_titles [Array] Collection of movie titles
    # @return [Aws::DynamoDB::Types::BatchExecuteStatementOutput]
    def batch_execute_write(batch_titles)
        request_items = batch_titles.map do |title, year|
            {
                statement: "DELETE FROM #{@table.name} WHERE title=? and year=?",
                parameters: [title, year]
            }
        end
        @dynamodb.client.batch_execute_statement({statements: request_items})
    end
end
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Scan a DynamoDB table using an AWS SDK

The following code examples show how to scan a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- [Accelerate reads with DAX](#)
- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public static async Task<int> ScanTableAsync(
    AmazonDynamoDBClient client,
    string tableName,
    int startYear,
    int endYear)
{
    var request = new ScanRequest
    {
        TableName = tableName,
        ExpressionAttributeNames = new Dictionary<string, string>
        {
            { "#yr", "year" },
        },
        ExpressionAttributeValues = new Dictionary<string,
        AttributeValue>
        {
            { ":y_a", new AttributeValue { N = startYear.ToString() } },
        }
    };
    var response = await client.ScanAsync(request);
    return response.Count;
}
```

```
        { ":y_z", new AttributeValue { N = endYear.ToString() } },
    },
    FilterExpression = "#yr between :y_a and :y_z",
    ProjectionExpression = "#yr, title, info.actors[0],
info.directors, info.running_time_secs",
    Limit = 10 // Set a limit to demonstrate using the
LastEvaluatedKey.
};

// Keep track of how many movies were found.
int foundCount = 0;

var response = new ScanResponse();
do
{
    response = await client.ScanAsync(request);
    foundCount += response.Items.Count;
    response.Items.ForEach(i => DisplayItem(i));
    request.ExclusiveStartKey = response.LastEvaluatedKey;
}
while (response.LastEvaluatedKey.Count > 0);
return foundCount;
}
```

- For API details, see [Scan](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_scan
#
# This function scans a DynamoDB table.
```

```
#  
# Parameters:  
#     -n table_name -- The name of the table.  
#     -f filter_expression -- The filter expression.  
#     -a expression_attribute_names -- Path to JSON file containing the  
#         expression attribute names.  
#     -v expression_attribute_values -- Path to JSON file containing the  
#         expression attribute values.  
#     [-p projection_expression] -- Optional projection expression.  
#  
# Returns:  
#     The items as json output.  
# And:  
#     0 - If successful.  
#     1 - If it fails.  
#####  
function dynamodb_scan() {  
    local table_name filter_expression expression_attribute_names  
    expression_attribute_values projection_expression response  
    local optionOPTARG # Required to use getopt command in a function.  
  
    # #####  
    # Function usage explanation  
    #####  
    function usage() {  
        echo "function dynamodb_scan"  
        echo "Scan a DynamoDB table."  
        echo " -n table_name -- The name of the table."  
        echo " -f filter_expression -- The filter expression."  
        echo " -a expression_attribute_names -- Path to JSON file containing the  
        #         expression attribute names."  
        echo " -v expression_attribute_values -- Path to JSON file containing the  
        #         expression attribute values."  
        echo " [-p projection_expression] -- Optional projection expression."  
        echo ""  
    }  
  
    while getopt "n:f:a:v:p:h" option; do  
        case "${option}" in  
            n) table_name="${OPTARG}" ;;  
            f) filter_expression="${OPTARG}" ;;  
            a) expression_attribute_names="${OPTARG}" ;;  
            v) expression_attribute_values="${OPTARG}" ;;  
            p) projection_expression="${OPTARG}" ;;  
        esac  
    done  
}
```

```
h)
  usage
  return 0
;;
\?) 
  echo "Invalid parameter"
  usage
  return 1
;;
esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$filter_expression" ]]; then
  errecho "ERROR: You must provide a filter expression with the -f parameter."
  usage
  return 1
fi

if [[ -z "$expression_attribute_names" ]]; then
  errecho "ERROR: You must provide expression attribute names with the -a parameter."
  usage
  return 1
fi

if [[ -z "$expression_attribute_values" ]]; then
  errecho "ERROR: You must provide expression attribute values with the -v parameter."
  usage
  return 1
fi

if [[ -z "$projection_expression" ]]; then
  response=$(aws dynamodb scan \
    --table-name "$table_name" \
    --filter-expression "$filter_expression" \
    --expression-attribute-names file://"$expression_attribute_names" \
```

```
--expression-attribute-values file://"$expression_attribute_values")  
else  
    response=$(aws dynamodb scan \  
        --table-name "$table_name" \  
        --filter-expression "$filter_expression" \  
        --expression-attribute-names file://"$expression_attribute_names" \  
        --expression-attribute-values file://"$expression_attribute_values" \  
        --projection-expression "$projection_expression")  
fi  
  
local error_code=${?}  
  
if [[ $error_code -ne 0 ]]; then  
    aws_cli_error_log $error_code  
    errecho "ERROR: AWS reports scan operation failed.$response"  
    return 1  
fi  
  
echo "$response"  
  
return 0  
}
```

The utility functions used in this example.

```
#####
# function errecho  
#  
# This function outputs everything sent to it to STDERR (standard error output).  
#####  
function errecho() {  
    printf "%s\n" "$*" 1>&2  
}  
  
#####
# function aws_cli_error_log()  
#  
# This function is used to log the error messages from the AWS CLI.  
#  
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.  
#
```

```
# The function expects the following argument:  
#           $1 - The error code returned by the AWS CLI.  
#  
# Returns:  
#           0: - Success.  
#  
#####  
function aws_cli_error_log() {  
    local err_code=$1  
    errecho "Error code : $err_code"  
    if [ "$err_code" == 1 ]; then  
        errecho " One or more S3 transfers failed."  
    elif [ "$err_code" == 2 ]; then  
        errecho " Command line failed to parse."  
    elif [ "$err_code" == 130 ]; then  
        errecho " Process received SIGINT."  
    elif [ "$err_code" == 252 ]; then  
        errecho " Command syntax invalid."  
    elif [ "$err_code" == 253 ]; then  
        errecho " The system environment or configuration was invalid."  
    elif [ "$err_code" == 254 ]; then  
        errecho " The service returned an error."  
    elif [ "$err_code" == 255 ]; then  
        errecho " 255 is a catch-all error."  
    fi  
  
    return 0  
}  
#####
```

- For API details, see [Scan](#) in *AWS CLI Command Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Scan an Amazon DynamoDB table.  
/*!  
 \sa scanTable()  
 \param tableName: Name for the DynamoDB table.  
 \param projectionExpression: An optional projection expression, ignored if  
 empty.  
 \param clientConfiguration: AWS client configuration.  
 \return bool: Function succeeded.  
 */  
  
bool AwsDoc::DynamoDB::scanTable(const Aws::String &tableName,  
                                  const Aws::String &projectionExpression,  
                                  const Aws::Client::ClientConfiguration  
&clientConfiguration) {  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
    Aws::DynamoDB::Model::ScanRequest request;  
    request.SetTableName(tableName);  
  
    if (!projectionExpression.empty())  
        request.SetProjectionExpression(projectionExpression);  
  
    // Perform scan on table.  
    const Aws::DynamoDB::Model::ScanOutcome &outcome =  
dynamoClient.Scan(request);  
    if (outcome.IsSuccess()) {  
        // Reference the retrieved items.  
        const Aws::Vector<Aws::Map<Aws::String,  
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();  
        if (!items.empty()) {  
            std::cout << "Number of items retrieved from scan: " << items.size()  
                << std::endl;  
            // Iterate each item and print.  
            for (const Aws::Map<Aws::String,  
Aws::DynamoDB::Model::AttributeValue> &itemMap: items) {  
                std::cout <<  
"*****"  
                << std::endl;  
                // Output each retrieved field and its value.  
                for (const auto &itemEntry: itemMap)  
                    std::cout << itemEntry.first << ":" <<  
itemEntry.second.GetS()  
                        << std::endl;  
            }  
    }
```

```
    }

    else {
        std::cout << "No item found in table: " << tableName << std::endl;
    }
}

else {
    std::cerr << "Failed to Scan items: " << outcome.GetError().GetMessage()
        << std::endl;
}

return outcome.IsSuccess();
}
```

- For API details, see [Scan](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To scan a table

The following scan example scans the entire MusicCollection table, and then narrows the results to songs by the artist "No One You Know". For each item, only the album title and song title are returned.

```
aws dynamodb scan \
--table-name MusicCollection \
--filter-expression "Artist = :a" \
--projection-expression "#ST, #AT" \
--expression-attribute-names file://expression-attribute-names.json \
--expression-attribute-values file://expression-attribute-values.json
```

Contents of expression-attribute-names.json:

```
{
    "#ST": "SongTitle",
    "#AT": "AlbumTitle"
}
```

Contents of expression-attribute-values.json:

```
{  
    ":a": {"S": "No One You Know"}  
}
```

Output:

```
{  
    "Count": 2,  
    "Items": [  
        {  
            "SongTitle": {  
                "S": "Call Me Today"  
            },  
            "AlbumTitle": {  
                "S": "Somewhat Famous"  
            }  
        },  
        {  
            "SongTitle": {  
                "S": "Scared of My Shadow"  
            },  
            "AlbumTitle": {  
                "S": "Blue Sky Blues"  
            }  
        }  
    "ScannedCount": 3,  
    "ConsumedCapacity": null  
}
```

For more information, see [Working with Scans in DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [Scan](#) in *AWS CLI Command Reference*.

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// Scan gets all movies in the DynamoDB table that were released in a range of
// years
// and projects them to return a reduced set of fields.
// The function uses the `expression` package to build the filter and projection
// expressions.
func (basics TableBasics) Scan(startYear int, endYear int) ([]Movie, error) {
    var movies []Movie
    var err error
    var response *dynamodb.ScanOutput
    filtEx := expression.Name("year").Between(expression.Value(startYear),
        expression.Value(endYear))
    projEx := expression.NamesList(
        expression.Name("year"), expression.Name("title"),
        expression.Name("info.rating"))
    expr, err :=
        expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
    if err != nil {
        log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
    } else {
```

```
scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
&dynamodb.ScanInput{
    TableName:                 aws.String(basics.TableName),
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    FilterExpression:         expr.Filter(),
    ProjectionExpression:    expr.Projection(),
})
for scanPaginator.HasMorePages() {
    response, err = scanPaginator.NextPage(context.TODO())
    if err != nil {
        log.Printf("Couldn't scan for movies released between %v and %v. Here's why: %v\n",
            startYear, endYear, err)
        break
    } else {
        var moviePage []Movie
        err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
        if err != nil {
            log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
            break
        } else {
            movies = append(movies, moviePage...)
        }
    }
}
return movies, err
}
```

```
// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
```

```
// sent to DynamoDB.  
func (movie Movie) GetKey() map[string]types.AttributeValue {  
    title, err := attributevalue.Marshal(movie.Title)  
    if err != nil {  
        panic(err)  
    }  
    year, err := attributevalue.Marshal(movie.Year)  
    if err != nil {  
        panic(err)  
    }  
    return map[string]types.AttributeValue{"title": title, "year": year}  
}  
  
// String returns the title, year, rating, and plot of a movie, formatted for the  
// example.  
func (movie Movie) String() string {  
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])  
}
```

- For API details, see [Scan](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Scans an Amazon DynamoDB table using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;  
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;  
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;  
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;  
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;  
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
```

```
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To scan items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, See the EnhancedScanRecords example.
 */

public class DynamoDBScanItems {
    public static void main(String[] args) {

        final String usage = """

            Usage:
            <tableName>

            Where:
            tableName - The Amazon DynamoDB table to get information from
            (for example, Music3).
            """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        scanItems(ddb, tableName);
        ddb.close();
    }
}
```

```
public static void scanItems(DynamoDbClient ddb, String tableName) {
    try {
        ScanRequest scanRequest = ScanRequest.builder()
            .tableName(tableName)
            .build();

        ScanResponse response = ddb.scan(scanRequest);
        for (Map<String, AttributeValue> item : response.items()) {
            Set<String> keys = item.keySet();
            for (String key : keys) {
                System.out.println("The key name is " + key + "\n");
                System.out.println("The value is " + item.get(key).s());
            }
        }
    } catch (DynamoDbException e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

- For API details, see [Scan](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [ScanCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, ScanCommand } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
    const command = new ScanCommand({  
        ProjectionExpression: "#Name, Color, AvgLifeSpan",  
        ExpressionAttributeNames: { "#Name": "Name" },  
        TableName: "Birds",  
    });  
  
    const response = await docClient.send(command);  
    for (const bird of response.Items) {  
        console.log(`#${bird.Name} - (${bird.Color}, ${bird.AvgLifeSpan})`);  
    }  
    return response;  
};
```

- For API details, see [Scan](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js.  
var AWS = require("aws-sdk");  
// Set the AWS Region.  
AWS.config.update({ region: "REGION" });  
  
// Create DynamoDB service object.  
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });  
  
const params = {  
    // Specify which items in the results are returned.  
    FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",  
    // Define the expression attribute value, which are substitutes for the values  
    // you want to compare.  
    ExpressionAttributeValues: {
```

```
":topic": { S: "SubTitle2" },
":s": { N: 1 },
":e": { N: 2 },
},
// Set the projection expression, which are the attributes that you want.
ProjectionExpression: "Season, Episode, Title, Subtitle",
TableName: "EPISODES_TABLE",
};

ddb.scan(params, function (err, data) {
if (err) {
  console.log("Error", err);
} else {
  console.log("Success", data);
  data.Items.forEach(function (element, index, array) {
    console.log(
      "printing",
      element.Title.S + " (" + element.Subtitle.S + ")"
    );
  });
}
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Scan](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun scanItems(tableNameVal: String) {
  val request = ScanRequest {
    tableName = tableNameVal
```

```
}

DynamoDbClient { region = "us-east-1" }.use { ddb ->
    val response = ddb.scan(request)
    response.items?.forEach { item ->
        item.keys.forEach { key ->
            println("The key name is $key\n")
            println("The value is ${item[key]}")
        }
    }
}
```

- For API details, see [Scan](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
$yearsKey = [
    'Key' => [
        'year' => [
            'N' => [
                'minRange' => 1990,
                'maxRange' => 1999,
            ],
        ],
    ],
];
$filter = "year between 1990 and 1999";
echo "\nHere's a list of all the movies released in the 90s:\n";
$result = $service->scan($tableName, $yearsKey, $filter);
foreach ($result['Items'] as $movie) {
    $movie = $marshal->unmarshalItem($movie);
    echo $movie['title'] . "\n";
```

```
}

public function scan(string $tableName, array $key, string $filters)
{
    $query = [
        'ExpressionAttributeNames' => ['#year' => 'year'],
        'ExpressionAttributeValues' => [
            ':min' => ['N' => '1990'],
            ':max' => ['N' => '1999'],
        ],
        'FilterExpression' => "#year between :min and :max",
        'TableName' => $tableName,
    ];
    return $this->dynamoDbClient->scan($query);
}
```

- For API details, see [Scan](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
def scan_movies(self, year_range):
    """
    Scans for movies that were released in a range of years.
    Uses a projection expression to return a subset of data for each movie.

    :param year_range: The range of years to retrieve.
    :return: The list of movies released in the specified years.
    """

    movies = []
    scan_kwargs = {
        "FilterExpression": Key("year").between(
            year_range["first"], year_range["second"]
        ),
        "ProjectionExpression": "#yr, title, info.rating",
        "ExpressionAttributeNames": {"#yr": "year"},
    }
    try:
        done = False
        start_key = None
        while not done:
            if start_key:
                scan_kwargs["ExclusiveStartKey"] = start_key
            response = self.table.scan(**scan_kwargs)
            movies.extend(response.get("Items", []))
            start_key = response.get("LastEvaluatedKey", None)
            done = start_key is None
    except ClientError as err:
        logger.error(
            "Couldn't scan for movies. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise

    return movies
```

- For API details, see [Scan](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Scans for movies that were released in a range of years.
    # Uses a projection expression to return a subset of data for each movie.
    #
    # @param year_range [Hash] The range of years to retrieve.
    # @return [Array] The list of movies released in the specified years.
    def scan_items(year_range)
        movies = []
        scan_hash = {
            filter_expression: "#yr between :start_yr and :end_yr",
            projection_expression: "#yr, title, info.rating",
            expression_attribute_names: {"#yr" => "year"},
            expression_attribute_values: {
                ":start_yr" => year_range[:start], ":end_yr" => year_range[:end]}
        }
        done = false
        start_key = nil
        until done
            scan_hash[:exclusive_start_key] = start_key unless start_key.nil?
            response = @table.scan(scan_hash)
            movies.concat(response.items) unless response.items.empty?
            start_key = response.last_evaluated_key
            done = start_key.nil?
        end
    end
end
```

```
    end
rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't scan for movies. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
else
    movies
end
```

- For API details, see [Scan](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
pub async fn list_items(client: &Client, table: &str, page_size: Option<i32>) ->
Result<(), Error> {
    let page_size = page_size.unwrap_or(10);
    let items: Result<Vec<_>, _> = client
        .scan()
        .table_name(table)
        .limit(page_size)
        .into_paginator()
        .items()
        .send()
        .collect()
        .await;

    println!("Items in table (up to {page_size}):");
    for item in items? {
        println!("    {:?}", item);
    }
}

Ok(())
}
```

- For API details, see [Scan](#) in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TRY.

```
" Scan movies for rating greater than or equal to the rating specified
DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevalueelist(
    ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_rating }| ) ) .
DATA(lt_filter_conditions) = VALUE /aws1/
cl_dyncondition=>tt_filterconditionmap(
    ( VALUE /aws1/cl_dyncondition=>ts_filterconditionmap_maprow(
        key = 'rating'
        value = NEW /aws1/cl_dyncondition(
            it_attributevalueelist = lt_attributelist
            iv_comparisonoperator = |GE|
        ) ) ) .
oo_scan_result = lo_dyn->scan( iv_tablename = iv_table_name
    it_scanfilter = lt_filter_conditions ).
DATA(lt_items) = oo_scan_result->get_items( ).
LOOP AT lt_items INTO DATA(lo_item).
    " You can loop over to get individual attributes.
    DATA(lo_title) = lo_item[ key = 'title' ]-value.
    DATA(lo_year) = lo_item[ key = 'year' ]-value.
ENDLOOP.
DATA(lv_count) = oo_scan_result->get_count( ).
MESSAGE 'Found ' && lv_count && ' items' TYPE 'I'.
CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table or index does not exist' TYPE 'E'.
ENDTRY.
```

- For API details, see [Scan](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Return an array of `Movie` objects released in the specified range of
/// years.
///
/// - Parameters:
///   - firstYear: The first year of movies to return.
///   - lastYear: The last year of movies to return.
///   - startKey: A starting point to resume processing; always use `nil`.
///
/// - Returns: An array of `Movie` objects describing the matching movies.
///
/// > Note: The `startKey` parameter is used by this function when
/// recursively calling itself, and should always be `nil` when calling
/// directly.
///
func getMovies(firstYear: Int, lastYear: Int,
               startKey: [Swift.String:DynamoDBClientTypes.AttributeValue]? = nil)
    async throws -> [Movie] {
    var movieList: [Movie] = []

    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }
```

```
let input = ScanInput(  
    consistentRead: true,  
    exclusiveStartKey: startKey,  
    expressionAttributeNames: [  
        "#y": "year"           // `year` is a reserved word, so use `#y`  
    instead.  
    ],  
    expressionAttributeValues: [  
        ":y1": .n(String(firstYear)),  
        ":y2": .n(String(lastYear))  
    ],  
    filterExpression: "#y BETWEEN :y1 AND :y2",  
    tableName: self.tableName  
)  
  
let output = try await client.scan(input: input)  
  
guard let items = output.items else {  
    return movieList  
}  
  
// Build an array of `Movie` objects for the returned items.  
  
for item in items {  
    let movie = try Movie(withItem: item)  
    movieList.append(movie)  
}  
  
// Call this function recursively to continue collecting matching  
// movies, if necessary.  
  
if output.lastEvaluatedKey != nil {  
    let movies = try await self.getMovies(firstYear: firstYear, lastYear:  
lastYear,  
                                         startKey: output.lastEvaluatedKey)  
    movieList += movies  
}  
return movieList  
}
```

- For API details, see [Scan](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Update an item in a DynamoDB table using an AWS SDK

The following code examples show how to update an item in a DynamoDB table.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <summary>
/// Updates an existing item in the movies table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="newMovie">A Movie object containing information for the movie to update.</param>
/// <param name="newInfo">A MovieInfo object that contains the information that will be changed.</param>
/// <param name="tableName">The name of the table that contains the movie.</param>
/// <returns>A Boolean value that indicates the success of the operation.</returns>
public static async Task<bool> UpdateItemAsync(
    AmazonDynamoDBClient client,
    Movie newMovie,
    MovieInfo newInfo,
```

```
        string tableName)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };
    var updates = new Dictionary<string, AttributeValueUpdate>
    {
        ["info.plot"] = new AttributeValueUpdate
        {
            Action = AttributeAction.PUT,
            Value = new AttributeValue { S = newInfo.Plot },
        },
        ["info.rating"] = new AttributeValueUpdate
        {
            Action = AttributeAction.PUT,
            Value = new AttributeValue { N = newInfo.Rank.ToString() },
        },
    };

    var request = new UpdateItemRequest
    {
        AttributeUpdates = updates,
        Key = key,
        TableName = tableName,
    };

    var response = await client.UpdateItemAsync(request);

    return response.HttpStatusCode == System.Net HttpStatusCode.OK;
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to update.
#     -e update_expression -- An expression that defines one or more
#                           attributes to be updated.
#     -v values -- Path to json file containing the update values.
#
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_update_item() {
    local table_name keys update_expression values response
    local option OPTARG # Required to use getopts command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_update_item"
    echo "Update an item in a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -k keys -- Path to json file containing the keys that identify the
#               item to update."
```

```
echo " -e update expression -- An expression that defines one or more
attributes to be updated."
echo " -v values -- Path to json file containing the update values."
echo ""
}

while getopts "n:k:e:v:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    k) keys="${OPTARG}" ;;
    e) update_expression="${OPTARG}" ;;
    v) values="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?) 
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi
if [[ -z "$update_expression" ]]; then
  errecho "ERROR: You must provide an update expression with the -e parameter."
  usage
  return 1
fi

if [[ -z "$values" ]]; then
  errecho "ERROR: You must provide a values json file path the -v parameter."
```

```
usage
return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    keys: $keys"
iecho "    update_expression: $update_expression"
iecho "    values: $values"

response=$(aws dynamodb update-item \
--table-name "$table_name" \
--key file://"$keys" \
--update-expression "$update_expression" \
--expression-attribute-values file://"$values")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
fi

return 0

}
```

The utility functions used in this example.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}
```

```
#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#       $1 - The error code returned by the AWS CLI.
#
# Returns:
#       0: - Success.
#
#####

function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
        errecho " The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho " 255 is a catch-all error."
    fi

    return 0
}
```

```
}
```

- For API details, see [UpdateItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
///! Update an Amazon DynamoDB table item.
/*!
 \sa updateItem()
 \param tableName: The table name.
 \param partitionKey: The partition key.
 \param partitionValue: The value for the partition key.
 \param attributeKey: The key for the attribute to be updated.
 \param attributeValue: The value for the attribute to be updated.
 \param clientConfiguration: AWS client configuration.
 \return bool: Function succeeded.
 */

/*
 * The example code only sets/updates an attribute value. It processes
 * the attribute value as a string, even if the value could be interpreted
 * as a number. Also, the example code does not remove an existing attribute
 * from the key value.
 */

bool AwsDoc::DynamoDB::updateItem(const Aws::String &tableName,
                                    const Aws::String &partitionKey,
                                    const Aws::String &partitionValue,
                                    const Aws::String &attributeKey,
                                    const Aws::String &attributeValue,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
```

```
// *** Define UpdateItem request arguments.  
// Define TableName argument.  
Aws::DynamoDB::Model::UpdateItemRequest request;  
request.SetTableName(tableName);  
  
// Define KeyName argument.  
Aws::DynamoDB::Model::AttributeValue attribValue;  
attribValue.SetS(partitionValue);  
request.AddKey(partitionKey, attribValue);  
  
// Construct the SET update expression argument.  
Aws::String update_expression("SET #a = :valueA");  
request.SetUpdateExpression(update_expression);  
  
// Construct attribute name argument.  
Aws::Map< Aws::String, Aws::String> expressionAttributeNames;  
expressionAttributeNames["#a"] = attributeKey;  
request.SetExpressionAttributeNames(expressionAttributeNames);  
  
// Construct attribute value argument.  
Aws::DynamoDB::Model::AttributeValue attributeUpdatedValue;  
attributeUpdatedValue.Sets(attribValue);  
Aws::Map< Aws::String, Aws::DynamoDB::Model::AttributeValue>  
expressionAttributeValues;  
expressionAttributeValues[":valueA"] = attributeUpdatedValue;  
request.SetExpressionAttributeValues(expressionAttributeValues);  
  
// Update the item.  
const Aws::DynamoDB::Model::UpdateItemOutcome &outcome =  
dynamoClient.UpdateItem(  
    request);  
if (outcome.IsSuccess()) {  
    std::cout << "Item was updated" << std::endl;  
}  
else {  
    std::cerr << outcome.GetError().GetMessage() << std::endl;  
}  
  
return outcome.IsSuccess();  
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To update an item in a table

The following update-item example updates an item in the MusicCollection table. It adds a new attribute (Year) and modifies the AlbumTitle attribute. All of the attributes in the item, as they appear after the update, are returned in the response.

```
aws dynamodb update-item \  
  --table-name MusicCollection \  
  --key file://key.json \  
  --update-expression "SET #Y = :y, #AT = :t" \  
  --expression-attribute-names file://expression-attribute-names.json \  
  --expression-attribute-values file://expression-attribute-values.json \  
  --return-values ALL_NEW \  
  --return-consumed-capacity TOTAL \  
  --return-item-collection-metrics SIZE
```

Contents of key.json:

```
{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"}  
}
```

Contents of expression-attribute-names.json:

```
{  
    "#Y":"Year", "#AT":"AlbumTitle"  
}
```

Contents of expression-attribute-values.json:

```
{  
    ":y":{"N": "2015"},  
    ":t":{"S": "Louder Than Ever"}  
}
```

Output:

```
{  
    "Attributes": {  
        "AlbumTitle": {  
            "S": "Louder Than Ever"  
        },  
        "Awards": {  
            "N": "10"  
        },  
        "Artist": {  
            "S": "Acme Band"  
        },  
        "Year": {  
            "N": "2015"  
        },  
        "SongTitle": {  
            "S": "Happy Day"  
        }  
    },  
    "ConsumedCapacity": {  
        "TableName": "MusicCollection",  
        "CapacityUnits": 3.0  
    },  
    "ItemCollectionMetrics": {  
        "ItemCollectionKey": {  
            "Artist": {  
                "S": "Acme Band"  
            }  
        },  
        "SizeEstimateRangeGB": [  
            0.0,  
            1.0  
        ]  
    }  
}
```

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

Example 2: To update an item conditionally

The following example updates an item in the MusicCollection table, but only if the existing item does not already have a Year attribute.

```
aws dynamodb update-item \
```

```
--table-name MusicCollection \
--key file://key.json \
--update-expression "SET #Y = :y, #AT = :t" \
--expression-attribute-names file://expression-attribute-names.json \
--expression-attribute-values file://expression-attribute-values.json \
--condition-expression "attribute_not_exists(#Y)"
```

Contents of key.json:

```
{  
    "Artist": {"S": "Acme Band"},  
    "SongTitle": {"S": "Happy Day"}  
}
```

Contents of expression-attribute-names.json:

```
{  
    "#Y": "Year",  
    "#AT": "AlbumTitle"  
}
```

Contents of expression-attribute-values.json:

```
{  
    ":y": {"N": "2015"},  
    ":t": {"S": "Louder Than Ever"}  
}
```

If the item already has a Year attribute, DynamoDB returns the following output.

An error occurred (ConditionalCheckFailedException) when calling the UpdateItem operation: The conditional request failed

For more information, see [Writing an Item](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [UpdateItem](#) in *AWS CLI Command Reference*.

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
// update
// expression.
func (basics TableBasics) UpdateMovie(movie Movie)
    (map[string]map[string]interface{}, error) {
    var err error
    var response *dynamodb.UpdateItemOutput
    var attributeMap map[string]map[string]interface{}
    update := expression.Set(expression.Name("info.rating"),
        expression.Value(movie.Info["rating"]))
    update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
    expr, err := expression.NewBuilder().WithUpdate(update).Build()
    if err != nil {
        log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
    } else {
        response, err = basics.DynamoDbClient.UpdateItem(context.TODO(),
            &dynamodb.UpdateItemInput{
                TableName:           aws.String(basics.TableName),
                Key:                movie.GetKey(),
```

```
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    UpdateExpression:         expr.Update(),
    ReturnValue:              types.ReturnValueUpdatedNew,
)
if err != nil {
    log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
} else {
    err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
    if err != nil {
        log.Printf("Couldn't unmarshal update response. Here's why: %v\n", err)
    }
}
}

return attributeMap, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}
```

```
// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [UpdateItem in AWS SDK for Go API Reference](#).

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Updates an item in a table using [DynamoDbClient](#).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To update an Amazon DynamoDB table using the AWS SDK for Java V2, its better
```

```
* practice to use the
* Enhanced Client, See the EnhancedModifyItem example.
*/
public class UpdateItem {
    public static void main(String[] args) {
        final String usage = """
            Usage:
            <tableName> <key> <keyVal> <name> <updateVal>
            Where:
            tableName - The Amazon DynamoDB table (for example, Music3).
            key - The name of the key in the table (for example, Artist).
            keyVal - The value of the key (for example, Famous Band).
            name - The name of the column where the value is updated (for
example, Awards).
            updateVal - The value used to update an item (for example,
14).

        Example:
            UpdateItem Music3 Artist Famous Band Awards 14
            """;
        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String name = args[3];
        String updateVal = args[4];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
        updateTableItem(ddb, tableName, key, keyVal, name, updateVal);
        ddb.close();
    }

    public static void updateTableItem(DynamoDbClient ddb,
        String tableName,
        String key,
```

```
        String keyVal,
        String name,
        String updateVal) {

    HashMap<String,AttributeValue> itemKey = new HashMap<>();
    itemKey.put(key,AttributeValue.builder()
        .s(keyVal)
        .build());

    HashMap<String,AttributeValueUpdate> updatedValues = new HashMap<>();
    updatedValues.put(name,AttributeValueUpdate.builder()
        .value(AttributeValue.builder().s(updateVal).build())
        .action(AttributeAction.PUT)
        .build());

    UpdateItemRequest request = UpdateItemRequest.builder()
        .tableName(tableName)
        .key(itemKey)
        .attributeUpdates(updatedValues)
        .build();

    try {
        ddb.updateItem(request);
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    System.out.println("The Amazon DynamoDB table was updated!");
}
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [UpdateCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [UpdateItem](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun updateTableItem(
```

```
        tableNameVal: String,  
        keyName: String,  
        keyVal: String,  
        name: String,  
        updateVal: String  
    ) {  
        val itemKey = mutableMapOf<String, AttributeValue>()  
        itemKey[keyName] = AttributeValue.S(keyVal)  
  
        val updatedValues = mutableMapOf<String, AttributeValueUpdate>()  
        updatedValues[name] = AttributeValueUpdate {  
            value = AttributeValue.S(updateVal)  
            action = AttributeAction.Put  
        }  
  
        val request = UpdateItemRequest {  
            tableName = tableNameVal  
            key = itemKey  
            attributeUpdates = updatedValues  
        }  
  
        DynamoDbClient { region = "us-east-1" }.use { ddb ->  
            ddb.updateItem(request)  
            println("Item in $tableNameVal was updated")  
        }  
    }  
}
```

- For API details, see [UpdateItem in AWS SDK for Kotlin API reference](#).

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
echo "What rating would you like to give {$movie['Item']['title']['S']}?  
\n";
```

```
$rating = 0;
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
    $rating = testable_readline("Rating (1-10): ");
}
$service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
$rating);

public function updateItemAttributeByKey(
    string $tableName,
    array $key,
    string $attributeName,
    string $attributeType,
    string $newValue
) {
    $this->dynamoDbClient->updateItem([
        'Key' => $key['Item'],
        'TableName' => $tableName,
        'UpdateExpression' => "set #NV=:NV",
        'ExpressionAttributeNames' => [
            '#NV' => $attributeName,
        ],
        'ExpressionAttributeValues' => [
            ':NV' => [
                $attributeType => $newValue
            ]
        ],
    ]);
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Update an item by using an update expression.

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
  
    def update_movie(self, title, year, rating, plot):  
        """  
        Updates rating and plot data for a movie in the table.  
  
        :param title: The title of the movie to update.  
        :param year: The release year of the movie to update.  
        :param rating: The updated rating to give the movie.  
        :param plot: The updated plot summary to give the movie.  
        :return: The fields that were updated, with their new values.  
        """  
        try:  
            response = self.table.update_item(  
                Key={"year": year, "title": title},  
                UpdateExpression="set info.rating=:r, info.plot=:p",  
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":  
plot},  
                ReturnValues="UPDATED_NEW",  
            )  
        except ClientError as err:  
            logger.error(  
                "Couldn't update movie %s in table %s. Here's why: %s: %s",  
                title,  
                self.table.name,  
                err.response["Error"]["Code"],  
                err.response["Error"]["Message"],  
            )  
            raise  
        else:  
            return response["Attributes"]
```

Update an item by using an update expression that includes an arithmetic operation.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table

    def update_rating(self, title, year, rating_change):
        """
        Updates the quality rating of a movie in the table by using an arithmetic
        operation in the update expression. By specifying an arithmetic
        operation,
        you can adjust a value in a single request, rather than first getting its
        value and then setting its new value.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating_change: The amount to add to the current rating for the
        movie.
        :return: The updated rating.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating = info.rating + :val",
                ExpressionAttributeValues={":val": Decimal(str(rating_change))},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

Update an item only when it meets certain conditions.

```
class UpdateQueryWrapper:  
    def __init__(self, table):  
        self.table = table  
  
    def remove_actors(self, title, year, actor_threshold):  
        """  
        Removes an actor from a movie, but only when the number of actors is  
        greater  
        than a specified threshold. If the movie does not list more than the  
        threshold,  
        no actors are removed.  
  
        :param title: The title of the movie to update.  
        :param year: The release year of the movie to update.  
        :param actor_threshold: The threshold of actors to check.  
        :return: The movie data after the update.  
        """  
        try:  
            response = self.table.update_item(  
                Key={"year": year, "title": title},  
                UpdateExpression="remove info.actors[0]",  
                ConditionExpression="size(info.actors) > :num",  
                ExpressionAttributeValues={":num": actor_threshold},  
                ReturnValues="ALL_NEW",  
            )  
        except ClientError as err:  
            if err.response["Error"]["Code"] ==  
                "ConditionalCheckFailedException":  
                logger.warning(  
                    "Didn't update %s because it has fewer than %s actors.",  
                    title,  
                    actor_threshold + 1,  
                )  
            else:  
                logger.error(  
                    "Couldn't update movie %s. Here's why: %s: %s",  
                    title,  
                    err.response["Error"]["Code"],  
                )
```

```
        err.response["Error"]["Message"],
    )
raise
else:
    return response["Attributes"]
```

- For API details, see [UpdateItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
  attr_reader :dynamo_resource
  attr_reader :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: "us-east-1")
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Updates rating and plot data for a movie in the table.
  #
  # @param movie [Hash] The title, year, plot, rating of the movie.
  def update_item(movie)

    response = @table.update_item(
      key: {"year" => movie[:year], "title" => movie[:title]},
      update_expression: "set info.rating=:r",
      expression_attribute_values: { ":r" => movie[:rating] },
      return_values: "UPDATED_NEW")
    rescue Aws::DynamoDB::Errors::ServiceError => e
```

```
    puts("Couldn't update movie #{movie[:title]} (#{$movie[:year]}) in table  
#{@table.name}\n")  
    puts("\t#{$e.code}: #{$e.message}")  
    raise  
else  
    response.attributes  
end
```

- For API details, see [UpdateItem](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
TRY.  
    oo_output = lo_dyn->updateitem(  
        iv_tablename      = iv_table_name  
        it_key            = it_item_key  
        it_attributeupdates = it_attribute_updates ).  
    MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.  
    CATCH /aws1/cx_dyncondalcheckfaile00.  
        MESSAGE 'A condition specified in the operation could not be evaluated.'  
        TYPE 'E'.  
        CATCH /aws1/cx_dyncsourcenotfoundex.  
            MESSAGE 'The table or index does not exist' TYPE 'E'.  
        CATCH /aws1/cx_dyntransactconflictex.  
            MESSAGE 'Another transaction is using the item' TYPE 'E'.  
    ENDTRY.
```

- For API details, see [UpdateItem](#) in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

Note

This is prerelease documentation for an SDK in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Update the specified movie with new `rating` and `plot` information.  
///  
/// - Parameters:  
///   - title: The title of the movie to update.  
///   - year: The release year of the movie to update.  
///   - rating: The new rating for the movie.  
///   - plot: The new plot summary string for the movie.  
///  
/// - Returns: An array of mappings of attribute names to their new  
///   listing each item actually changed. Items that didn't need to change  
///   aren't included in this list. `nil` if no changes were made.  
///  
func update(title: String, year: Int, rating: Double? = nil, plot: String? =  
nil) async throws  
    -> [Swift.String:DynamoDBClientTypes.AttributeValue]? {  
guard let client = self.ddbClient else {  
    throw MoviesError.UninitializedClient  
}  
  
// Build the update expression and the list of expression attribute  
// values. Include only the information that's changed.  
  
var expressionParts: [String] = []  
var attrValues: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
```

```
if rating != nil {
    expressionParts.append("info.rating=:r")
    attrValues[":r"] = .n(String(rating!))
}
if plot != nil {
    expressionParts.append("info.plot=:p")
    attrValues[":p"] = .s(plot!)
}
let expression: String = "set \\" + (expressionParts.joined(separator: ", ")) + "\""

let input = UpdateItemInput(
    // Create substitution tokens for the attribute values, to ensure
    // no conflicts in expression syntax.
    expressionAttributeValues: attrValues,
    // The key identifying the movie to update consists of the release
    // year and title.
    key: [
        "year": .n(String(year)),
        "title": .s(title)
    ],
    returnValues: .updatedNew,
    tableName: self.tableName,
    updateExpression: expression
)
let output = try await client.updateItem(input: input)

guard let attributes: [Swift.String:DynamoDBClientTypes.AttributeValue] =
output.attributes else {
    throw MoviesError.InvalidAttributes
}
return attributes
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Write a batch of DynamoDB items using an AWS SDK

The following code examples show how to write a batch of DynamoDB items.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Get started with tables, items, and queries](#)

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Writes a batch of items to the movie table.

```
/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The full path to the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public static List<Movie> ImportMovies(string movieFileName)
{
    if (!File.Exists(movieFileName))
    {
        return null;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonSerializer.Deserialize<List<Movie>>(
        json,
        new JsonSerializerOptions
    {
```

```
        PropertyNameCaseInsensitive = true
    });

    // Now return the first 250 entries.
    return allMovies.GetRange(0, 250);
}

/// <summary>
/// Writes 250 items to the movie table.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="movieFileName">A string containing the full path to
/// the JSON file containing movie data.</param>
/// <returns>A long integer value representing the number of movies
/// imported from the JSON file.</returns>
public static async Task<long> BatchWriteItemsAsync(
    AmazonDynamoDBClient client,
    string movieFileName)
{
    var movies = ImportMovies(movieFileName);
    if (movies is null)
    {
        Console.WriteLine("Couldn't find the JSON file with movie
data.");
        return 0;
    }

    var context = new DynamoDBContext(client);

    var movieBatch = context.CreateBatchWrite<Movie>();
    movieBatch.AddPutItems(movies);

    Console.WriteLine("Adding imported movies to the table.");
    await movieBatch.ExecuteAsync();

    return movies.Count;
}
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#####
# function dynamodb_batch_write_item
#
# This function writes a batch of items into a DynamoDB table.
#
# Parameters:
#     -i item -- Path to json file containing the items to write.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_batch_write_item() {
    local item response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_batch_write_item"
    echo "Write a batch of items into a DynamoDB table."
    echo " -i item -- Path to json file containing the items to write."
    echo ""
}

while getopt "i:h" option; do
    case "${option}" in
        i) item="${OPTARG}" ;;
        h)
            usage
            return 0
    ;;
    esac
done
```

```
\?)  
    echo "Invalid parameter"  
    usage  
    return 1  
;;  
esac  
done  
export OPTIND=1  
  
if [[ -z "$item" ]]; then  
    errecho "ERROR: You must provide an item with the -i parameter."  
    usage  
    return 1  
fi  
  
iecho "Parameters:\n"  
iecho "    table_name: $table_name"  
iecho "    item: $item"  
iecho ""  
  
response=$(aws dynamodb batch-write-item \  
    --request-items file://"$item")  
  
local error_code=${?}  
  
if [[ $error_code -ne 0 ]]; then  
    aws_cli_error_log $error_code  
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"  
    return 1  
fi  
  
return 0  
}
```

The utility functions used in this example.

```
#####  
# function iecho  
#  
# This function enables the script to display the specified text only if  
# the global variable $VERBOSE is set to true.  
#####
```

```
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-help-return-codes.
#
# The function expects the following argument:
#       $1 - The error code returned by the AWS CLI.
#
# Returns:
#       0: - Success.
#
#####

function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
    if [ "$err_code" == 1 ]; then
        errecho " One or more S3 transfers failed."
    elif [ "$err_code" == 2 ]; then
        errecho " Command line failed to parse."
    elif [ "$err_code" == 130 ]; then
        errecho " Process received SIGINT."
    elif [ "$err_code" == 252 ]; then
        errecho " Command syntax invalid."
    elif [ "$err_code" == 253 ]; then
        errecho " The system environment or configuration was invalid."
    elif [ "$err_code" == 254 ]; then
```

```
    errecho "  The service returned an error."
    elif [ "$err_code" == 255 ]; then
        errecho "  255 is a catch-all error."
    fi

    return 0
}
```

- For API details, see [BatchWriteItem](#) in *AWS CLI Command Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
#!/ Batch write items from a JSON file.
/*
\sa batchWriteItem()
\param jsonFilePath: JSON file path.
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/

/*
 * The input for this routine is a JSON file that you can download from the
following URL:
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
SampleData.html.
 *
 * The JSON data uses the BatchWriteItem API request syntax. The JSON strings are
 * converted toAttributeValue objects. These AttributeValue objects will then
generate
 * JSON strings when constructing the BatchWriteItem request, essentially
outputting
 * their input.
 *
```

```
* This is perhaps an artificial example, but it demonstrates the APIs.  
*/  
  
bool AwsDoc::DynamoDB::batchWriteItem(const Aws::String &jsonFilePath,  
                                      const Aws::Client::ClientConfiguration  
&clientConfiguration) {  
    std::ifstream fileStream(jsonFilePath);  
  
    if (!fileStream) {  
        std::cerr << "Error: could not open file '" << jsonFilePath << "'."  
              << std::endl;  
    }  
  
    std::stringstream stringstream;  
    stringstream << fileStream.rdbuf();  
    Aws::Utils::Json::JsonValue jsonValue(stringStream);  
  
    Aws::DynamoDB::Model::BatchWriteItemRequest batchWriteItemRequest;  
    Aws::Map<Aws::String, Aws::Utils::Json::JsonValue> level1Map =  
    jsonValue.View().GetAllObjects();  
    for (const auto &level1Entry: level1Map) {  
        const Aws::Utils::Json::JsonValue &entriesView = level1Entry.second;  
        const Aws::String &tableName = level1Entry.first;  
        // The JSON entries at this level are as follows:  
        // key - table name  
        // value - list of request objects  
        if (!entriesView.IsListType()) {  
            std::cerr << "Error: JSON file entry '"  
                  << tableName << "' is not a list." << std::endl;  
            continue;  
        }  
  
        Aws::Utils::Array<Aws::Utils::Json::JsonValue> entries =  
        entriesView.AsArray();  
  
        Aws::Vector<Aws::DynamoDB::Model::WriteRequest> writeRequests;  
        if (AwsDoc::DynamoDB::addWriteRequests(tableName, entries,  
                                              writeRequests)) {  
            batchWriteItemRequest.AddRequestItems(tableName, writeRequests);  
        }  
    }  
  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
```

```
Aws::DynamoDB::Model::BatchWriteItemOutcome outcome =
dynamoClient.BatchWriteItem(
    batchWriteItemRequest);

if (outcome.IsSuccess()) {
    std::cout << "DynamoDB::BatchWriteItem was successful." << std::endl;
}
else {
    std::cerr << "Error with DynamoDB::BatchWriteItem. "
        << outcome.GetError().GetMessage()
        << std::endl;
}

return true;
}

//! Convert requests in JSON format to a vector of WriteRequest objects.
/*!
\sa addWriteRequests()
\param tableName: Name of the table for the write operations.
\param requestsJson: Request data in JSON format.
\param writeRequests: Vector to receive the WriteRequest objects.
\return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::addWriteRequests(const Aws::String &tableName,
                                         const
                                         Aws::Utils::Array<Aws::Utils::Json::JsonValue> &requestsJson,
                                         Aws::Vector<Aws::DynamoDB::Model::WriteRequest> &writeRequests) {
    for (size_t i = 0; i < requestsJson.GetLength(); ++i) {
        const Aws::Utils::Json::JsonValue &requestsEntry = requestsJson[i];
        if (!requestsEntry.IsObject()) {
            std::cerr << "Error: incorrect requestsEntry type "
                << requestsEntry.WriteReadable() << std::endl;
            return false;
        }

        Aws::Map<Aws::String, Aws::Utils::Json::JsonValue> requestsMap =
        requestsEntry.GetAllObjects();

        for (const auto &request: requestsMap) {
            const Aws::String &requestType = request.first;
            const Aws::Utils::Json::JsonValue &requestJsonValue = request.second;
```

```
        if (requestType == "PutRequest") {
            if (!requestJsonView.ValueExists("Item")) {
                std::cerr << "Error: item key missing for requests "
                << requestJsonView.WriteReadable() << std::endl;
                return false;
            }
            Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
            attributes;
            if (!getAttributeObjectsMap(requestJsonView.GetObject("Item"),
                attributes)) {
                std::cerr << "Error getting attributes "
                << requestJsonView.WriteReadable() << std::endl;
                return false;
            }
        }

        Aws::DynamoDB::Model::PutRequest putRequest;
        putRequest.SetItem(attributes);
        writeRequests.push_back(
            Aws::DynamoDB::Model::WriteRequest().WithPutRequest(
                putRequest));
    }
    else {
        std::cerr << "Error: unimplemented request type '" << requestType
        << '.' << std::endl;
    }
}

return true;
}

//! Generate a map of AttributeValue objects from JSON records.
/*! \sa getAttributeObjectsMap()
 * \param jsonView: JSONView of attribute records.
 * \param writeRequests: Map to receive the AttributeValue objects.
 * \return bool: Function succeeded.
 */
bool
AwsDoc::DynamoDB::getAttributeObjectsMap(const Aws::Utils::Json::JsonValue
    &jsonView,
                                            Aws::Map<Aws::String,
    Aws::DynamoDB::Model::AttributeValue> &attributes) {
```

```
Aws::Map< Aws::String, Aws::Utils::Json::JsonView> objectsMap =  
jsonView.GetAllObjects();  
for (const auto &entry: objectsMap) {  
    const Aws::String &attributeKey = entry.first;  
    const Aws::Utils::Json::JsonView &attributeJsonValue = entry.second;  
  
    if (!attributeJsonValue.IsObject()) {  
        std::cerr << "Error: attribute not an object "  
              << attributeJsonValue.WriteReadable() << std::endl;  
        return false;  
    }  
  
    attributes.emplace(attributeKey,  
  
Aws::DynamoDB::Model::AttributeValue(attributeJsonValue));  
}  
  
return true;  
}
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To add multiple items to a table

The following batch-write-item example adds three new items to the MusicCollection table using a batch of three PutItem requests. It also requests information about the number of write capacity units consumed by the operation and any item collections modified by the operation.

```
aws dynamodb batch-write-item \  
--request-items file://request-items.json \  
--return-consumed-capacity INDEXES \  
--return-item-collection-metrics SIZE
```

Contents of request-items.json:

```
{
```

```
"MusicCollection": [
    {
        "PutRequest": {
            "Item": {
                "Artist": {"S": "No One You Know"},
                "SongTitle": {"S": "Call Me Today"},
                "AlbumTitle": {"S": "Somewhat Famous"}
            }
        }
    },
    {
        "PutRequest": {
            "Item": {
                "Artist": {"S": "Acme Band"},
                "SongTitle": {"S": "Happy Day"},
                "AlbumTitle": {"S": "Songs About Life"}
            }
        }
    },
    {
        "PutRequest": {
            "Item": {
                "Artist": {"S": "No One You Know"},
                "SongTitle": {"S": "Scared of My Shadow"},
                "AlbumTitle": {"S": "Blue Sky Blues"}
            }
        }
    }
]
```

Output:

```
{
    "UnprocessedItems": {},
    "ItemCollectionMetrics": {
        "MusicCollection": [
            {
                "ItemCollectionKey": {
                    "Artist": {
                        "S": "No One You Know"
                    }
                },
                ...
            }
        ]
    }
}
```

```
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ],
    },
    {
        "ItemCollectionKey": {
            "Artist": {
                "S": "Acme Band"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
],
"ConsumedCapacity": [
{
    "TableName": "MusicCollection",
    "CapacityUnits": 6.0,
    "Table": {
        "CapacityUnits": 3.0
    },
    "LocalSecondaryIndexes": {
        "AlbumTitleIndex": {
            "CapacityUnits": 3.0
        }
    }
}
]
```

For more information, see [Batch Operations](#) in the *Amazon DynamoDB Developer Guide*.

- For API details, see [BatchWriteItem](#) in *AWS CLI Command Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// AddMovieBatch adds a slice of movies to the DynamoDB table. The function sends
// batches of 25 movies to DynamoDB until all movies are added or it reaches the
// specified maximum.
func (basics TableBasics) AddMovieBatch(movies []Movie, maxMovies int) (int,
    error) {
    var err error
    var item map[string]types.AttributeValue
    written := 0
    batchSize := 25 // DynamoDB allows a maximum batch size of 25 items.
    start := 0
    end := start + batchSize
    for start < maxMovies && start < len(movies) {
        var writeReqs []types.WriteRequest
        if end > len(movies) {
            end = len(movies)
        }
        for _, movie := range movies[start:end] {
            item, err = attributevalue.MarshalMap(movie)
            if err != nil {
```

```
    log.Printf("Couldn't marshal movie %v for batch writing. Here's why: %v\n",
    movie.Title, err)
} else {
    writeReqs = append(
        writeReqs,
        types.WriteRequest{PutRequest: &types.PutRequest{Item: item}},
    )
}
}
_, err = basics.DynamoDbClient.BatchWriteItem(context.TODO(),
&dynamodb.BatchWriteItemInput{
    RequestItems: map[string][]types.WriteRequest{basics.TableName: writeReqs}},
if err != nil {
    log.Printf("Couldn't add a batch of movies to %v. Here's why: %v\n",
    basics.TableName, err)
} else {
    written += len(writeReqs)
}
start = end
end += batchSize
}

return written, err
}

// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
}
```

```
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Inserts many items into a table by using the service client.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchWriteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.BatchWriteItemResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutRequest;
import software.amazon.awssdk.services.dynamodb.model.WriteRequest;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
```

```
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development environment,
 * including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class BatchWriteItems {
    public static void main(String[] args){
        final String usage = """
            Usage:
            <tableName>
            Where:
            tableName - The Amazon DynamoDB table (for example, Music).\n"""
        ;
        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        addBatchItems(dynamoDbClient, tableName);
    }

    public static void addBatchItems(DynamoDbClient dynamoDbClient, String tableName) {
        // Specify the updates you want to perform.
        List<WriteRequest> writeRequests = new ArrayList<>();

        // Set item 1.
        Map<String, AttributeValue> item1Attributes = new HashMap<>();
        item1Attributes.put("Artist",
        AttributeValue.builder().s("Artist1").build());
        item1Attributes.put("Rating", AttributeValue.builder().s("5").build());
        item1Attributes.put("Comments", AttributeValue.builder().s("Great
song!").build());
    }
}
```

```
        item1Attributes.put("SongTitle",
AttributeValue.builder().s("SongTitle1").build()));

writeRequests.add(WriteRequest.builder().putRequest(PutRequest.builder().item(item1Attrib

        // Set item 2.
Map<String, AttributeValue> item2Attributes = new HashMap<>();
item2Attributes.put("Artist",
AttributeValue.builder().s("Artist2").build());
item2Attributes.put("Rating", AttributeValue.builder().s("4").build());
item2Attributes.put("Comments", AttributeValue.builder().s("Nice
melody.").build());
item2Attributes.put("SongTitle",
AttributeValue.builder().s("SongTitle2").build());

writeRequests.add(WriteRequest.builder().putRequest(PutRequest.builder().item(item2Attrib

try {
    // Create the BatchWriteItemRequest.
    BatchWriteItemRequest batchWriteItemRequest =
BatchWriteItemRequest.builder()
    .requestItems(Map.of(tableName, writeRequests))
    .build();

    // Execute the BatchWriteItem operation.
    BatchWriteItemResponse batchWriteItemResponse =
dynamoDbClient.batchWriteItem(batchWriteItemRequest);

    // Process the response.
    System.out.println("Batch write successful: " +
batchWriteItemResponse);

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

Inserts many items into a table by using the enhanced client.

```
import com.example.dynamodb.Customer;
```

```
import com.example.dynamodb.Music;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import
software.amazon.awssdk.enhanced.dynamodb.model.BatchWriteItemEnhancedRequest;
import software.amazon.awssdk.enhanced.dynamodb.model.WriteBatch;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneOffset;

/*
 * Before running this code example, create an Amazon DynamoDB table named
Customer with these columns:
 * - id - the id of the record that is the key
 * - custName - the customer name
 * - email - the email value
 * - registrationDate - an instant value when the item was added to the table
 *
 * Also, ensure that you have set up your development environment, including your
credentials.
*
* For information, see this documentation topic:
*
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
*/
public class EnhancedBatchWriteItems {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
        DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
            .build();
        putBatchRecords(enhancedClient);
        ddb.close();
    }
}
```

```
}

    public static void putBatchRecords(DynamoDbEnhancedClient enhancedClient)
{
    try {
        DynamoDbTable<Customer> customerMappedTable =
enhancedClient.table("Customer",
                        TableSchema.fromBean(Customer.class));
        DynamoDbTable<Music> musicMappedTable =
enhancedClient.table("Music",
                        TableSchema.fromBean(Music.class));
        LocalDate localDate = LocalDate.parse("2020-04-07");
        LocalDateTime localDateTime = localDate.atStartOfDay();
        Instant instant =
localDateTime.toInstant(ZoneOffset.UTC);

        Customer record2 = new Customer();
        record2.setCustName("Fred Pink");
        record2.setId("id110");
        record2.setEmail("fredp@noserver.com");
        record2.setRegistrationDate(instant);

        Customer record3 = new Customer();
        record3.setCustName("Susan Pink");
        record3.setId("id120");
        record3.setEmail("spink@noserver.com");
        record3.setRegistrationDate(instant);

        Customer record4 = new Customer();
        record4.setCustName("Jerry orange");
        record4.setId("id101");
        record4.setEmail("jorange@noserver.com");
        record4.setRegistrationDate(instant);

        BatchWriteItemEnhancedRequest
batchWriteItemEnhancedRequest = BatchWriteItemEnhancedRequest
                                .builder()
                                .writeBatches(
WriteBatch.builder(Customer.class) // add items to the Customer

                                // table

                                .mappedTableResource(customerMappedTable)
```

```
.addPutItem(builder -> builder.item(record2))

.addPutItem(builder -> builder.item(record3))

.addPutItem(builder -> builder.item(record4))

.WriteBatch.builder(Music.class) // delete an item from the Music

// table

.mappedTableResource(musicMappedTable)

.addDeleteItem(builder -> builder.key(
    Key.builder().partitionValue(
        "Famous Band"))

.build()))
.build());
.build();

// Add three items to the Customer table and delete one
item from the Music
// table.

enhancedClient.batchWriteItem(batchWriteItemEnhancedRequest);
System.out.println("done");

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [BatchWrite](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";
import { readFileSync } from "fs";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";

const dirname = dirnameFromMetaUrl(import.meta.url);

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const file = readFileSync(
    `${dirname}../../../../resources/sample_files/movies.json`,
  );

  const movies = JSON.parse(file.toString());

  // chunkArray is a local convenience function. It takes an array and returns
  // a generator function. The generator function yields every N items.
  const movieChunks = chunkArray(movies, 25);
```

```
// For every chunk of 25 movies, make one BatchWrite request.  
for (const chunk of movieChunks) {  
    const putRequests = chunk.map((movie) => ({  
        PutRequest: {  
            Item: movie,  
        },  
    }));  
  
    const command = new BatchWriteCommand({  
        RequestItems: {  
            // An existing table is required. A composite key of 'title' and 'year'  
            // is recommended  
            // to account for duplicate titles.  
            ["BatchWriteMoviesTable"]: putRequests,  
        },  
    });  
  
    await docClient.send(command);  
}  
};
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create DynamoDB service object  
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });  
  
var params = {  
    RequestItems: {  
        TABLE_NAME: [  
    },  
};
```

```
{  
  PutRequest: {  
    Item: {  
      KEY: { N: "KEY_VALUE" },  
      ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },  
      ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },  
    },  
  },  
},  
{  
  PutRequest: {  
    Item: {  
      KEY: { N: "KEY_VALUE" },  
      ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },  
      ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },  
    },  
  },  
},  
],  
},  
};  
  
ddb.batchWriteItem(params, function (err, data) {  
  if (err) {  
    console.log("Error", err);  
  } else {  
    console.log("Success", data);  
  }  
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [BatchWriteItem](#) in [AWS SDK for JavaScript API Reference](#).

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function writeBatch(string $TableName, array $Batch, int $depth = 2)
{
    if (--$depth <= 0) {
        throw new Exception("Max depth exceeded. Please try with fewer batch
items or increase depth.");
    }

    $marshal = new Marshaler();
    $total = 0;
    foreach (array_chunk($Batch, 25) as $Items) {
        foreach ($Items as $Item) {
            $BatchWrite['RequestItems'][$TableName][] = ['PutRequest' =>
['Item' => $marshal->marshalItem($Item)]];
        }
        try {
            echo "Batching another " . count($Items) . " for a total of " .
($total += count($Items)) . " items!\n";
            $response = $this->dynamoDbClient->batchWriteItem($BatchWrite);
            $BatchWrite = [];
        } catch (Exception $e) {
            echo "uh oh...";
            echo $e->getMessage();
            die();
        }
        if ($total >= 250) {
            echo "250 movies is probably enough. Right? We can stop there.
\n";
            break;
        }
    }
}
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class Movies:  
    """Encapsulates an Amazon DynamoDB table of movie data."""  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource  
        # The table variable is set during the scenario in the call to  
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.  
        self.table = None  
  
    def write_batch(self, movies):  
        """  
        Fills an Amazon DynamoDB table with the specified data, using the Boto3  
        Table.batch_writer() function to put the items in the table.  
        Inside the context manager, Table.batch_writer builds a list of  
        requests. On exiting the context manager, Table.batch_writer starts  
        sending  
        batches of write requests to Amazon DynamoDB and automatically  
        handles chunking, buffering, and retrying.  
  
        :param movies: The data to put in the table. Each item must contain at  
        least  
            the keys required by the schema that was specified when  
            the  
            table was created.  
        """  
        try:            ...
```

```
        with self.table.batch_writer() as writer:
            for movie in movies:
                writer.put_item(Item=movie)
        except ClientError as err:
            logger.error(
                "Couldn't load data into table %s. Here's why: %s: %s",
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class DynamoDBBasics
    attr_reader :dynamo_resource
    attr_reader :table

    def initialize(table_name)
        client = Aws::DynamoDB::Client.new(region: "us-east-1")
        @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
        @table = @dynamo_resource.table(table_name)
    end

    # Fills an Amazon DynamoDB table with the specified data. Items are sent in
    # batches of 25 until all items are written.
    #
    # @param movies [Enumerable] The data to put in the table. Each item must
    # contain at least
```

```
#           the keys required by the schema that was specified
when the
#           table was created.
def write_batch(movies)
  index = 0
  slice_size = 25
  while index < movies.length
    movie_items = []
    movies[index, slice_size].each do |movie|
      movie_items.append({put_request: { item: movie }})
    end
    @dynamo_resource.client.batch_write_item({request_items: { @table.name =>
  movie_items }})
    index += slice_size
  end
rescue Aws::DynamoDB::Errors::ServiceError => e
  puts(
    "Couldn't load data into table #{@table.name}. Here's why:")
  puts("\t#{e.code}: #{e.message}")
  raise
end
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Ruby API Reference*.

Swift

SDK for Swift

 **Note**

This is prerelease documentation for an SDK in preview release. It is subject to change.

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// Populate the movie database from the specified JSON file.  
///  
/// - Parameter jsonPath: Path to a JSON file containing movie data.  
///  
func populate(jsonPath: String) async throws {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    // Create a Swift `URL` and use it to load the file into a `Data`  
    // object. Then decode the JSON into an array of `Movie` objects.  
  
    let fileUrl = URL(fileURLWithPath: jsonPath)  
    let jsonData = try Data(contentsOf: fileUrl)  
  
    var movieList = try JSONDecoder().decode([Movie].self, from: jsonData)  
  
    // Truncate the list to the first 200 entries or so for this example.  
  
    if movieList.count > 200 {  
        movieList = Array(movieList[...199])  
    }  
  
    // Before sending records to the database, break the movie list into  
    // 25-entry chunks, which is the maximum size of a batch item request.  
  
    let count = movieList.count  
    let chunks = stride(from: 0, to: count, by: 25).map {  
        Array(movieList[$0 ..< Swift.min($0 + 25, count)])  
    }  
  
    // For each chunk, create a list of write request records and populate  
    // them with `PutRequest` requests, each specifying one movie from the  
    // chunk. Once the chunk's items are all in the `PutRequest` list,  
    // send them to Amazon DynamoDB using the  
    // `DynamoDBClient.batchWriteItem()` function.  
  
    for chunk in chunks {  
        var requestList: [DynamoDBClientTypes.WriteRequest] = []  
  
        for movie in chunk {  
            let item = try await movie.getAsItem()  
            let request = DynamoDBClientTypes.WriteRequest(  
                item: item,  
                conditionExpression: nil,  
                expressionAttributeNames: nil,  
                expressionAttributeValues: nil,  
                returnConsumedCapacity: nil,  
                returnItemCollectionMetrics: nil,  
                returnNewImage: nil,  
                returnSize: nil,  
                returnValidated: nil  
            )  
            requestList.append(request)  
        }  
        try await client.batchWriteItem(requestList: requestList)  
    }  
}
```

```
        putRequest: .init(
            item: item
        )
    )
    requestList.append(request)
}

let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
_ = try await client.batchWriteItem(input: input)
}
}
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Scenarios for DynamoDB using AWS SDKs

The following code examples show you how to implement common scenarios in DynamoDB with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within DynamoDB. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- [Accelerate DynamoDB reads with DAX using an AWS SDK](#)
- [Get started with DynamoDB tables, items, and queries using an AWS SDK](#)
- [Query a DynamoDB table by using batches of PartiQL statements and an AWS SDK](#)
- [Query a DynamoDB table using PartiQL and an AWS SDK](#)
- [Use a document model for DynamoDB using an AWS SDK](#)
- [Use a high-level object persistence model for DynamoDB using an AWS SDK](#)

Accelerate DynamoDB reads with DAX using an AWS SDK

The following code example shows how to:

- Create and write data to a table with both the DAX and SDK clients.
- Get, query, and scan the table with both clients and compare their performance.

For more information, see [Developing with the DynamoDB Accelerator Client](#).

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a table with either the DAX or Boto3 client.

```
import boto3

def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table_name = "TryDaxTable"
    params = {
        "TableName": table_name,
        "KeySchema": [
            {"AttributeName": "partition_key", "KeyType": "HASH"},
            {"AttributeName": "sort_key", "KeyType": "RANGE"},
        ],
    }
```

```
        "AttributeDefinitions": [
            {"AttributeName": "partition_key", "AttributeType": "N"},
            {"AttributeName": "sort_key", "AttributeType": "N"},
        ],
        "ProvisionedThroughput": {"ReadCapacityUnits": 10, "WriteCapacityUnits": 10},
    }
}

table = dyn_resource.create_table(**params)
print(f"Creating {table_name}...")
table.wait_until_exists()
return table

if __name__ == "__main__":
    dax_table = create_dax_table()
    print(f"Created table.")
```

Write test data to the table.

```
import boto3


def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate
    the
                    table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    some_data = "X" * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(
                Item={
```

```
        "partition_key": partition_key,
        "sort_key": sort_key,
        "some_data": some_data,
    }
)
print(f"Put item ({partition_key}, {sort_key}) succeeded.")

if __name__ == "__main__":
    write_key_count = 10
    write_item_size = 1000
    print(
        f"Writing {write_key_count*write_key_count} items to the table."
        f"Each item is {write_item_size} characters."
    )
    write_data_to_dax_table(write_key_count, write_item_size)
```

Get items for a number of iterations for both the DAX client and the Boto3 client and report the time spent for each.

```
import argparse
import sys
import time
import amazonadax
import boto3

def get_item_test(key_count, iterations, dyn_resource=None):
    """
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each
                      iteration.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")
```

```
table = dyn_resource.Table("TryDaxTable")
start = time.perf_counter()
for _ in range(iterations):
    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.get_item(
                Key={"partition_key": partition_key, "sort_key": sort_key}
            )
            print(".", end="")
            sys.stdout.flush()
print()
end = time.perf_counter()
return start, end

if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_key_count = 10
    test_iterations = 50
    if args.endpoint_url:
        print(
            f"Getting each item from the table {test_iterations} times, "
            f"using the DAX client."
        )
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
            as dax:
                test_start, test_end = get_item_test(
                    test_key_count, test_iterations, dyn_resource=dax
                )
    else:
        print(
            f"Getting each item from the table {test_iterations} times, "
            f"using the Boto3 client."
```

```
        )
    test_start, test_end = get_item_test(test_key_count, test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/ test_iterations}."
    )
```

Query the table for a number of iterations for both the DAX client and the Boto3 client and report the time spent for each.

```
import argparse
import time
import sys
import amazonadax
import boto3
from boto3.dynamodb.conditions import Key


def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param partition_key: The partition key value to use in the query. The query
                          returns items that have partition keys equal to this
                          value.
    :param sort_keys: The range of sort key values for the query. The query
                     returns
                         items that have sort key values between these two values.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    key_condition_expression = Key("partition_key").eq(partition_key) & Key(
        "sort_key"
    ).between(*sort_keys)
```

```
start = time.perf_counter()
for _ in range(iterations):
    table.query(KeyConditionExpression=key_condition_expression)
    print(".", end="")
    sys.stdout.flush()
print()
end = time.perf_counter()
return start, end

if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_partition_key = 5
    test_sort_keys = (2, 9)
    test_iterations = 100
    if args.endpoint_url:
        print(f"Querying the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondex.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = query_test(
                test_partition_key, test_sort_keys, test_iterations,
                dyn_resource=dax
            )
    else:
        print(f"Querying the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations
        )

    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
```

```
f"{{(test_end - test_start)/test_iterations}}."  
)
```

Scan the table for a number of iterations for both the DAX client and the Boto3 client and report the time spent for each.

```
import argparse  
import time  
import sys  
import amazondax  
import boto3  
  
  
def scan_test(iterations, dyn_resource=None):  
    """  
    Scans the table a specified number of times. The time before the  
    first iteration and the time after the last iteration are both captured  
    and reported.  
  
    :param iterations: The number of iterations to run.  
    :param dyn_resource: Either a Boto3 or DAX resource.  
    :return: The start and end times of the test.  
    """  
    if dyn_resource is None:  
        dyn_resource = boto3.resource("dynamodb")  
  
    table = dyn_resource.Table("TryDaxTable")  
    start = time.perf_counter()  
    for _ in range(iterations):  
        table.scan()  
        print(".", end="")  
        sys.stdout.flush()  
    print()  
    end = time.perf_counter()  
    return start, end  
  
  
if __name__ == "__main__":  
    # pylint: disable=not-context-manager  
    parser = argparse.ArgumentParser()  
    parser.add_argument(  
        "endpoint_url",
```

```
nargs="?",  
      help="When specified, the DAX cluster endpoint. Otherwise, DAX is not  
used.",  
)  
args = parser.parse_args()  
  
test_iterations = 100  
if args.endpoint_url:  
    print(f"Scanning the table {test_iterations} times, using the DAX  
client.")  
    # Use a with statement so the DAX client closes the cluster after  
completion.  
    with amazondex.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)  
as dax:  
    test_start, test_end = scan_test(test_iterations, dyn_resource=dax)  
else:  
    print(f"Scanning the table {test_iterations} times, using the Boto3  
client.")  
    test_start, test_end = scan_test(test_iterations)  
print(  
    f"Total time: {test_end - test_start:.4f} sec. Average time: "  
    f"{{(test_end - test_start)/test_iterations}}."  
)
```

Delete the table.

```
import boto3  
  
def delete_dax_table(dyn_resource=None):  
    """  
    Deletes the demonstration table.  
  
    :param dyn_resource: Either a Boto3 or DAX resource.  
    """  
    if dyn_resource is None:  
        dyn_resource = boto3.resource("dynamodb")  
  
    table = dyn_resource.Table("TryDaxTable")  
    table.delete()  
  
    print(f"Deleting {table.name}...")
```

```
table.wait_until_not_exists()

if __name__ == "__main__":
    delete_dax_table()
    print("Table deleted!")
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
 - [CreateTable](#)
 - [DeleteTable](#)
 - [GetItem](#)
 - [PutItem](#)
 - [Query](#)
 - [Scan](#)

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Get started with DynamoDB tables, items, and queries using an AWS SDK

The following code examples show how to:

- Create a table that can hold movie data.
- Put, get, and update a single movie in the table.
- Write movie data to the table from a sample JSON file.
- Query for movies that were released in a given year.
- Scan for movies that were released in a range of years.
- Delete a movie from the table, then delete the table.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// This example application performs the following basic Amazon DynamoDB
// functions:
//
//      CreateTableAsync
//      PutItemAsync
//      UpdateItemAsync
//      BatchWriteItemAsync
//      GetItemAsync
//      DeleteItemAsync
//      Query
//      Scan
//      DeleteItemAsync
//
using Amazon.DynamoDBv2;
using DynamoDB_Actions;

public class DynamoDB_Basics
{
    // Separator for the console display.
    private static readonly string SepBar = new string('-', 80);

    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();

        var tableName = "movie_table";

        // Relative path to moviedata.json in the local repository.
        var movieFileName = @"..\..\..\..\..\..\..\resources\sample_files
\movies.json";

        DisplayInstructions();
    }
}
```

```
// Create a new table and wait for it to be active.  
Console.WriteLine($"Creating the new table: {tableName}");  
  
var success = await DynamoDbMethods.CreateMovieTableAsync(client,  
tableName);  
  
if (success)  
{  
    Console.WriteLine($"\\nTable: {tableName} successfully created.");  
}  
else  
{  
    Console.WriteLine($"\\nCould not create {tableName}.");  
}  
  
WaitForEnter();  
  
// Add a single new movie to the table.  
var newMovie = new Movie  
{  
    Year = 2021,  
    Title = "Spider-Man: No Way Home",  
};  
  
success = await DynamoDbMethods.PutItemAsync(client, newMovie,  
tableName);  
if (success)  
{  
    Console.WriteLine($"Added {newMovie.Title} to the table.");  
}  
else  
{  
    Console.WriteLine("Could not add movie to table.");  
}  
  
WaitForEnter();  
  
// Update the new movie by adding a plot and rank.  
var newInfo = new MovieInfo  
{  
    Plot = "With Spider-Man's identity now revealed, Peter asks" +  
          "Doctor Strange for help. When a spell goes wrong, dangerous"  
}
```

+

```
                "foes from other worlds start to appear, forcing Peter to" +
                "discover what it truly means to be Spider-Man.",  
            Rank = 9,  
        };  
  
        success = await DynamoDbMethods.UpdateItemAsync(client, newMovie,  
newInfo, tableName);  
        if (success)  
        {  
            Console.WriteLine($"Successfully updated the movie:  
{newMovie.Title}");  
        }  
        else  
        {  
            Console.WriteLine("Could not update the movie.");  
        }  
  
        WaitForEnter();  
  
        // Add a batch of movies to the DynamoDB table from a list of  
        // movies in a JSON file.  
        var itemCount = await DynamoDbMethods.BatchWriteItemsAsync(client,  
movieFileName);  
        Console.WriteLine($"Added {itemCount} movies to the table.");  
  
        WaitForEnter();  
  
        // Get a movie by key. (partition + sort)  
        var lookupMovie = new Movie  
        {  
            Title = "Jurassic Park",  
            Year = 1993,  
        };  
  
        Console.WriteLine("Looking for the movie \"Jurassic Park\".");  
        var item = await DynamoDbMethods.GetItemAsync(client, lookupMovie,  
tableName);  
        if (item.Count > 0)  
        {  
            DynamoDbMethods.DisplayItem(item);  
        }  
        else  
        {  
            Console.WriteLine($"Couldn't find {lookupMovie.Title}");  
        }
```

```
}

WaitForEnter();

// Delete a movie.
var movieToDelete = new Movie
{
    Title = "The Town",
    Year = 2010,
};

success = await DynamoDbMethods.DeleteItemAsync(client, tableName,
movieToDelete);

if (success)
{
    Console.WriteLine($"Successfully deleted {movieToDelete.Title}.");
}
else
{
    Console.WriteLine($"Could not delete {movieToDelete.Title}.");
}

WaitForEnter();

// Use Query to find all the movies released in 2010.
int findYear = 2010;
Console.WriteLine($"Movies released in {findYear}");
var queryCount = await DynamoDbMethods.QueryMoviesAsync(client,
tableName, findYear);
Console.WriteLine($"Found {queryCount} movies released in {findYear}");

WaitForEnter();

// Use Scan to get a list of movies from 2001 to 2011.
int startYear = 2001;
int endYear = 2011;
var scanCount = await DynamoDbMethods.ScanTableAsync(client, tableName,
startYear, endYear);
Console.WriteLine($"Found {scanCount} movies released between {startYear}
and {endYear}");

WaitForEnter();
```

```
// Delete the table.  
success = await DynamoDbMethods.DeleteTableAsync(client, tableName);  
  
if (success)  
{  
    Console.WriteLine($"Successfully deleted {tableName}");  
}  
else  
{  
    Console.WriteLine($"Could not delete {tableName}");  
}  
  
Console.WriteLine("The DynamoDB Basics example application is done.");  
  
WaitForEnter();  
}  
  
/// <summary>  
/// Displays the description of the application on the console.  
/// </summary>  
private static void DisplayInstructions()  
{  
    Console.Clear();  
    Console.WriteLine();  
    Console.Write(new string(' ', 28));  
    Console.WriteLine("DynamoDB Basics Example");  
    Console.WriteLine(SepBar);  
    Console.WriteLine("This demo application shows the basics of using  
DynamoDB with the AWS SDK.");  
    Console.WriteLine(SepBar);  
    Console.WriteLine("The application does the following:");  
    Console.WriteLine("\t1. Creates a table with partition: year and  
sort:title.");  
    Console.WriteLine("\t2. Adds a single movie to the table.");  
    Console.WriteLine("\t3. Adds movies to the table from moviedata.json.");  
    Console.WriteLine("\t4. Updates the rating and plot of the movie that was  
just added.");  
    Console.WriteLine("\t5. Gets a movie using its key (partition + sort).");  
    Console.WriteLine("\t6. Deletes a movie.");  
    Console.WriteLine("\t7. Uses QueryAsync to return all movies released in  
a given year.");  
    Console.WriteLine("\t8. Uses ScanAsync to return all movies released  
within a range of years.");  
}
```

```
        Console.WriteLine("\t9. Finally, it deletes the table that was just  
created.");  
        WaitForEnter();  
    }  
  
    /// <summary>  
    /// Simple method to wait for the Enter key to be pressed.  
    /// </summary>  
    private static void WaitForEnter()  
{  
    Console.WriteLine("\nPress <Enter> to continue.");  
    Console.WriteLine(SepBar);  
    _ = Console.ReadLine();  
}  
}
```

Creates a table to contain movie data.

```
    /// <summary>  
    /// Creates a new Amazon DynamoDB table and then waits for the new  
    /// table to become active.  
    /// </summary>  
    /// <param name="client">An initialized Amazon DynamoDB client object.</  
param>  
    /// <param name="tableName">The name of the table to create.</param>  
    /// <returns>A Boolean value indicating the success of the operation.</  
returns>  
    public static async Task<bool> CreateMovieTableAsync(AmazonDynamoDBClient  
client, string tableName)  
    {  
        var response = await client.CreateTableAsync(new CreateTableRequest  
        {  
            TableName = tableName,  
            AttributeDefinitions = new List<AttributeDefinition>()  
            {  
                new AttributeDefinition  
                {  
                    AttributeName = "title",  
                    AttributeType = ScalarAttributeType.S,  
                },  
            },  
            KeySchema = new List<KeySchemaElement>()  
            {  
                new KeySchemaElement  
                {  
                    AttributeName = "id",  
                    KeyType = KeyType.Primary  
                },  
            },  
            BillingMode = BillingMode.Payer  
        },  
        CreateTableOptions  
    }  
}
```

```
        new AttributeDefinition
        {
            AttributeName = "year",
            AttributeType = ScalarAttributeType.N,
        },
    },
    KeySchema = new List<KeySchemaElement>()
{
    new KeySchemaElement
    {
        AttributeName = "year",
        KeyType = KeyType.HASH,
    },
    new KeySchemaElement
    {
        AttributeName = "title",
        KeyType = KeyType.RANGE,
    },
},
ProvisionedThroughput = new ProvisionedThroughput
{
    ReadCapacityUnits = 5,
    WriteCapacityUnits = 5,
},
});

// Wait until the table is ACTIVE and then report success.
Console.WriteLine("Waiting for table to become active...");

var request = new DescribeTableRequest
{
    TableName = response.TableDescription.TableName,
};

TableStatus status;

int sleepDuration = 2000;

do
{
    System.Threading.Thread.Sleep(sleepDuration);

    var describeTableResponse = await
client.DescribeTableAsync(request);
```

```
        status = describeTableResponse.Table.TableStatus;

        Console.WriteLine(".");
    }
    while (status != "ACTIVE");

    return status == TableStatus.ACTIVE;
}
```

Adds a single movie to the table.

```
/// <summary>
/// Adds a new item to the table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="newMovie">A Movie object containing information for
/// the movie to add to the table.</param>
/// <param name="tableName">The name of the table where the item will be
added.</param>
/// <returns>A Boolean value that indicates the results of adding the
item.</returns>
public static async Task<bool> PutItemAsync(AmazonDynamoDBClient client,
Movie newMovie, string tableName)
{
    var item = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };

    var request = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
    };

    var response = await client.PutItemAsync(request);
    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

Updates a single item in a table.

```
/// <summary>
/// Updates an existing item in the movies table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</param>
/// <param name="newMovie">A Movie object containing information for the movie to update.</param>
/// <param name="newInfo">A MovieInfo object that contains the information that will be changed.</param>
/// <param name="tableName">The name of the table that contains the movie.</param>
/// <returns>A Boolean value that indicates the success of the operation.</returns>
public static async Task<bool> UpdateItemAsync(
    AmazonDynamoDBClient client,
    Movie newMovie,
    MovieInfo newInfo,
    string tableName)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };
    var updates = new Dictionary<string, AttributeValueUpdate>
    {
        ["info.plot"] = new AttributeValueUpdate
        {
            Action = AttributeAction.PUT,
            Value = new AttributeValue { S = newInfo.Plot },
        },
        ["info.rating"] = new AttributeValueUpdate
        {
            Action = AttributeAction.PUT,
            Value = new AttributeValue { N = newInfo.Rank.ToString() },
        },
    };
}
```

```
};

var request = new UpdateItemRequest
{
    AttributeUpdates = updates,
    Key = key,
    TableName = tableName,
};

var response = await client.UpdateItemAsync(request);

return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

Retrieves a single item from the movie table.

```
/// <summary>
/// Gets information about an existing movie from the table.
/// </summary>
/// <param name="client">An initialized Amazon DynamoDB client object.</
param>
/// <param name="newMovie">A Movie object containing information about
/// the movie to retrieve.</param>
/// <param name="tableName">The name of the table containing the movie.</
param>
/// <returns>A Dictionary object containing information about the item
/// retrieved.</returns>
public static async Task<Dictionary<string, AttributeValue>>
GetItemAsync(AmazonDynamoDBClient client, Movie newMovie, string tableName)
{
    var key = new Dictionary<string, AttributeValue>
    {
        ["title"] = new AttributeValue { S = newMovie.Title },
        ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
    };

    var request = new GetItemRequest
    {
        Key = key,
        TableName = tableName,
```

```
};

        var response = await client.GetItemAsync(request);
        return response.Item;
}
```

Writes a batch of items to the movie table.

```
/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The full path to the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public static List<Movie> ImportMovies(string movieFileName)
{
    if (!File.Exists(movieFileName))
    {
        return null;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonSerializer.Deserialize<List<Movie>>(
        json,
        new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        });
}

// Now return the first 250 entries.
return allMovies.GetRange(0, 250);
}

/// <summary>
/// Writes 250 items to the movie table.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="movieFileName">A string containing the full path to
/// the JSON file containing movie data.</param>
```

```
/// <returns>A long integer value representing the number of movies
/// imported from the JSON file.</returns>
public static async Task<long> BatchWriteItemsAsync(
    AmazonDynamoDBClient client,
    string movieFileName)
{
    var movies = ImportMovies(movieFileName);
    if (movies is null)
    {
        Console.WriteLine("Couldn't find the JSON file with movie
data.");
        return 0;
    }

    var context = new DynamoDBContext(client);

    var movieBatch = context.CreateBatchWrite<Movie>();
    movieBatch.AddPutItems(movies);

    Console.WriteLine("Adding imported movies to the table.");
    await movieBatch.ExecuteAsync();

    return movies.Count;
}
```

Deletes a single item from the table.

```
/// <summary>
/// Deletes a single item from a DynamoDB table.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="tableName">The name of the table from which the item
/// will be deleted.</param>
/// <param name="movieToDelete">A movie object containing the title and
/// year of the movie to delete.</param>
/// <returns>A Boolean value indicating the success or failure of the
/// delete operation.</returns>
public static async Task<bool> DeleteItemAsync(
    AmazonDynamoDBClient client,
    string tableName,
```

```
        Movie movieToDelete)
    {
        var key = new Dictionary<string, AttributeValue>
        {
            ["title"] = new AttributeValue { S = movieToDelete.Title },
            ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
        };

        var request = new DeleteItemRequest
        {
            TableName = tableName,
            Key = key,
        };

        var response = await client.DeleteItemAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}
```

Queries the table for movies released in a particular year.

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="client">The initialized DynamoDB client object.</param>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public static async Task<int> QueryMoviesAsync(AmazonDynamoDBClient
client, string tableName, int year)
{
    var movieTable = Table.LoadTable(client, tableName);
    var filter = new QueryFilter("year", QueryOperator.Equal, year);

    Console.WriteLine("\nFind movies released in: {year}:");

    var config = new QueryOperationConfig()
    {
```

```
        Limit = 10, // 10 items per page.
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string>
        {
            "title",
            "year",
        },
        ConsistentRead = true,
        Filter = filter,
    };

    // Value used to track how many movies match the
    // supplied criteria.
    var moviesFound = 0;

    Search search = movieTable.Query(config);
    do
    {
        var movieList = await search.GetNextSetAsync();
        moviesFound += movieList.Count;

        foreach (var movie in movieList)
        {
            DisplayDocument(movie);
        }
    }
    while (!search.IsDone);

    return moviesFound;
}
```

Scans the table for movies released in a range of years.

```
public static async Task<int> ScanTableAsync(
    AmazonDynamoDBClient client,
    string tableName,
    int startYear,
    int endYear)
{
    var request = new ScanRequest
    {
```

```
        TableName = tableName,
        ExpressionAttributeNames = new Dictionary<string, string>
        {
            { "#yr", "year" },
        },
        ExpressionAttributeValues = new Dictionary<string,
AttributeValue>
        {
            { ":y_a", new AttributeValue { N = startYear.ToString() } },
            { ":y_z", new AttributeValue { N = endYear.ToString() } },
        },
        FilterExpression = "#yr between :y_a and :y_z",
        ProjectionExpression = "#yr, title, info.actors[0],
info.directors, info.running_time_secs",
        Limit = 10 // Set a limit to demonstrate using the
LastEvaluatedKey.
    };

    // Keep track of how many movies were found.
    int foundCount = 0;

    var response = new ScanResponse();
    do
    {
        response = await client.ScanAsync(request);
        foundCount += response.Items.Count;
        response.Items.ForEach(i => DisplayItem(i));
        request.ExclusiveStartKey = response.LastEvaluatedKey;
    }
    while (response.LastEvaluatedKey.Count > 0);
    return foundCount;
}
```

Deletes the movie table.

```
public static async Task<bool> DeleteTableAsync(AmazonDynamoDBClient
client, string tableName)
{
    var request = new DeleteTableRequest
    {
        TableName = tableName,
```

```
};

        var response = await client.DeleteTableAsync(request);
        if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
        {
            Console.WriteLine($"Table {response.TableDescription.TableName} successfully deleted.");
            return true;
        }
        else
        {
            Console.WriteLine("Could not delete table.");
            return false;
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for .NET API Reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)
 - [DeleteItem](#)
 - [DeleteTable](#)
 - [DescribeTable](#)
 - [GetItem](#)
 - [PutItem](#)
 - [Query](#)
 - [Scan](#)
 - [UpdateItem](#)

Bash

AWS CLI with Bash script

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

The DynamoDB getting started scenario.

```
#####
# function dynamodb_getting_started_movies
#
# Scenario to create an Amazon DynamoDB table and perform a series of operations
# on the table.
#
# Returns:
#     0 - If successful.
#     1 - If an error occurred.
#####
function dynamodb_getting_started_movies() {

    source ./dynamodb_operations.sh

    key_schema_json_file="dynamodb_key_schema.json"
    attribute_definitions_json_file="dynamodb_attr_def.json"
    item_json_file="movie_item.json"
    key_json_file="movie_key.json"
    batch_json_file="batch.json"
    attribute_names_json_file="attribute_names.json"
    attributes_values_json_file="attribute_values.json"

    echo_repeat "*" 88
    echo
    echo "Welcome to the Amazon DynamoDB getting started demo."
    echo
    echo_repeat "*" 88
    echo

    local table_name
    echo -n "Enter a name for a new DynamoDB table: "
```

```
get_input
table_name=$get_input_result

local provisioned_throughput="ReadCapacityUnits=5,WriteCapacityUnits=5"

echo '['
{"AttributeName": "year", "KeyType": "HASH"},  
 {"AttributeName": "title", "KeyType": "RANGE"}  
'>"$key_schema_json_file"

echo '['
{"AttributeName": "year", "AttributeType": "N"},  
 {"AttributeName": "title", "AttributeType": "S"}  
'>"$attribute_definitions_json_file"

if dynamodb_create_table -n "$table_name" -a "$attribute_definitions_json_file"  
\  
-k "$key_schema_json_file" -p "$provisioned_throughput" 1>/dev/null; then  
echo "Created a DynamoDB table named $table_name"  
else  
errecho "The table failed to create. This demo will exit."  
clean_up  
return 1  
fi

echo "Waiting for the table to become active...."

if dynamodb_wait_table_active -n "$table_name"; then  
echo "The table is now active."  
else  
errecho "The table failed to become active. This demo will exit."  
cleanup "$table_name"  
return 1  
fi

echo
echo_repeat "*" 88
echo

echo -n "Enter the title of a movie you want to add to the table: "
get_input
local added_title
added_title=$get_input_result
```

```
local added_year
get_int_input "What year was it released? "
added_year=$get_input_result

local rating
get_float_input "On a scale of 1 - 10, how do you rate it? " "1" "10"
rating=$get_input_result

local plot
echo -n "Summarize the plot for me: "
get_input
plot=$get_input_result

echo '{
  "year": {"N" : "'"$added_year"'"},
  "title": {"S" : "'"$added_title"'"},
  "info": {"M" : {"plot": {"S" : "'"$plot"'"}, "rating":
  {"N" :'"$rating"'"} } }
}' >"$item_json_file"

if dynamodb_put_item -n "$table_name" -i "$item_json_file"; then
  echo "The movie '$added_title' was successfully added to the table
'$table_name'."
else
  errecho "Put item failed. This demo will exit."
  clean_up "$table_name"
  return 1
fi

echo
echo_repeat "*" 88
echo

echo "Let's update your movie '$added_title'." 
get_float_input "You rated it $rating, what new rating would you give it? " "1"
"10"
rating=$get_input_result

echo -n "You summarized the plot as '$plot'." 
echo "What would you say now? "
get_input
plot=$get_input_result

echo '{
```

```
"year": {"N" :"'$added_year'"},  
"title": {"S" : "'$added_title'"}  
}' >"$key_json_file"  
  
echo '{  
":r": {"N" :"'$rating'"},  
":p": {"S" : "'$plot'"}  
' >"$item_json_file"  
  
local update_expression="SET info.rating = :r, info.plot = :p"  
  
if dynamodb_update_item -n "$table_name" -k "$key_json_file" -e  
"$update_expression" -v "$item_json_file"; then  
    echo "Updated '$added_title' with new attributes."  
else  
    errecho "Update item failed. This demo will exit."  
    clean_up "$table_name"  
    return 1  
fi  
  
echo  
echo_repeat "*" 88  
echo  
  
echo "We will now use batch write to upload 150 movie entries into the table."  
  
local batch_json  
for batch_json in movie_files/movies_*.json; do  
    echo "{$table_name : $(<">$batch_json") }" >"$batch_json_file"  
    if dynamodb_batch_write_item -i "$batch_json_file" 1>/dev/null; then  
        echo "Entries in $batch_json added to table."  
    else  
        errecho "Batch write failed. This demo will exit."  
        clean_up "$table_name"  
        return 1  
    fi  
done  
  
local title="The Lord of the Rings: The Fellowship of the Ring"  
local year="2001"  
  
if get_yes_no_input "Let's move on...do you want to get info about '$title'?  
(y/n) "; then  
    echo '{
```

```
"year": {"N" :"'$year'"},  
"title": {"S" : "'$title'"}  
}' >"$key_json_file"  
local info  
info=$(dynamodb_get_item -n "$table_name" -k "$key_json_file")  
  
# shellcheck disable=SC2181  
if [[ ${?} -ne 0 ]]; then  
    errecho "Get item failed. This demo will exit."  
    clean_up "$table_name"  
    return 1  
fi  
  
echo "Here is what I found:"  
echo "$info"  
fi  
  
local ask_for_year=true  
while [[ "$ask_for_year" == true ]]; do  
    echo "Let's get a list of movies released in a given year."  
    get_int_input "Enter a year between 1972 and 2018: " "1972" "2018"  
    year=$get_input_result  
    echo '{  
    "#n": "year"  
}' >"$attribute_names_json_file"  
  
    echo '{  
    ":v": {"N" :"'$year'"}  
}' >"$attributes_values_json_file"  
  
    response=$(dynamodb_query -n "$table_name" -k "#n=:v" -a  
"$attribute_names_json_file" -v "$attributes_values_json_file")  
  
    # shellcheck disable=SC2181  
    if [[ ${?} -ne 0 ]]; then  
        errecho "Query table failed. This demo will exit."  
        clean_up "$table_name"  
        return 1  
    fi  
  
    echo "Here is what I found:"  
    echo "$response"  
  
    if ! get_yes_no_input "Try another year? (y/n) "; then
```

```
    ask_for_year=false
  fi
done

echo "Now let's scan for movies released in a range of years. Enter a year: "
get_int_input "Enter a year between 1972 and 2018: " "1972" "2018"
local start=$get_input_result

get_int_input "Enter another year: " "1972" "2018"
local end=$get_input_result

echo '{
  "#n": "year"
}' >"$attribute_names_json_file"

echo '{
  ":v1": {"N" : """$start"""},
  ":v2": {"N" : """$end"""}
}' >"$attributes_values_json_file"

response=$(dynamodb_scan -n "$table_name" -f "#n BETWEEN :v1 AND :v2" -a
"$attribute_names_json_file" -v "$attributes_values_json_file")

# shellcheck disable=SC2181
if [[ ${?} -ne 0 ]]; then
  errecho "Scan table failed. This demo will exit."
  clean_up "$table_name"
  return 1
fi

echo "Here is what I found:"
echo "$response"

echo
echo_repeat "*" 88
echo

echo "Let's remove your movie '$added_title' from the table."

if get_yes_no_input "Do you want to remove '$added_title'? (y/n) "; then
  echo '{
"year": {"N" : """$added_year"""},
"title": {"S" : """$added_title"""}
}' >"$key_json_file"
```

```
if ! dynamodb_delete_item -n "$table_name" -k "$key_json_file"; then
    errecho "Delete item failed. This demo will exit."
    clean_up "$table_name"
    return 1
fi
fi

if get_yes_no_input "Do you want to delete the table '$table_name'? (y/n) ";
then
    if ! clean_up "$table_name"; then
        return 1
    fi
else
    if ! clean_up; then
        return 1
    fi
fi
fi

return 0
}
```

The DynamoDB functions used in this scenario.

```
#####
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table to create.
#     -a attribute_definitions -- JSON file path of a list of attributes and
#       their types.
#     -k key_schema -- JSON file path of a list of attributes and their key
#       types.
#     -p provisioned_throughput -- Provisioned throughput settings for the
#       table.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
```

```
function dynamodb_create_table() {
    local table_name attribute_definitions key_schema provisioned_throughput
    response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_create_table"
    echo "Creates an Amazon DynamoDB table."
    echo " -n table_name -- The name of the table to create."
    echo " -a attribute_definitions -- JSON file path of a list of attributes and
their types."
    echo " -k key_schema -- JSON file path of a list of attributes and their key
types."
    echo " -p provisioned_throughput -- Provisioned throughput settings for the
table."
    echo ""
}

# Retrieve the calling parameters.
while getopt "n:a:k:p:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        a) attribute_definitions="${OPTARG}" ;;
        k) key_schema="${OPTARG}" ;;
        p) provisioned_throughput="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
```

```
        return 1
    fi

    if [[ -z "$attribute_definitions" ]]; then
        errecho "ERROR: You must provide an attribute definitions json file path the
-a parameter."
        usage
        return 1
    fi

    if [[ -z "$key_schema" ]]; then
        errecho "ERROR: You must provide a key schema json file path the -k
parameter."
        usage
        return 1
    fi

    if [[ -z "$provisioned_throughput" ]]; then
        errecho "ERROR: You must provide a provisioned throughput json file path the
-p parameter."
        usage
        return 1
    fi

    iecho "Parameters:\n"
    iecho "    table_name:    $table_name"
    iecho "    attribute_definitions:    $attribute_definitions"
    iecho "    key_schema:    $key_schema"
    iecho "    provisioned_throughput:    $provisioned_throughput"
    iecho ""

response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --key-schema file://"$key_schema" \
    --provisioned-throughput "$provisioned_throughput")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
fi
```

```
    return 0
}

#####
# function dynamodb_describe_table
#
# This function returns the status of a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#
# Response:
#     - TableStatus:
#         And:
#             0 - Table is active.
#             1 - If it fails.
#####

function dynamodb_describe_table {
    local table_name
    local option OPTARG # Required to use getopts command in a function.

#####
# Function usage explanation
#####

function usage() {
    echo "function dynamodb_describe_table"
    echo "Describe the status of a DynamoDB table."
    echo "  -n table_name -- The name of the table."
    echo ""
}

#
# Retrieve the calling parameters.
while getopts "n:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}";;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
    esac
done
```

```
;;
esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

local table_status
table_status=$((
    aws dynamodb describe-table \
    --table-name "$table_name" \
    --output text \
    --query 'Table.TableStatus'
))

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log "$error_code"
    errecho "ERROR: AWS reports describe-table operation failed.$table_status"
    return 1
fi

echo "$table_status"

return 0
}

#####
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -i item -- Path to json file containing the item values.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
```

```
#####
function dynamodb_put_item() {
    local table_name item response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -i item -- Path to json file containing the item values."
    echo ""
}

while getopt "n:i:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        i) item="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?) 
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
fi
```

```
iecho "Parameters:\n"
iecho "    table_name:  $table_name"
iecho "    item:      $item"
iecho ""
iecho ""

response=$(aws dynamodb put-item \
--table-name "$table_name" \
--item file://"$item")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
  aws_cli_error_log $error_code
  errecho "ERROR: AWS reports put-item operation failed.$response"
  return 1
fi

return 0

}

#####
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to update.
#     -e update expression -- An expression that defines one or more
#                           attributes to be updated.
#     -v values -- Path to json file containing the update values.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_update_item() {
  local table_name keys update_expression values response
  local option OPTARG # Required to use getopt command in a function.
```

```
#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_update_item"
    echo "Update an item in a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -k keys -- Path to json file containing the keys that identify the
item to update."
    echo " -e update_expression -- An expression that defines one or more
attributes to be updated."
    echo " -v values -- Path to json file containing the update values."
    echo ""
}

while getopts "n:k:e:v:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        k) keys="${OPTARG}" ;;
        e) update_expression="${OPTARG}" ;;
        v) values="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
```

```
        return 1
    fi
    if [[ -z "$update_expression" ]]; then
        errecho "ERROR: You must provide an update expression with the -e parameter."
        usage
        return 1
    fi

    if [[ -z "$values" ]]; then
        errecho "ERROR: You must provide a values json file path the -v parameter."
        usage
        return 1
    fi

    iecho "Parameters:\n"
    iecho "    table_name: $table_name"
    iecho "    keys: $keys"
    iecho "    update_expression: $update_expression"
    iecho "    values: $values"

    response=$(aws dynamodb update-item \
        --table-name "$table_name" \
        --key file://"$keys" \
        --update-expression "$update_expression" \
        --expression-attribute-values file://"$values")

    local error_code=${?}

    if [[ $error_code -ne 0 ]]; then
        aws_cli_error_log $error_code
        errecho "ERROR: AWS reports update-item operation failed.$response"
        return 1
    fi

    return 0
}

#####
# function dynamodb_batch_write_item
#
# This function writes a batch of items into a DynamoDB table.
#
# Parameters:
```

```
#      -i item  -- Path to json file containing the items to write.
#
# Returns:
#      0 - If successful.
#      1 - If it fails.
#####
function dynamodb_batch_write_item() {
    local item response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_batch_write_item"
    echo "Write a batch of items into a DynamoDB table."
    echo " -i item  -- Path to json file containing the items to write."
    echo ""
}

while getopts "i:h" option; do
    case "${option}" in
        i) item="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    item: $item"
```

```
iecho ""

response=$(aws dynamodb batch-write-item \
--request-items file://"$item")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
fi

return 0
}

#####
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to get.
#     [-q query] -- Optional JMESPath query expression.
#
# Returns:
#     The item as text output.
# And:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_get_item() {
    local table_name keys query response
    local option OPTARG # Required to use getopt command in a function.

    #
    # Function usage explanation
    #

    function usage() {
        echo "function dynamodb_get_item"
        echo "Get an item from a DynamoDB table."
        echo " -n table_name -- The name of the table."
    }
}
```

```
echo " -k keys -- Path to json file containing the keys that identify the
item to get."
echo " [-q query] -- Optional JMESPath query expression."
echo ""
}
query=""
while getopts "n:k:q:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    k) keys="${OPTARG}" ;;
    q) query="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?) 
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi

if [[ -n "$query" ]]; then
  response=$(aws dynamodb get-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --output text \
    --query "$query")
else
  response=$(
```

```
aws dynamodb get-item \
--table-name "$table_name" \
--key file://"${keys}" \
--output text
)
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
fi

if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\\t/s/\\t//1" # Remove initial tab that the JMSEPath
query inserts on some strings.
else
    echo "$response"
fi

return 0
}

#####
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k key_condition_expression -- The key condition expression.
#     -a attribute_names -- Path to JSON file containing the attribute names.
#     -v attribute_values -- Path to JSON file containing the attribute values.
#     [-p projection_expression] -- Optional projection expression.
#
# Returns:
#     The items as json output.
# And:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_query() {
```

```
local table_name key_condition_expression attribute_names attribute_values
projection_expression response
local option OPTARG # Required to use getopt command in a function.

# ######
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_query"
    echo "Query a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -k key_condition_expression -- The key condition expression."
    echo " -a attribute_names -- Path to JSON file containing the attribute
names."
    echo " -v attribute_values -- Path to JSON file containing the attribute
values."
    echo " [-p projection_expression] -- Optional projection expression."
    echo ""
}

while getopt "n:k:a:v:p:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        k) key_condition_expression="${OPTARG}" ;;
        a) attribute_names="${OPTARG}" ;;
        v) attribute_values="${OPTARG}" ;;
        p) projection_expression="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
```

```
fi

if [[ -z "$key_condition_expression" ]]; then
    errecho "ERROR: You must provide a key condition expression with the -k
parameter."
    usage
    return 1
fi

if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
fi

if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
fi

if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
        --table-name "$table_name" \
        --key-condition-expression "$key_condition_expression" \
        --expression-attribute-names file://"$attribute_names" \
        --expression-attribute-values file://"$attribute_values")
else
    response=$(aws dynamodb query \
        --table-name "$table_name" \
        --key-condition-expression "$key_condition_expression" \
        --expression-attribute-names file://"$attribute_names" \
        --expression-attribute-values file://"$attribute_values" \
        --projection-expression "$projection_expression")
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
fi
```

```
echo "$response"

return 0
}

#####
# function dynamodb_scan
#
# This function scans a DynamoDB table.
#
# Parameters:
#   -n table_name -- The name of the table.
#   -f filter_expression -- The filter expression.
#   -a expression_attribute_names -- Path to JSON file containing the
#   expression attribute names.
#   -v expression_attribute_values -- Path to JSON file containing the
#   expression attribute values.
#   [-p projection_expression] -- Optional projection expression.
#
# Returns:
#   The items as json output.
# And:
#   0 - If successful.
#   1 - If it fails.
#####

function dynamodb_scan() {
    local table_name filter_expression expression_attribute_names
    expression_attribute_values projection_expression response
    local option OPTARG # Required to use getopt command in a function.

    # #####
    # Function usage explanation
    #####
    function usage() {
        echo "function dynamodb_scan"
        echo "Scan a DynamoDB table."
        echo " -n table_name -- The name of the table."
        echo " -f filter_expression -- The filter expression."
        echo " -a expression_attribute_names -- Path to JSON file containing the
        expression attribute names."
        echo " -v expression_attribute_values -- Path to JSON file containing the
        expression attribute values."
        echo " [-p projection_expression] -- Optional projection expression."
        echo ""
    }
}
```

```
}

while getopts "n:f:a:v:p:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        f) filter_expression="${OPTARG}" ;;
        a) expression_attribute_names="${OPTARG}" ;;
        v) expression_attribute_values="${OPTARG}" ;;
        p) projection_expression="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$filter_expression" ]]; then
    errecho "ERROR: You must provide a filter expression with the -f parameter."
    usage
    return 1
fi

if [[ -z "$expression_attribute_names" ]]; then
    errecho "ERROR: You must provide expression attribute names with the -a parameter."
    usage
    return 1
fi

if [[ -z "$expression_attribute_values" ]]; then
    errecho "ERROR: You must provide expression attribute values with the -v parameter."
```

```
usage
return 1
fi

if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb scan \
        --table-name "$table_name" \
        --filter-expression "$filter_expression" \
        --expression-attribute-names file://"$expression_attribute_names" \
        --expression-attribute-values file://"$expression_attribute_values")
else
    response=$(aws dynamodb scan \
        --table-name "$table_name" \
        --filter-expression "$filter_expression" \
        --expression-attribute-names file://"$expression_attribute_names" \
        --expression-attribute-values file://"$expression_attribute_values" \
        --projection-expression "$projection_expression")
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports scan operation failed.$response"
    return 1
fi

echo "$response"

return 0
}

#####
# function dynamodb_delete_item
#
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table.
#     -k keys -- Path to json file containing the keys that identify the item
#               to delete.
#
# Returns:
#     0 - If successful.
```

```
#      1 - If it fails.
#####
function dynamodb_delete_item() {
    local table_name keys response
    local option OPTARG # Required to use getopt command in a function.

#####
# Function usage explanation
#####
function usage() {
    echo "function dynamodb_delete_item"
    echo "Delete an item from a DynamoDB table."
    echo " -n table_name -- The name of the table."
    echo " -k keys -- Path to json file containing the keys that identify the
item to delete."
    echo ""
}
while getopt "n:k:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        k) keys="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
```

```
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho "    keys: $keys"
iecho ""

response=$(aws dynamodb delete-item \
--table-name "$table_name" \
--key file://"$keys")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
fi

return 0

}

#####
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
#     -n table_name -- The name of the table to delete.
#
# Returns:
#     0 - If successful.
#     1 - If it fails.
#####
function dynamodb_delete_table() {
    local table_name response
    local option OPTARG # Required to use getopts command in a function.

    # bashsupport disable=BP5008
    function usage() {
        echo "function dynamodb_delete_table"
        echo "Deletes an Amazon DynamoDB table."
        echo " -n table_name -- The name of the table to delete."
    }
}
```

```
echo ""
}

# Retrieve the calling parameters.
while getopts "n:h" option; do
    case "${option}" in
        n) table_name="${OPTARG}" ;;
        h)
            usage
            return 0
            ;;
        \?)
            echo "Invalid parameter"
            usage
            return 1
            ;;
    esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
fi

iecho "Parameters:\n"
iecho "    table_name: $table_name"
iecho ""

response=$(aws dynamodb delete-table \
    --table-name "$table_name")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
fi

return 0
}
```

The utility functions used in this scenario.

```
#####
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#####
function iecho() {
    if [[ $VERBOSE == true ]]; then
        echo "$@"
    fi
}

#####
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####
function errecho() {
    printf "%s\n" "$*" 1>&2
}

#####
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
# help-return-codes.
#
# The function expects the following argument:
#     $1 - The error code returned by the AWS CLI.
#
# Returns:
#     0: - Success.
#
#####
function aws_cli_error_log() {
    local err_code=$1
    errecho "Error code : $err_code"
```

```
if [ "$err_code" == 1 ]; then
    errecho " One or more S3 transfers failed."
elif [ "$err_code" == 2 ]; then
    errecho " Command line failed to parse."
elif [ "$err_code" == 130 ]; then
    errecho " Process received SIGINT."
elif [ "$err_code" == 252 ]; then
    errecho " Command syntax invalid."
elif [ "$err_code" == 253 ]; then
    errecho " The system environment or configuration was invalid."
elif [ "$err_code" == 254 ]; then
    errecho " The service returned an error."
elif [ "$err_code" == 255 ]; then
    errecho " 255 is a catch-all error."
fi

return 0
}
```

- For API details, see the following topics in *AWS CLI Command Reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)
 - [DeleteItem](#)
 - [DeleteTable](#)
 - [DescribeTable](#)
 - [GetItem](#)
 - [PutItem](#)
 - [Query](#)
 - [Scan](#)
 - [UpdateItem](#)

C++

SDK for C++**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
{  
    Aws::Client::ClientConfiguration clientConfig;  
    // 1. Create a table with partition: year (N) and sort: title (S).  
    CreateTable()  
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {  
  
            AwsDoc::DynamoDB::dynamodbGettingStartedScenario(clientConfig);  
  
            // 9. Delete the table. (DeleteTable)  
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);  
        }  
    }  
  
//! Scenario to modify and query a DynamoDB table.  
/*!  
 \sa dynamodbGettingStartedScenario()  
 \param clientConfiguration: AWS client configuration.  
 \return bool: Function succeeded.  
 */  
bool AwsDoc::DynamoDB::dynamodbGettingStartedScenario(  
    const Aws::Client::ClientConfiguration &clientConfiguration) {  
    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "  
        << std::endl;  
    std::cout << "Welcome to the Amazon DynamoDB getting started demo." <<  
    std::endl;  
    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "  
        << std::endl;  
  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
  
    // 2. Add a new movie.  
    Aws::String title;
```

```
float rating;
int year;
Aws::String plot;
{
    title = askQuestion(
        "Enter the title of a movie you want to add to the table: ");
    year = askQuestionForInt("What year was it released? ");
    rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                    1, 10);
    plot = askQuestion("Summarize the plot for me: ");

    Aws::DynamoDB::Model::PutItemRequest putItemRequest;
    putItemRequest.SetTableName(MOVIE_TABLE_NAME);

    putItemRequest.AddItem(YEAR_KEY,
                           Aws::DynamoDB::Model::AttributeValue().SetN(year));
    putItemRequest.AddItem(TITLE_KEY,
                           Aws::DynamoDB::Model::AttributeValue().SetS(title));

    // Create attribute for the info map.
    Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

    std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
    Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
        ALLOCATION_TAG.c_str());
    ratingAttribute->SetN(rating);
    infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

    std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
    Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
        ALLOCATION_TAG.c_str());
    plotAttribute->SetS(plot);
    infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);

    putItemRequest.AddItem(INFO_KEY, infoMapAttribute);

    Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
        putItemRequest);
    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to add an item: " <<
        outcome.GetError().GetMessage()
```

```
        << std::endl;
    return false;
}

std::cout << "\nAdded '" << title << "' to '" << MOVIE_TABLE_NAME << "'."
    << std::endl;

// 3. Update the rating and plot of the movie by using an update expression.
{
    rating = askQuestionForFloatRange(
        Aws::String("\nLet's update your movie.\nYou rated it " +
        std::to_string(rating)
        + ", what new rating would you give it? ", 1, 10);
    plot = askQuestion(Aws::String("You summarized the plot as '") + plot +
        "'.\nWhat would you say now? ");

    Aws::DynamoDB::Model::UpdateItemRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);
    request.AddKey(TITLE_KEY,
        Aws::DynamoDB::Model::AttributeValue().SetS(title));
    request.AddKey(YEAR_KEY,
        Aws::DynamoDB::Model::AttributeValue().SetN(year));
    std::stringstream expressionStream;
    expressionStream << "set " << INFO_KEY << "." << RATING_KEY << " :=r, "
        << INFO_KEY << "." << PLOT_KEY << " :=p";
    request.SetUpdateExpression(expressionStream.str());
    request.SetExpressionAttributeValues({
        {"":r",
        Aws::DynamoDB::Model::AttributeValue().SetN(
            rating)},
        {"":p",
        Aws::DynamoDB::Model::AttributeValue().SetS(
            plot)}
    });

    request.SetReturnValues(Aws::DynamoDB::Model::ReturnValue::UPDATED_NEW);

    const Aws::DynamoDB::Model::UpdateItemOutcome &result =
dynamoClient.UpdateItem(
        request);
    if (!result.IsSuccess()) {
        std::cerr << "Error updating movie " + result.GetError().GetMessage()
            << std::endl;
    }
}
```

```
        return false;
    }
}

std::cout << "\nUpdated '" << title << "' with new attributes:" << std::endl;

// 4. Put 250 movies in the table from moviedata.json.
{
    std::cout << "Adding movies from a json file to the database." <<
std::endl;
    const size_t MAX_SIZE_FOR_BATCH_WRITE = 25;
    const size_t MOVIES_TO_WRITE = 10 * MAX_SIZE_FOR_BATCH_WRITE;
    Aws::String jsonString = getMovieJSON();
    if (!jsonString.empty()) {
        Aws::Utils::Json::JsonValue json(jsonString);
        Aws::Utils::Array<Aws::Utils::Json::JsonValue> movieJsons =
json.View().AsArray();
        Aws::Vector<Aws::DynamoDB::Model::WriteRequest> writeRequests;

        // To add movies with a cross-section of years, use an appropriate
increment
        // value for iterating through the database.
        size_t increment = movieJsons.GetLength() / MOVIES_TO_WRITE;
        for (size_t i = 0; i < movieJsons.GetLength(); i += increment) {
            writeRequests.push_back(Aws::DynamoDB::Model::WriteRequest());
            Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
putItems = movieJsonViewToAttributeMap(
                movieJsons[i]);
            Aws::DynamoDB::Model::PutRequest putRequest;
            putRequest.SetItem(putItems);
            writeRequests.back().SetPutRequest(putRequest);
            if (writeRequests.size() == MAX_SIZE_FOR_BATCH_WRITE) {
                Aws::DynamoDB::Model::BatchWriteItemRequest request;
                request.AddRequestItems(MOVIE_TABLE_NAME, writeRequests);
                const Aws::DynamoDB::Model::BatchWriteItemOutcome &outcome =
dynamoClient.BatchWriteItem(
                    request);
                if (!outcome.IsSuccess()) {
                    std::cerr << "Unable to batch write movie data: "
                        << outcome.GetError().GetMessage()
                        << std::endl;
                    writeRequests.clear();
                    break;
                }
            }
        }
    }
}
```

```
        else {
            std::cout << "Added batch of " << writeRequests.size()
                << " movies to the database."
                << std::endl;
        }
        writeRequests.clear();
    }
}

std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
    << std::endl;

// 5. Get a movie by Key (partition + sort).
{
    Aws::String titleToGet("King Kong");
    Aws::String answer = askQuestion(Aws::String(
        "Let's move on...Would you like to get info about '" + titleToGet
+
        "'? (y/n) "));
    if (answer == "y") {
        Aws::DynamoDB::Model::GetItemRequest request;
        request.SetTableName(MOVIE_TABLE_NAME);
        request.AddKey(TITLE_KEY,
            Aws::DynamoDB::Model::AttributeValue().SetS(titleToGet));
        request.AddKey(YEAR_KEY,
            Aws::DynamoDB::Model::AttributeValue().SetN(1933));

        const Aws::DynamoDB::Model::GetItemOutcome &result =
dynamoClient.GetItem(
            request);
        if (!result.IsSuccess()) {
            std::cerr << "Error " << result.GetError().GetMessage();
        }
        else {
            const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = result.GetResult().GetItem();
            if (!item.empty()) {
                std::cout << "\nHere's what I found:" << std::endl;
                printMovieInfo(item);
            }
            else {
```

```
        std::cout << "\nThe movie was not found in the database."
        << std::endl;
    }
}
}

// 6. Use Query with a key condition expression to return all movies
//     released in a given year.
Aws::String doAgain = "n";
do {
    Aws::DynamoDB::Model::QueryRequest req;

    req.SetTableName(MOVIE_TABLE_NAME);

    // "year" is a DynamoDB reserved keyword and must be replaced with an
    // expression attribute name.
    req.SetKeyConditionExpression("#dynobase_year = :valueToMatch");
    req.SetExpressionAttributeNames({{"#dynobase_year", YEAR_KEY}});

    int yearToMatch = askQuestionForIntRange(
        "\nLet's get a list of movies released in"
        " a given year. Enter a year between 1972 and 2018 ",
        1972, 2018);
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
attributeValues;
    attributeValues.emplace(":valueToMatch",
                           Aws::DynamoDB::Model::AttributeValue().SetN(
                               yearToMatch));
    req.SetExpressionAttributeValues(attributeValues);

    const Aws::DynamoDB::Model::QueryOutcome &result =
dynamoClient.Query(req);
    if (result.IsSuccess()) {
        const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetResult().GetItems();
        if (!items.empty()) {
            std::cout << "\nThere were " << items.size()
                << " movies in the database from "
                << yearToMatch << "." << std::endl;
            for (const auto &item: items) {
                printMovieInfo(item);
            }
            doAgain = "n";
        }
    }
}
```

```
        }

        else {
            std::cout << "\nNo movies from " << yearToMatch
                << " were found in the database"
                << std::endl;
            doAgain = askQuestion(Aws::String("Try another year? (y/n) "));
        }
    }
    else {
        std::cerr << "Failed to Query items: " <<
result.GetError().GetMessage()
                << std::endl;
    }

} while (doAgain == "y");

// 7. Use Scan to return movies released within a range of years.
//      Show how to paginate data using ExclusiveStartKey. (Scan +
FilterExpression)
{
    int startYear = askQuestionForIntRange("\nNow let's scan a range of years
"
                                         "for movies in the database. Enter
a start year: ",
                                         1972, 2018);
    int endYear = askQuestionForIntRange("\nEnter an end year: ",
                                         startYear, 2018);
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
    do {
        Aws::DynamoDB::Model::ScanRequest scanRequest;
        scanRequest.SetTableName(MOVIE_TABLE_NAME);
        scanRequest.SetFilterExpression(
            "#dynobase_year >= :startYear AND #dynobase_year
<= :endYear");
        scanRequest.SetExpressionAttributeNames({{"#dynobase_year",
YEAR_KEY}});
    }

    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
attributeValues;
    attributeValues.emplace(":startYear",
                           Aws::DynamoDB::Model::AttributeValue().SetN(
                               startYear));
    attributeValues.emplace(":endYear",
```

```
        Aws::DynamoDB::Model::AttributeValue().SetN(
            endYear));
    scanRequest.SetExpressionAttributeValues(attributeValues);

    if (!exclusiveStartKey.empty()) {
        scanRequest.SetExclusiveStartKey(exclusiveStartKey);
    }

    const Aws::DynamoDB::Model::ScanOutcome &result = dynamoClient.Scan(
        scanRequest);
    if (result.IsSuccess()) {
        const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetResult().GetItems();
        if (!items.empty()) {
            std::stringstream stringStream;
            stringStream << "\nFound " << items.size() << " movies in one
scan."
                << " How many would you like to see? ";
            size_t count = askQuestionForInt(stringStream.str());
            for (size_t i = 0; i < count && i < items.size(); ++i) {
                printMovieInfo(items[i]);
            }
        }
        else {
            std::cout << "\nNo movies in the database between " <<
startYear <<
                " and " << endYear << "." << std::endl;
        }
    }

    exclusiveStartKey = result.GetResult().GetLastEvaluatedKey();
    if (!exclusiveStartKey.empty()) {
        std::cout << "Not all movies were retrieved. Scanning for
more."
                << std::endl;
    }
    else {
        std::cout << "All movies were retrieved with this scan."
                << std::endl;
    }
}
else {
    std::cerr << "Failed to Scan movies: "
        << result.GetError().GetMessage() << std::endl;
}
```

```
        } while (!exclusiveStartKey.empty());
    }

    // 8. Delete a movie. (DeleteItem)
    {
        std::stringstream stringStream;
        stringStream << "\nWould you like to delete the movie " << title
                      << " from the database? (y/n) ";
        Aws::String answer = askQuestion(stringStream.str());
        if (answer == "y") {
            Aws::DynamoDB::Model::DeleteItemRequest request;
            request.AddKey(YEAR_KEY,
                           Aws::DynamoDB::Model::AttributeValue().SetN(year));
            request.AddKey(TITLE_KEY,
                           Aws::DynamoDB::Model::AttributeValue().SetS(title));
            request.SetTableName(MOVIE_TABLE_NAME);

            const Aws::DynamoDB::Model::DeleteItemOutcome &result =
dynamoClient.DeleteItem(
                request);
            if (result.IsSuccess()) {
                std::cout << "\nRemoved \"\" << title << "\" from the database."
                           << std::endl;
            }
            else {
                std::cerr << "Failed to delete the movie: "
                           << result.GetError().GetMessage()
                           << std::endl;
            }
        }
    }

    return true;
}

//! Routine to convert a JsonView object to an attribute map.
/*!
 \sa movieJsonViewToAttributeMap()
 \param jsonView: Json view object.
 \return map: Map that can be used in a DynamoDB request.
 */
Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
AwsDoc::DynamoDB::movieJsonViewToAttributeMap(
    const Aws::Utils::Json::JsonView &jsonView) {
```

```
Aws::Map< Aws::String, Aws::DynamoDB::Model::AttributeValue> result;

if (jsonView.KeyExists(YEAR_KEY)) {
    result[YEAR_KEY].SetN(jsonView.GetInteger(YEAR_KEY));
}
if (jsonView.KeyExists(TITLE_KEY)) {
    result[TITLE_KEY].SetS(jsonView.GetString(TITLE_KEY));
}
if (jsonView.KeyExists(INFO_KEY)) {
    Aws::Map< Aws::String, const
std::shared_ptr< Aws::DynamoDB::Model::AttributeValue>> infoMap;
    Aws::Utils::Json::JsonValue infoView = jsonView.GetObject(INFO_KEY);
    if (infoView.KeyExists(RATING_KEY)) {
        std::shared_ptr< Aws::DynamoDB::Model::AttributeValue> attributeValue
= std::make_shared< Aws::DynamoDB::Model::AttributeValue>();
        attributeValue->SetN(infoView.GetDouble(RATING_KEY));
        infoMap.emplace(std::make_pair(RATING_KEY, attributeValue));
    }
    if (infoView.KeyExists(PLOT_KEY)) {
        std::shared_ptr< Aws::DynamoDB::Model::AttributeValue> attributeValue
= std::make_shared< Aws::DynamoDB::Model::AttributeValue>();
        attributeValue->SetS(infoView.GetString(PLOT_KEY));
        infoMap.emplace(std::make_pair(PLOT_KEY, attributeValue));
    }
}

result[INFO_KEY].SetM(infoMap);
}

return result;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*! \sa createMoviesDynamoDBTable()
 * \param clientConfiguration: AWS client configuration.
 * \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;

    {
```

```
Aws::DynamoDB::Model::CreateTableRequest request;

Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
yearAttributeDefinition.SetAttributeName(YEAR_KEY);
yearAttributeDefinition.SetAttributeType(
    Aws::DynamoDB::Model::ScalarAttributeType::N);
request.AddAttributeDefinitions(yearAttributeDefinition);

Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
yearAttributeDefinition.SetAttributeName(TITLE_KEY);
yearAttributeDefinition.SetAttributeType(
    Aws::DynamoDB::Model::ScalarAttributeType::S);
request.AddAttributeDefinitions(titleAttributeDefinition);

Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
    Aws::DynamoDB::Model::KeyType::HASH);
request.AddKeySchema(yearKeySchema);

Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
    Aws::DynamoDB::Model::KeyType::RANGE);
request.AddKeySchema(titleKeySchema);

Aws::DynamoDB::Model::ProvisionedThroughput throughput;
throughput.WithReadCapacityUnits(
    PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
    PROVISIONED_THROUGHPUT_UNITS);
request.SetProvisionedThroughput(throughput);
request.SetTableName(MOVIE_TABLE_NAME);

std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
    request);
if (!result.IsSuccess()) {
    if (result.GetError().GetErrorCode() ==
        Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
        std::cout << "Table already exists." << std::endl;
        movieTableAlreadyExisted = true;
    }
    else {
        std::cerr << "Failed to create table: "
    }
}
```

```
                << result.GetError().GetMessage());
        return false;
    }
}

// Wait for table to become active.
if (!movieTableAlreadyExisted) {
    std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
        << "' to become active...." << std::endl;
    if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
clientConfiguration)) {
        return false;
    }
    std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
        << std::endl;
}

return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*! \sa deleteMoviesDynamoDBTable()
 * \param clientConfiguration: AWS client configuration.
 * \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
dynamoClient.DeleteTable(
        request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
            << result.GetResult().GetTableDescription().GetTableName()
            << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
```

```
        << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*! \sa waitTableActive()
 * \param waitTableActive: The DynamoDB table's name.
 * \param clientConfiguration: AWS client configuration.
 * \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                         const Aws::Client::ClientConfiguration
                                         &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
dynamoClient.DescribeTable(
            request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                  << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
}
```

```
    }  
    return false;  
}
```

- For API details, see the following topics in *AWS SDK for C++ API Reference*.

- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

Go

SDK for Go V2

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run an interactive scenario to create the table and perform actions on it.

```
// RunMovieScenario is an interactive example that shows you how to use the AWS  
// SDK for Go  
// to create and use an Amazon DynamoDB table that stores data about movies.  
//  
// 1. Create a table that can hold movie data.  
// 2. Put, get, and update a single movie in the table.  
// 3. Write movie data to the table from a sample JSON file.
```

```
// 4. Query for movies that were released in a given year.  
// 5. Scan for movies that were released in a range of years.  
// 6. Delete a movie from the table.  
// 7. Delete the table.  
  
//  
// This example creates a DynamoDB service client from the specified sdkConfig so  
// that  
// you can replace it with a mocked or stubbed config for unit testing.  
//  
// It uses a questioner from the `demotools` package to get input during the  
// example.  
// This package can be found in the ..\..\demotools folder of this repo.  
//  
// The specified movie sampler is used to get sample data from a URL that is  
// loaded  
// into the named table.  
func RunMovieScenario(  
    sdkConfig aws.Config, questioner demotools.IQuestioner, tableName string,  
    movieSampler actions.IMovieSampler) {  
    defer func() {  
        if r := recover(); r != nil {  
            fmt.Printf("Something went wrong with the demo.")  
        }  
    }()  
  
    log.Println(strings.Repeat("-", 88))  
    log.Println("Welcome to the Amazon DynamoDB getting started demo.")  
    log.Println(strings.Repeat("-", 88))  
  
    tableBasics := actions.TableBasics{TableName: tableName,  
        DynamoDbClient: dynamodb.NewFromConfig(sdkConfig)}  
  
    exists, err := tableBasics.TableExists()  
    if err != nil {  
        panic(err)  
    }  
    if !exists {  
        log.Printf("Creating table %v...\n", tableName)  
        _, err = tableBasics.CreateMovieTable()  
        if err != nil {  
            panic(err)  
        } else {  
            log.Printf("Created table %v.\n", tableName)  
        }  
    }  
}
```

```
    } else {
        log.Printf("Table %v already exists.\n", tableName)
    }

    var customMovie actions.Movie
    customMovie.Title = questioner.Ask("Enter a movie title to add to the table:",
        []demotools.IAnswerValidator{demotools.NotEmpty{}})
    customMovie.Year = questioner.AskInt("What year was it released?",
        []demotools.IAnswerValidator{demotools.NotEmpty{}, demotools.InIntRange{
            Lower: 1900, Upper: 2030}})
    customMovie.Info = map[string]interface{}{}
    customMovie.Info["rating"] = questioner.AskFloat64(
        "Enter a rating between 1 and 10:", []demotools.IAnswerValidator{
            demotools.NotEmpty{}, demotools.InFloatRange{Lower: 1, Upper: 10}})
    customMovie.Info["plot"] = questioner.Ask("What's the plot? ",
        []demotools.IAnswerValidator{demotools.NotEmpty{}})
    err = tableBasics.AddMovie(customMovie)
    if err == nil {
        log.Printf("Added %v to the movie table.\n", customMovie.Title)
    }
    log.Println(strings.Repeat("-", 88))

    log.Printf("Let's update your movie. You previously rated it %v.\n",
        customMovie.Info["rating"])
    customMovie.Info["rating"] = questioner.AskFloat64(
        "What new rating would you give it?", []demotoools.IAnswerValidator{
            demotools.NotEmpty{}, demotoools.InFloatRange{Lower: 1, Upper: 10}})
    log.Printf("You summarized the plot as '%v'.\n", customMovie.Info["plot"])
    customMovie.Info["plot"] = questioner.Ask("What would you say now?",
        []demotools.IAnswerValidator{demotools.NotEmpty{}})
    attributes, err := tableBasics.UpdateMovie(customMovie)
    if err == nil {
        log.Printf("Updated %v with new values.\n", customMovie.Title)
        for _, attVal := range attributes {
            for valKey, val := range attVal {
                log.Printf("\t%v: %v\n", valKey, val)
            }
        }
    }
    log.Println(strings.Repeat("-", 88))

    log.Printf("Getting movie data from %v and adding 250 movies to the table...\n",
        movieSampler.GetURL())
    movies := movieSampler.GetSampleMovies()
```

```
written, err := tableBasics.AddMovieBatch(movies, 250)
if err != nil {
    panic(err)
} else {
    log.Printf("Added %v movies to the table.\n", written)
}

show := 10
if show > written {
    show = written
}
log.Printf("The first %v movies in the table are:", show)
for index, movie := range movies[:show] {
    log.Printf("\t%v. %v\n", index+1, movie.Title)
}
movieIndex := questioner.AskInt(
    "Enter the number of a movie to get info about it: ",
    []demotools.IAnswerValidator{
        demotools.InIntRange{Lower: 1, Upper: show},
    })
movie, err := tableBasics.GetMovie(movies[movieIndex-1].Title,
    movies[movieIndex-1].Year)
if err == nil {
    log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

log.Println("Let's get a list of movies released in a given year.")
releaseYear := questioner.AskInt("Enter a year between 1972 and 2018: ",
    []demotools.IAnswerValidator{demotoools.InIntRange{Lower: 1972, Upper: 2018}},
)
releases, err := tableBasics.Query(releaseYear)
if err == nil {
    if len(releases) == 0 {
        log.Printf("I couldn't find any movies released in %v!\n", releaseYear)
    } else {
        for _, movie = range releases {
            log.Println(movie)
        }
    }
}
log.Println(strings.Repeat("-", 88))

log.Println("Now let's scan for movies released in a range of years.")
```

```
startYear := questioner.AskInt("Enter a year: ", []demotools.IAnswerValidator{
    demotools.InIntRange{Lower: 1972, Upper: 2018}})
endYear := questioner.AskInt("Enter another year: ",
    []demotools.IAnswerValidator{
        demotools.InIntRange{Lower: 1972, Upper: 2018}})
releases, err = tableBasics.Scan(startYear, endYear)
if err == nil {
    if len(releases) == 0 {
        log.Printf("I couldn't find any movies released between %v and %v!\n",
            startYear, endYear)
    } else {
        log.Printf("Found %v movies. In this list, the plot is <nil> because "+
            "we used a projection expression when scanning for items to return only "+
            "the title, year, and rating.\n", len(releases))
        for _, movie := range releases {
            log.Println(movie)
        }
    }
}
log.Println(strings.Repeat("-", 88))

var tables []string
if questioner.AskBool("Do you want to list all of your tables? (y/n) ", "y") {
    tables, err = tableBasics.ListTables()
    if err == nil {
        log.Printf("Found %v tables:", len(tables))
        for _, table := range tables {
            log.Printf("\t%v", table)
        }
    }
}
log.Println(strings.Repeat("-", 88))

log.Printf("Let's remove your movie '%v'.\n", customMovie.Title)
if questioner.AskBool("Do you want to delete it from the table? (y/n) ", "y") {
    err = tableBasics.DeleteMovie(customMovie)
}
if err == nil {
    log.Printf("Deleted %v.\n", customMovie.Title)
}

if questioner.AskBool("Delete the table, too? (y/n)", "y") {
    err = tableBasics.DeleteTable()
} else {
```

```
    log.Println("Don't forget to delete the table when you're done or you might " +
        "incur charges on your account.")
    }
    if err == nil {
        log.Printf("Deleted table %v.\n", tableBasics.TableName)
    }

    log.Println(strings.Repeat("-", 88))
    log.Println("Thanks for watching!")
    log.Println(strings.Repeat("-", 88))
}
```

Define a Movie struct that is used in this example.

```
// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}
```

```
// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

Create a struct and methods that call DynamoDB actions.

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
// examples.
// It contains a DynamoDB service client that is used to act on the specified
// table.
type TableBasics struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// TableExists determines whether a DynamoDB table exists.
func (basics TableBasics) TableExists() (bool, error) {
    exists := true
    _, err := basics.DynamoDbClient.DescribeTable(
        context.TODO(), &dynamodb.DescribeTableInput{TableName:
            aws.String(basics.TableName),
    )
    if err != nil {
        var notFoundEx *types.ResourceNotFoundException
        if errors.As(err, &notFoundEx) {
            log.Printf("Table %v does not exist.\n", basics.TableName)
            err = nil
        } else {
            log.Printf("Couldn't determine existence of table %v. Here's why: %v\n",
                basics.TableName, err)
        }
        exists = false
    }
    return exists, err
}
```

```
// CreateMovieTable creates a DynamoDB table with a composite primary key defined
// as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable() (*types.TableDescription, error) {
    var tableDesc *types.TableDescription
    table, err := basics.DynamoDbClient.CreateTable(context.TODO(),
        &dynamodb.CreateTableInput{
            AttributeDefinitions: []types.AttributeDefinition{{
               AttributeName: aws.String("year"),
                AttributeType: types.ScalarAttributeTypeN,
            }, {
               AttributeName: aws.String("title"),
                AttributeType: types.ScalarAttributeTypeS,
            }},
            KeySchema: []types.KeySchemaElement{{
               AttributeName: aws.String("year"),
                KeyType: types.KeyTypeHash,
            }, {
               AttributeName: aws.String("title"),
                KeyType: types.KeyTypeRange,
            }},
            TableName: aws.String(basics.TableName),
            ProvisionedThroughput: &types.ProvisionedThroughput{
                ReadCapacityUnits: aws.Int64(10),
                WriteCapacityUnits: aws.Int64(10),
            },
        })
    if err != nil {
        log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
    } else {
        waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
        err = waiter.Wait(context.TODO(), &dynamodb.DescribeTableInput{
            TableName: aws.String(basics.TableName)}, 5*time.Minute)
        if err != nil {
            log.Printf("Wait for table exists failed. Here's why: %v\n", err)
        }
        tableDesc = table.TableDescription
    }
    return tableDesc, err
}
```

```
}
```

```
// ListTables lists the DynamoDB table names for the current account.
func (basics TableBasics) ListTables() ([]string, error) {
    var tableNames []string
    var output *dynamodb.ListTablesOutput
    var err error
    tablePaginator := dynamodb.NewListTablesPaginator(basics.DynamoDbClient,
    &dynamodb.ListTablesInput{})
    for tablePaginator.HasMorePages() {
        output, err = tablePaginator.NextPage(context.TODO())
        if err != nil {
            log.Printf("Couldn't list tables. Here's why: %v\n", err)
            break
        } else {
            tableNames = append(tableNames, output.TableNames...)
        }
    }
    return tableNames, err
}
```

```
// AddMovie adds a movie to the DynamoDB table.
func (basics TableBasics) AddMovie(movie Movie) error {
    item, err := attributevalue.MarshalMap(movie)
    if err != nil {
        panic(err)
    }
    _, err = basics.DynamoDbClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
        TableName: aws.String(basics.TableName), Item: item,
    })
    if err != nil {
        log.Printf("Couldn't add item to table. Here's why: %v\n", err)
    }
    return err
}
```

```
// UpdateMovie updates the rating and plot of a movie that already exists in the
```

```
// DynamoDB table. This function uses the `expression` package to build the
// update
// expression.
func (basics TableBasics) UpdateMovie(movie Movie)
(map[string]map[string]interface{}, error) {
var err error
var response *dynamodb.UpdateItemOutput
var attributeMap map[string]map[string]interface{}
update := expression.Set(expression.Name("info.rating"),
expression.Value(movie.Info["rating"]))
update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
expr, err := expression.NewBuilder().WithUpdate(update).Build()
if err != nil {
log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
} else {
response, err = basics.DynamoDbClient.UpdateItem(context.TODO(),
&dynamodb.UpdateItemInput{
TableName: aws.String(basics.TableName),
Key: movie.GetKey(),
ExpressionAttributeNames: expr.Names(),
ExpressionAttributeValues: expr.Values(),
UpdateExpression: expr.Update(),
ReturnValues: types.ReturnValueUpdatedNew,
})
if err != nil {
log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
} else {
err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
if err != nil {
log.Printf("Couldn't unmarshal update response. Here's why: %v\n", err)
}
}
}
return attributeMap, err
}
```

```
// AddMovieBatch adds a slice of movies to the DynamoDB table. The function sends
// batches of 25 movies to DynamoDB until all movies are added or it reaches the
// specified maximum.
func (basics TableBasics) AddMovieBatch(movies []Movie, maxMovies int) (int,
error) {
var err error
```

```
var item map[string]types.AttributeValue
written := 0
batchSize := 25 // DynamoDB allows a maximum batch size of 25 items.
start := 0
end := start + batchSize
for start < maxMovies && start < len(movies) {
    var writeReqs []types.WriteRequest
    if end > len(movies) {
        end = len(movies)
    }
    for _, movie := range movies[start:end] {
        item, err = attributevalue.MarshalMap(movie)
        if err != nil {
            log.Printf("Couldn't marshal movie %v for batch writing. Here's why: %v\n",
                      movie.Title, err)
        } else {
            writeReqs = append(
                writeReqs,
                types.WriteRequest{PutRequest: &types.PutRequest{Item: item}},
            )
        }
    }
    _, err = basics.DynamoDbClient.BatchWriteItem(context.TODO(),
&dynamodb.BatchWriteItemInput{
    RequestItems: map[string][]types.WriteRequest{basics.TableName: writeReqs}})
    if err != nil {
        log.Printf("Couldn't add a batch of movies to %v. Here's why: %v\n",
                  basics.TableName, err)
    } else {
        written += len(writeReqs)
    }
    start = end
    end += batchSize
}

return written, err
}

// GetMovie gets movie data from the DynamoDB table by using the primary
// composite key
// made of title and year.
func (basics TableBasics) GetMovie(title string, year int) (Movie, error) {
```

```
movie := Movie{Title: title, Year: year}
response, err := basics.DynamoDbClient.GetItem(context.TODO(),
&dynamodb.GetItemInput{
    Key: movie.GetKey(), TableName: aws.String(basics.TableName),
})
if err != nil {
    log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
} else {
    err = attributevalue.UnmarshalMap(response.Item, &movie)
    if err != nil {
        log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
    }
}
return movie, err
}

// Query gets all movies in the DynamoDB table that were released in the
// specified year.
// The function uses the `expression` package to build the key condition
// expression
// that is used in the query.
func (basics TableBasics) Query(releaseYear int) ([]Movie, error) {
    var err error
    var response *dynamodb.QueryOutput
    var movies []Movie
    keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
    expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
    if err != nil {
        log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
    } else {
        queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
&dynamodb.QueryInput{
    TableName:                aws.String(basics.TableName),
    ExpressionAttributeNames: expr.Names(),
    ExpressionAttributeValues: expr.Values(),
    KeyConditionExpression:   expr.KeyCondition(),
})
        for queryPaginator.HasMorePages() {
            response, err = queryPaginator.NextPage(context.TODO())
            if err != nil {
                log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
releaseYear, err)
            }
        }
    }
}
```

```
        break
    } else {
        var moviePage []Movie
        err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
        if err != nil {
            log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
            break
        } else {
            movies = append(movies, moviePage...)
        }
    }
}
}

return movies, err
}

// Scan gets all movies in the DynamoDB table that were released in a range of
// years
// and projects them to return a reduced set of fields.
// The function uses the `expression` package to build the filter and projection
// expressions.
func (basics TableBasics) Scan(startYear int, endYear int) ([]Movie, error) {
    var movies []Movie
    var err error
    var response *dynamodb.ScanOutput
    filtEx := expression.Name("year").Between(expression.Value(startYear),
        expression.Value(endYear))
    projEx := expression.NamesList(
        expression.Name("year"), expression.Name("title"),
        expression.Name("info.rating"))
    expr, err :=
        expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
    if err != nil {
        log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
    } else {
        scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
            &dynamodb.ScanInput{
                TableName: aws.String(basics.TableName),
                ExpressionAttributeNames: expr.Names(),
                ExpressionAttributeValues: expr.Values(),
                FilterExpression: expr.Filter(),
                ProjectionExpression: expr.Projection(),
            })
        for {
            page, err := scanPaginator.NextPage()
            if err != nil {
                log.Printf("Error getting next page of scan results: %v\n", err)
                break
            }
            movies = append(movies, page.Movies...)
        }
    }
}
```

```
)  
for scanPaginator.HasMorePages() {  
    response, err = scanPaginator.NextPage(context.TODO())  
    if err != nil {  
        log.Printf("Couldn't scan for movies released between %v and %v. Here's why:  
%v\n",  
        startYear, endYear, err)  
        break  
    } else {  
        var moviePage []Movie  
        err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)  
        if err != nil {  
            log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)  
            break  
        } else {  
            movies = append(movies, moviePage...)  
        }  
    }  
}  
}  
return movies, err  
}
```

```
// DeleteMovie removes a movie from the DynamoDB table.  
func (basics TableBasics) DeleteMovie(movie Movie) error {  
    _, err := basics.DynamoDbClient.DeleteItem(context.TODO(),  
&dynamodb.DeleteItemInput{  
    TableName: aws.String(basics.TableName), Key: movie.GetKey(),  
})  
if err != nil {  
    log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,  
err)  
}  
return err  
}
```

```
// DeleteTable deletes the DynamoDB table and all of its data.  
func (basics TableBasics) DeleteTable() error {  
    _, err := basics.DynamoDbClient.DeleteTable(context.TODO(),  
&dynamodb.DeleteTableInput{
```

```
    TableName: aws.String(basics.TableName)})  
if err != nil {  
    log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)  
}  
return err  
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.

- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a DynamoDB table.

```
// Create a table with a Sort key.  
public static void createTable(DynamoDbClient ddb, String tableName) {  
    DynamoDbWaiter dbWaiter = ddb.waiter();  
    ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();
```

```
// Define attributes.
attributeDefinitions.add(AttributeDefinition.builder()
    .attributeName("year")
    .attributeType("N")
    .build());

attributeDefinitions.add(AttributeDefinition.builder()
    .attributeName("title")
    .attributeType("S")
    .build());

ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
KeySchemaElement key = KeySchemaElement.builder()
    .attributeName("year")
    .keyType(KeyType.HASH)
    .build();

KeySchemaElement key2 = KeySchemaElement.builder()
    .attributeName("title")
    .keyType(KeyType.RANGE)
    .build();

// Add KeySchemaElement objects to the list.
tableKey.add(key);
tableKey.add(key2);

CreateTableRequest request = CreateTableRequest.builder()
    .keySchema(tableKey)
    .provisionedThroughput(ProvisionedThroughput.builder()
        .readCapacityUnits(10L)
        .writeCapacityUnits(10L)
        .build())
    .attributeDefinitions(attributeDefinitions)
    .tableName(tableName)
    .build();

try {
    CreateTableResponse response = ddb.createTable(request);
    DescribeTableRequest tableRequest = DescribeTableRequest.builder()
        .tableName(tableName)
        .build();

    // Wait until the Amazon DynamoDB table is created.
```

```
        WaiterResponse<DescribeTableResponse> waiterResponse =
    dbWaiter.waitUntilTableExists(tableRequest);
    waiterResponse.matched().response().ifPresent(System.out::println);
    String newTable = response.tableDescription().tableName();
    System.out.println("The " + newTable + " was successfully created.");

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

Create a helper function to download and extract the sample JSON file.

```
// Load data into the table.
public static void loadData(DynamoDbClient ddb, String tableName, String
fileName) throws IOException {
    DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();

    DynamoDbTable<Movies> mappedTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
    JsonParser parser = new JsonFactory().createParser(new File(fileName));
    com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
    Iterator<JsonNode> iter = rootNode.iterator();
    ObjectNode currentNode;
    int t = 0;
    while (iter.hasNext()) {
        // Only add 200 Movies to the table.
        if (t == 200)
            break;
        currentNode = (ObjectNode) iter.next();

        int year = currentNode.path("year").asInt();
        String title = currentNode.path("title").asText();
        String info = currentNode.path("info").toString();

        Movies movies = new Movies();
        movies.setYear(year);
        movies.setTitle(title);
```

```
        movies.setInfo(info);

        // Put the data into the Amazon DynamoDB Movie table.
        mappedTable.putItem(movies);
        t++;
    }
}
```

Get an item from a table.

```
public static void getItem(DynamoDbClient ddb) {

    HashMap<String, AttributeValue> keyToGet = new HashMap<>();
    keyToGet.put("year", AttributeValue.builder()
        .n("1933")
        .build());

    keyToGet.put("title", AttributeValue.builder()
        .s("King Kong")
        .build());

    GetItemRequest request = GetItemRequest.builder()
        .key(keyToGet)
        .tableName("Movies")
        .build();

    try {
        Map<String, AttributeValue> returnedItem =
ddb.getItem(request).item();

        if (returnedItem != null) {
            Set<String> keys = returnedItem.keySet();
            System.out.println("Amazon DynamoDB table attributes: \n");

            for (String key1 : keys) {
                System.out.format("%s: %s\n", key1,
returnedItem.get(key1).toString());
            }
        } else {
            System.out.format("No item found with the key %s!\n", "year");
        }
    }
}
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

Full example.

```
/**  
 * Before running this Java V2 code example, set up your development  
 * environment, including your credentials.  
 *  
 * For more information, see the following documentation topic:  
 *  
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html  
 *  
 * This Java example performs these tasks:  
 *  
 * 1. Creates the Amazon DynamoDB Movie table with partition and sort key.  
 * 2. Puts data into the Amazon DynamoDB table from a JSON document using the  
 * Enhanced client.  
 * 3. Gets data from the Movie table.  
 * 4. Adds a new item.  
 * 5. Updates an item.  
 * 6. Uses a Scan to query items using the Enhanced client.  
 * 7. Queries all items where the year is 2013 using the Enhanced Client.  
 * 8. Deletes the table.  
 */  
  
public class Scenario {  
    public static final String DASHES = new String(new char[80]).replace("\0",  
"-");  
  
    public static void main(String[] args) throws IOException {  
        final String usage = """  
  
            Usage:  
                <fileName>  
  
            Where:  
"/>
```

```
        fileName - The path to the moviedata.json file that you can
download from the Amazon DynamoDB Developer Guide.

""";

if (args.length != 1) {
    System.out.println(usage);
    System.exit(1);
}

String tableName = "Movies";
String fileName = args[0];
Region region = Region.US_EAST_1;
DynamoDbClient ddb = DynamoDbClient.builder()
    .region(region)
    .build();

System.out.println(DASHES);
System.out.println("Welcome to the Amazon DynamoDB example scenario.");
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println(
    "1. Creating an Amazon DynamoDB table named Movies with a key
named year and a sort key named title.");
createTable(ddb, tableName);
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("2. Loading data into the Amazon DynamoDB table.");
loadData(ddb, tableName, fileName);
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("3. Getting data from the Movie table.");
getItem(ddb);
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("4. Putting a record into the Amazon DynamoDB
table.");
putRecord(ddb);
System.out.println(DASHES);

System.out.println(DASHES);
```

```
System.out.println("5. Updating a record.");
updateTableItem(ddb, tableName);
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("6. Scanning the Amazon DynamoDB table.");
scanMovies(ddb, tableName);
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("7. Querying the Movies released in 2013.");
queryTable(ddb);
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("8. Deleting the Amazon DynamoDB table.");
deleteDynamoDBTable(ddb, tableName);
System.out.println(DASHES);

ddb.close();
}

// Create a table with a Sort key.
public static void createTable(DynamoDbClient ddb, String tableName) {
    DynamoDbWaiter dbWaiter = ddb.waiter();
    ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

    // Define attributes.
    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("year")
        .attributeType("N")
        .build());

    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("title")
        .attributeType("S")
        .build());

    ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
    KeySchemaElement key = KeySchemaElement.builder()
        .attributeName("year")
        .keyType(KeyType.HASH)
        .build();
}
```

```
KeySchemaElement key2 = KeySchemaElement.builder()
    .attributeName("title")
    .keyType(KeyType.RANGE)
    .build();

// Add KeySchemaElement objects to the list.
tableKey.add(key);
tableKey.add(key2);

CreateTableRequest request = CreateTableRequest.builder()
    .keySchema(tableKey)
    .provisionedThroughput(ProvisionedThroughput.builder()
        .readCapacityUnits(10L)
        .writeCapacityUnits(10L)
        .build())
    .attributeDefinitions(attributeDefinitions)
    .tableName(tableName)
    .build();

try {
    CreateTableResponse response = ddb.createTable(request);
    DescribeTableRequest tableRequest = DescribeTableRequest.builder()
        .tableName(tableName)
        .build();

    // Wait until the Amazon DynamoDB table is created.
    WaiterResponse<DescribeTableResponse> waiterResponse =
    dbWaiter.waitUntilTableExists(tableRequest);
    waiterResponse.matched().response().ifPresent(System.out::println);
    String newTable = response.tableDescription().tableName();
    System.out.println("The " + newTable + " was successfully created.");

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}

// Query the table.
public static void queryTable(DynamoDbClient ddb) {
    try {
        DynamoDbEnhancedClient enhancedClient =
        DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
```

```
        .build();

        DynamoDbTable<Movies> custTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
        QueryConditional queryConditional = QueryConditional
            .keyEqualTo(Key.builder()
                .partitionValue(2013)
            .build());

        // Get items in the table and write out the ID value.
        Iterator<Movies> results =
custTable.query(queryConditional).items().iterator();
        String result = "";

        while (results.hasNext()) {
            Movies rec = results.next();
            System.out.println("The title of the movie is " +
rec.getTitle());
            System.out.println("The movie information is " + rec.getInfo());
        }

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

// Scan the table.
public static void scanMovies(DynamoDbClient ddb, String tableName) {
    System.out.println("***** Scanning all movies.\n");
    try {
        DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
            .build();

        DynamoDbTable<Movies> custTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
        Iterator<Movies> results = custTable.scan().items().iterator();
        while (results.hasNext()) {
            Movies rec = results.next();
            System.out.println("The movie title is " + rec.getTitle());
            System.out.println("The movie year is " + rec.getYear());
        }
    }
}
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String tableName, String
fileName) throws IOException {
        DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
            .build();

        DynamoDbTable<Movies> mappedTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
        ObjectNode currentNode;
        int t = 0;
        while (iter.hasNext()) {
            // Only add 200 Movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();

            Movies movies = new Movies();
            movies.setYear(year);
            movies.setTitle(title);
            movies.setInfo(info);

            // Put the data into the Amazon DynamoDB Movie table.
            mappedTable.putItem(movies);
            t++;
        }
    }

    // Update the record to include show only directors.
```

```
public static void updateTableItem(DynamoDbClient ddb, String tableName) {  
    HashMap<String, AttributeValue> itemKey = new HashMap<>();  
    itemKey.put("year", AttributeValue.builder().n("1933").build());  
    itemKey.put("title", AttributeValue.builder().s("King Kong").build());  
  
    HashMap<String,AttributeValueUpdate> updatedValues = new HashMap<>();  
    updatedValues.put("info", AttributeValueUpdate.builder()  
        .value(AttributeValue.builder().s("{\"directors\":[\"Merian C.  
Cooper\",\"Ernest B. Schoedsack\"]}")  
        .build())  
        .action(AttributeAction.PUT)  
        .build());  
  
    UpdateItemRequest request = UpdateItemRequest.builder()  
        .tableName(tableName)  
        .key(itemKey)  
        .attributeUpdates(updatedValues)  
        .build();  
  
    try {  
        ddb.updateItem(request);  
    } catch (ResourceNotFoundException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    }  
  
    System.out.println("Item was updated!");  
}  
  
public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)  
{  
    DeleteTableRequest request = DeleteTableRequest.builder()  
        .tableName(tableName)  
        .build();  
  
    try {  
        ddb.deleteTable(request);  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
}
```

```
        }
        System.out.println(tableName + " was successfully deleted!");
    }

    public static void putRecord(DynamoDbClient ddb) {
        try {
            DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

            DynamoDbTable<Movies> table = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));

            // Populate the Table.
            Movies record = new Movies();
            record.setYear(2020);
            record.setTitle("My Movie2");
            record.setInfo("no info");
            table.putItem(record);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Added a new movie to the table.");
    }

    public static void getItem(DynamoDbClient ddb) {

        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put("year", AttributeValue.builder()
            .n("1933")
            .build());

        keyToGet.put("title", AttributeValue.builder()
            .s("King Kong")
            .build());

        GetItemRequest request = GetItemRequest.builder()
            .key(keyToGet)
            .tableName("Movies")
            .build();
    }
}
```

```
try {
    Map<String, AttributeValue> returnedItem =
ddb.getItem(request).item();

    if (returnedItem != null) {
        Set<String> keys = returnedItem.keySet();
        System.out.println("Amazon DynamoDB table attributes: \n");

        for (String key1 : keys) {
            System.out.format("%s: %s\n", key1,
returnedItem.get(key1).toString());
        }
    } else {
        System.out.format("No item found with the key %s!\n", "year");
    }

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.

- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { readFileSync } from "fs";
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";

/**
 * This module is a convenience library. It abstracts Amazon DynamoDB's data type
 * descriptors (such as S, N, B, and BOOL) by marshalling JavaScript objects into
 * AttributeValue shapes.
 */
import {
  BatchWriteCommand,
  DeleteCommand,
  DynamoDBDocumentClient,
  GetCommand,
  PutCommand,
  UpdateCommand,
  paginateQuery,
  paginateScan,
} from "@aws-sdk/lib-dynamodb";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";
```

```
const dirname = dirnameFromMetaUrl(import.meta.url);
const tableName = getUniqueName("Movies");
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);

export const main = async () => {
  /**
   * Create a table.
   */

  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "year",
        // 'N' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "N",
      },
      { AttributeName: "title", AttributeType: "S" },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html
    KeySchema: [
      // The way your data is accessed determines how you structure your keys.
      // The movies table will be queried for movies by year. It makes sense
      // to make year our partition (HASH) key.
      { AttributeName: "year", KeyType: "HASH" },
      { AttributeName: "title", KeyType: "RANGE" },
    ],
  });
}
```

```
log("Creating a table.");
const createTableResponse = await client.send(createTableCommand);
log(`Table created: ${JSON.stringify(createTableResponse.TableDescription)}`);

// This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
// You can't write to a table before it's active.
log("Waiting for the table to be active.");
await waitUntilTableExists({ client }, { TableName: tableName });
log("Table active.");

/**
 * Add a movie to the table.
 */

log("Adding a single movie to the table.");
// PutCommand is the first example usage of 'lib-dynamodb'.
const putCommand = new PutCommand({
    TableName: tableName,
    Item: {
        // In 'client-dynamodb', the AttributeValue would be required (`year: { N: 1981 }`)
        // 'lib-dynamodb' simplifies the usage ( `year: 1981` )
        year: 1981,
        // The preceding KeySchema defines 'title' as our sort (RANGE) key, so
        'title'
        // is required.
        title: "The Evil Dead",
        // Every other attribute is optional.
        info: {
            genres: ["Horror"],
        },
    },
});
await docClient.send(putCommand);
log("The movie was added.");

/**
 * Get a movie from the table.
 */

log("Getting a single movie from the table.");
const getCommand = new GetCommand({
    TableName: tableName,
    // Requires the complete primary key. For the movies table, the primary key
```

```
// is only the id (partition key).
Key: {
    year: 1981,
    title: "The Evil Dead",
},
// Set this to make sure that recent writes are reflected.
// For more information, see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html.
ConsistentRead: true,
});

const getResponse = await docClient.send(getCommand);
log(`Got the movie: ${JSON.stringify(getResponse.Item)}`);

/**
 * Update a movie in the table.
*/
log("Updating a single movie in the table.");
const updateCommand = new UpdateCommand({
    TableName: tableName,
    Key: { year: 1981, title: "The Evil Dead" },
    // This update expression appends "Comedy" to the list of genres.
    // For more information on update expressions, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.UpdateExpressions.html
    UpdateExpression: "set #i.#g = list_append(#i.#g, :vals)",
    ExpressionAttributeNames: { "#i": "info", "#g": "genres" },
    ExpressionAttributeValues: {
        ":vals": ["Comedy"],
    },
    ReturnValue: "ALL_NEW",
});
const updateResponse = await docClient.send(updateCommand);
log(`Movie updated: ${JSON.stringify(updateResponse.Attributes)}`);

/**
 * Delete a movie from the table.
*/
log("Deleting a single movie from the table.");
const deleteCommand = new DeleteCommand({
    TableName: tableName,
    Key: { year: 1981, title: "The Evil Dead" },
});
```

```
await client.send(deleteCommand);
log("Movie deleted.");

/**
 * Upload a batch of movies.
 */

log("Adding movies from local JSON file.");
const file = readFileSync(
  `${dirname}../../../../resources/sample_files/movies.json`,
);
const movies = JSON.parse(file.toString());
// chunkArray is a local convenience function. It takes an array and returns
// a generator function. The generator function yields every N items.
const movieChunks = chunkArray(movies, 25);
// For every chunk of 25 movies, make one BatchWrite request.
for (const chunk of movieChunks) {
  const putRequests = chunk.map((movie) => ({
    PutRequest: {
      Item: movie,
    },
  }));
}

const command = new BatchWriteCommand({
  RequestItems: {
    [tableName]: putRequests,
  },
});

await docClient.send(command);
}
log("Movies added.");

/**
 * Query for movies by year.
 */

log("Querying for all movies from 1981.");
const paginatedQuery = paginateQuery(
  { client: docClient },
  {
    TableName: tableName,
    //For more information about query expressions, see
  }
);
```

```
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Query.html#Query.KeyConditionExpressions
    KeyConditionExpression: "#y = :y",
    // 'year' is a reserved word in DynamoDB. Indicate that it's an attribute
    // name by using an expression attribute name.
    ExpressionAttributeNames: { "#y": "year" },
    ExpressionAttributeValues: { ":y": 1981 },
    ConsistentRead: true,
},
);
/***
 * @type { Record<string, any>[] };
 */
const movies1981 = [];
for await (const page of paginatedQuery) {
    movies1981.push(...page.Items);
}
log(`Movies: ${movies1981.map((m) => m.title).join(", ")}`);

/**
 * Scan the table for movies between 1980 and 1990.
 */

log(`Scan for movies released between 1980 and 1990`);
// A 'Scan' operation always reads every item in the table. If your design
requires
// the use of 'Scan', consider indexing your table or changing your design.
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-
scan.html
const paginatedScan = paginateScan(
    { client: docClient },
{
    TableName: tableName,
    // Scan uses a filter expression instead of a key condition expression.
Scan will
    // read the entire table and then apply the filter.
    FilterExpression: "#y between :y1 and :y2",
    ExpressionAttributeNames: { "#y": "year" },
    ExpressionAttributeValues: { ":y1": 1980, ":y2": 1990 },
    ConsistentRead: true,
},
);
/***
 * @type { Record<string, any>[] };

```

```
 */
const movies1980to1990 = [];
for await (const page of paginatedScan) {
    movies1980to1990.push(...page.Items);
}
log(
    `Movies: ${movies1980to1990
        .map((m) => `${m.title} (${m.year})`)
        .join(", ")}`);
);

/**
 * Delete the table.
 */

const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
log(`Deleting table ${tableName}.`);
await client.send(deleteTableCommand);
log("Table deleted.");
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)
 - [DeleteItem](#)
 - [DeleteTable](#)
 - [DescribeTable](#)
 - [GetItem](#)
 - [PutItem](#)
 - [Query](#)
 - [Scan](#)
 - [UpdateItem](#)

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a DynamoDB table.

```
suspend fun createScenarioTable(tableNameVal: String, key: String) {  
    val attDef = AttributeDefinition {  
        attributeName = key  
        attributeType = ScalarAttributeType.N  
    }  
  
    val attDef1 = AttributeDefinition {  
        attributeName = "title"  
        attributeType = ScalarAttributeType.S  
    }  
  
    val keySchemaVal = KeySchemaElement {  
        attributeName = key  
        keyType = KeyType.Hash  
    }  
  
    val keySchemaVal1 = KeySchemaElement {  
        attributeName = "title"  
        keyType = KeyType.Range  
    }  
  
    val provisionedVal = ProvisionedThroughput {  
        readCapacityUnits = 10  
        writeCapacityUnits = 10  
    }  
  
    val request = CreateTableRequest {  
        attributeDefinitions = listOf(attDef, attDef1)  
        keySchema = listOf(keySchemaVal, keySchemaVal1)  
        provisionedThroughput = provisionedVal  
        tableName = tableNameVal  
    }  
}
```

```
}

DynamoDbClient { region = "us-east-1" }.use { ddb ->

    val response = ddb.createTable(request)
    ddb.waitUntilTableExists { // suspend call
        tableName = tableNameVal
    }
    println("The table was successfully created
${response.tableDescription?.tableArn}")
}
}
```

Create a helper function to download and extract the sample JSON file.

```
// Load data into the table.
suspend fun loadData(tableName: String, fileName: String) {
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode

    var t = 0
    while (iter.hasNext()) {
        if (t == 50) {
            break
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()
        putMovie(tableName, year, title, info)
        t++
    }
}

suspend fun putMovie(
    tableNameVal: String,
    year: Int,
    title: String,
    info: String
```

```
) {  
    val itemValues = mutableMapOf<String, AttributeValue>()  
    val strVal = year.toString()  
    // Add all content to the table.  
    itemValues["year"] = AttributeValue.N(strVal)  
    itemValues["title"] = AttributeValue.S(title)  
    itemValues["info"] = AttributeValue.S(info)  
  
    val request = PutItemRequest {  
        tableName = tableNameVal  
        item = itemValues  
    }  
  
    DynamoDbClient { region = "us-east-1" }.use { ddb ->  
        ddb.putItem(request)  
        println("Added $title to the Movie table.")  
    }  
}
```

Get an item from a table.

```
suspend fun getMovie(tableNameVal: String, keyName: String, keyVal: String) {  
    val keyToGet = mutableMapOf<String, AttributeValue>()  
    keyToGet[keyName] = AttributeValue.N(keyVal)  
    keyToGet["title"] = AttributeValue.S("King Kong")  
  
    val request = GetItemRequest {  
        key = keyToGet  
        tableName = tableNameVal  
    }  
  
    DynamoDbClient { region = "us-east-1" }.use { ddb ->  
        val returnedItem = ddb.getItem(request)  
        val numbersMap = returnedItem.item  
        numbersMap?.forEach { key1 ->  
            println(key1.key)  
            println(key1.value)  
        }  
    }  
}
```

Full example.

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
        <fileName>

        Where:
        fileName - The path to the moviedata.json you can download from the
        Amazon DynamoDB Developer Guide.

    """

    if (args.size != 1) {
        println(usage)
        exitProcess(1)
    }

    // Get the moviedata.json from the Amazon DynamoDB Developer Guide.
    val tableName = "Movies"
    val fileName = args[0]
    val partitionAlias = "#a"

    println("Creating an Amazon DynamoDB table named Movies with a key named id
and a sort key named title.")
    createScenarioTable(tableName, "year")
    loadData(tableName, fileName)
    getMovie(tableName, "year", "1933")
    scanMovies(tableName)
    val count = queryMovieTable(tableName, "year", partitionAlias)
    println("There are $count Movies released in 2013.")
    deleteIssuesTable(tableName)
}

suspend fun createScenarioTable(tableNameVal: String, key: String) {
    val attDef = AttributeDefinition {
        attributeName = key
        attributeType = ScalarAttributeType.N
    }

    val attDef1 = AttributeDefinition {
        attributeName = "title"
        attributeType = ScalarAttributeType.S
    }
```

```
val keySchemaVal = KeySchemaElement {  
    attributeName = key  
    keyType = KeyType.Hash  
}  
  
val keySchemaVal1 = KeySchemaElement {  
    attributeName = "title"  
    keyType = KeyType.Range  
}  
  
val provisionedVal = ProvisionedThroughput {  
    readCapacityUnits = 10  
    writeCapacityUnits = 10  
}  
  
val request = CreateTableRequest {  
    attributeDefinitions = listOf(attDef, attDef1)  
    keySchema = listOf(keySchemaVal, keySchemaVal1)  
    provisionedThroughput = provisionedVal  
    tableName = tableNameVal  
}  
  
DynamoDbClient { region = "us-east-1" }.use { ddb ->  
  
    val response = ddb.createTable(request)  
    ddb.waitUntilTableExists { // suspend call  
        tableName = tableNameVal  
    }  
    println("The table was successfully created  
${response.tableDescription?.tableArn}")  
}  
}  
  
// Load data into the table.  
suspend fun loadData(tableName: String, fileName: String) {  
    val parser = JsonFactory().createParser(File(fileName))  
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)  
    val iter: Iterator<JsonNode> = rootNode.iterator()  
    var currentNode: ObjectNode  
  
    var t = 0  
    while (iter.hasNext()) {  
        if (t == 50) {  
            break  
        }  
    }  
}
```

```
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()
        putMovie(tableName, year, title, info)
        t++
    }
}

suspend fun putMovie(
    tableNameVal: String,
    year: Int,
    title: String,
    info: String
) {
    val itemValues = mutableMapOf<String, AttributeValue>()
    val strVal = year.toString()
    // Add all content to the table.
    itemValues["year"] = AttributeValue.N(strVal)
    itemValues["title"] = AttributeValue.S(title)
    itemValues["info"] = AttributeValue.S(info)

    val request = PutItemRequest {
        tableName = tableNameVal
        item = itemValues
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println("Added $title to the Movie table.")
    }
}

suspend fun getMovie(tableNameVal: String, keyName: String, keyVal: String) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.N(keyVal)
    keyToGet["title"] = AttributeValue.S("King Kong")

    val request = GetItemRequest {
        key = keyToGet
        tableName = tableNameVal
    }
}
```

```
DynamoDbClient { region = "us-east-1" }.use { ddb ->
    val returnedItem = ddb.getItem(request)
    val numbersMap = returnedItem.item
    numbersMap?.forEach { key1 ->
        println(key1.key)
        println(key1.value)
    }
}

suspend fun deleteIssuesTable(tableNameVal: String) {
    val request = DeleteTableRequest {
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}

suspend fun queryMovieTable(
    tableNameVal: String,
    partitionKeyName: String,
    partitionAlias: String
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = "year"

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.N("2013")

    val request = QueryRequest {
        tableName = tableNameVal
        keyConditionExpression = "$partitionAlias = :$partitionKeyName"
        expressionAttributeNames = attrNameAlias
        this.expressionAttributeValues = attrValues
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        val response = ddb.query(request)
        return response.count
    }
}
```

```
        }
```

```
}
```

```
suspend fun scanMovies(tableNameVal: String) {
```

```
    val request = ScanRequest {
```

```
        tableName = tableNameVal
```

```
    }
```

```
    DynamoDbClient { region = "us-east-1" }.use { ddb ->
```

```
        val response = ddb.scan(request)
```

```
        response.items?.forEach { item ->
```

```
            item.keys.forEach { key ->
```

```
                println("The key name is $key\n")
```

```
                println("The value is ${item[key]}")
```

```
            }
```

```
        }
```

```
}
```

```
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)
 - [DeleteItem](#)
 - [DeleteTable](#)
 - [DescribeTable](#)
 - [GetItem](#)
 - [PutItem](#)
 - [Query](#)
 - [Scan](#)
 - [UpdateItem](#)

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
namespace DynamoDb\Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;
use DynamoDb\DynamoDBService;

use function AwsUtilities\loadMovieData;
use function AwsUtilities\testable_readline;

class GettingStartedWithDynamoDB
{
    public function run()
    {
        echo("\n");
        echo("-----\n");
        print("Welcome to the Amazon DynamoDB getting started demo using PHP!\n");
        echo("-----\n");

        $uuid = uniqid();
        $service = new DynamoDBService();

        $tableName = "ddb_demo_table_{$uuid}";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );
    }
}
```

```
echo "Waiting for table...";  
$service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>  
$tableName]);  
echo "table $tableName found!\n";  
  
echo "What's the name of the last movie you watched?\n";  
while (empty($movieName)) {  
    $movieName = testable_readline("Movie name: ");  
}  
echo "And what year was it released?\n";  
$movieYear = "year";  
while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {  
    $movieYear = testable_readline("Year released: ");  
}  
  
$service->putItem([  
    'Item' => [  
        'year' => [  
            'N' => "$movieYear",  
        ],  
        'title' => [  
            'S' => $movieName,  
        ],  
    ],  
    'TableName' => $tableName,  
]);  
  
echo "How would you rate the movie from 1-10?\n";  
$rating = 0;  
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1  
|| $rating > 10) {  
    $rating = testable_readline("Rating (1-10): ");  
}  
echo "What was the movie about?\n";  
while (empty($plot)) {  
    $plot = testable_readline("Plot summary: ");  
}  
$key = [  
    'Item' => [  
        'title' => [  
            'S' => $movieName,  
        ],  
        'year' => [  
            'N' => $movieYear,
```

```
        ],
    ]
];
$attributes = ["rating" =>
[
    'AttributeName' => 'rating',
    'AttributeType' => 'N',
    'Value' => $rating,
],
'plot' => [
    'AttributeName' => 'plot',
    'AttributeType' => 'S',
    'Value' => $plot,
]
];
$service->updateItemAttributesByKey($tableName, $key, $attributes);
echo "Movie added and updated.";

$batch = json_decode(loadMovieData());

$service->writeBatch($tableName, $batch);

$movie = $service->getItemByKey($tableName, $key);
echo "\nThe movie {$movie['Item']['title'][['S']]}" was released in
{$movie['Item']['year'][['N']]}.";
echo "What rating would you like to give {$movie['Item']['title'][['S']]}"?
\n";
$rating = 0;
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
    $rating = testable_readline("Rating (1-10): ");
}
$service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
$rating);

$movie = $service->getItemByKey($tableName, $key);
echo "Ok, you have rated {$movie['Item']['title'][['S']]}" as a
{$movie['Item']['rating'][['N']]}\n";

$service->deleteItemByKey($tableName, $key);
echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";
```

```
echo "That's okay though. The book was better. Now, for something
lighter, in what year were you born?\n";
$birthYear = "not a number";
while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
    $birthYear = testable_readline("Birth year: ");
}
$birthKey = [
    'Key' => [
        'year' => [
            'N' => "$birthYear",
        ],
    ],
];
$result = $service->query($tableName, $birthKey);
$marshal = new Marshaler();
echo "Here are the movies in our collection released the year you were
born:\n";
$oops = "Oops! There were no movies released in that year (that we know
of).\n";
$display = "";
foreach ($result['Items'] as $movie) {
    $movie = $marshal->unmarshalItem($movie);
    $display .= $movie['title'] . "\n";
}
echo ($display) ?: $oops;

$yearsKey = [
    'Key' => [
        'year' => [
            'N' => [
                'minRange' => 1990,
                'maxRange' => 1999,
            ],
        ],
    ],
];
$filter = "year between 1990 and 1999";
echo "\nHere's a list of all the movies released in the 90s:\n";
$result = $service->scan($tableName, $yearsKey, $filter);
foreach ($result['Items'] as $movie) {
    $movie = $marshal->unmarshalItem($movie);
    echo $movie['title'] . "\n";
}
```

```
        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}
```

- For API details, see the following topics in *AWS SDK for PHP API Reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)
 - [DeleteItem](#)
 - [DeleteTable](#)
 - [DescribeTable](#)
 - [GetItem](#)
 - [PutItem](#)
 - [Query](#)
 - [Scan](#)
 - [UpdateItem](#)

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a class that encapsulates a DynamoDB table.

```
from decimal import Decimal
from io import BytesIO
import json
import logging
import os
from pprint import pprint
import requests
```

```
from zipfile import ZipFile
import boto3
from boto3.dynamodb.conditions import Key
from botocore.exceptions import ClientError
from question import Question

logger = logging.getLogger(__name__)

class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data."""

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None

    def exists(self, table_name):
        """
        Determines whether a table exists. As a side effect, stores the table in
        a member variable.

        :param table_name: The name of the table to check.
        :return: True when the table exists; otherwise, False.
        """
        try:
            table = self.dyn_resource.Table(table_name)
            table.load()
            exists = True
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                exists = False
            else:
                logger.error(
                    "Couldn't check for existence of %s. Here's why: %s: %s",
                    table_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise


```

```
        else:
            self.table = table
        return exists

    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"}, # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"}, # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                ProvisionedThroughput={
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10,
                },
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table
```

```
def list_tables(self):
    """
    Lists the Amazon DynamoDB tables for the current account.

    :return: The list of tables.
    """
    try:
        tables = []
        for table in self.dyn_resource.tables.all():
            print(table.name)
            tables.append(table)
    except ClientError as err:
        logger.error(
            "Couldn't list tables. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return tables

def write_batch(self, movies):
    """
    Fills an Amazon DynamoDB table with the specified data, using the Boto3
    Table.batch_writer() function to put the items in the table.
    Inside the context manager, Table.batch_writer builds a list of
    requests. On exiting the context manager, Table.batch_writer starts
    sending
    batches of write requests to Amazon DynamoDB and automatically
    handles chunking, buffering, and retrying.

    :param movies: The data to put in the table. Each item must contain at
    least
                    the keys required by the schema that was specified when
    the
                    table was created.
    """
    try:
        with self.table.batch_writer() as writer:
            for movie in movies:
                writer.put_item(Item=movie)
    except ClientError as err:
        logger.error(
```

```
        "Couldn't load data into table %s. Here's why: %s: %s",
        self.table.name,
        err.response["Error"]["Code"],
        err.response["Error"]["Message"],
    )
    raise

def add_movie(self, title, year, plot, rating):
    """
    Adds a movie to the table.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :param plot: The plot summary of the movie.
    :param rating: The quality rating of the movie.
    """
    try:
        self.table.put_item(
            Item={
                "year": year,
                "title": title,
                "info": {"plot": plot, "rating": Decimal(str(rating))},
            }
        )
    except ClientError as err:
        logger.error(
            "Couldn't add movie %s to table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise

def get_movie(self, title, year):
    """
    Gets movie data from the table for a specific movie.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :return: The data about the requested movie.
    """

```

```
try:
    response = self.table.get_item(Key={"year": year, "title": title})
except ClientError as err:
    logger.error(
        "Couldn't get movie %s from table %s. Here's why: %s: %s",
        title,
        self.table.name,
        err.response["Error"]["Code"],
        err.response["Error"]["Message"],
    )
    raise
else:
    return response["Item"]

def update_movie(self, title, year, rating, plot):
    """
    Updates rating and plot data for a movie in the table.

    :param title: The title of the movie to update.
    :param year: The release year of the movie to update.
    :param rating: The updated rating to give the movie.
    :param plot: The updated plot summary to give the movie.
    :return: The fields that were updated, with their new values.
    """
    try:
        response = self.table.update_item(
            Key={"year": year, "title": title},
            UpdateExpression="set info.rating=:r, info.plot=:p",
            ExpressionAttributeValues={":r": Decimal(str(rating)), ":p": plot},
            ReturnValues="UPDATED_NEW",
        )
    except ClientError as err:
        logger.error(
            "Couldn't update movie %s in table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Attributes"]
```

```
def query_movies(self, year):
    """
    Queries for movies that were released in the specified year.

    :param year: The year to query.
    :return: The list of movies that were released in the specified year.
    """

    try:
        response =
self.table.query(KeyConditionExpression=Key("year").eq(year))
    except ClientError as err:
        logger.error(
            "Couldn't query for movies released in %s. Here's why: %s: %s",
            year,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Items"]

def scan_movies(self, year_range):
    """
    Scans for movies that were released in a range of years.
    Uses a projection expression to return a subset of data for each movie.

    :param year_range: The range of years to retrieve.
    :return: The list of movies released in the specified years.
    """

    movies = []
    scan_kwargs = {
        "FilterExpression": Key("year").between(
            year_range["first"], year_range["second"]
        ),
        "ProjectionExpression": "#yr, title, info.rating",
        "ExpressionAttributeNames": {"#yr": "year"},
    }
    try:
        done = False
        start_key = None
        while not done:
```

```
        if start_key:
            scan_kwargs["ExclusiveStartKey"] = start_key
        response = self.table.scan(**scan_kwargs)
        movies.extend(response.get("Items", []))
        start_key = response.get("LastEvaluatedKey", None)
        done = start_key is None
    except ClientError as err:
        logger.error(
            "Couldn't scan for movies. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise

    return movies


def delete_movie(self, title, year):
    """
    Deletes a movie from the table.

    :param title: The title of the movie to delete.
    :param year: The release year of the movie to delete.
    """
    try:
        self.table.delete_item(Key={"year": year, "title": title})
    except ClientError as err:
        logger.error(
            "Couldn't delete movie %s. Here's why: %s: %s",
            title,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise


def delete_table(self):
    """
    Deletes the table.
    """
    try:
        self.table.delete()
        self.table = None
    except ClientError as err:
```

```
        logger.error(
            "Couldn't delete table. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

Create a helper function to download and extract the sample JSON file.

```
def get_sample_movie_data(movie_file_name):
    """
    Gets sample movie data, either from a local file or by first downloading it
    from
        the Amazon DynamoDB developer guide.

    :param movie_file_name: The local file name where the movie data is stored in
    JSON format.
    :return: The movie data as a dict.
    """
    if not os.path.isfile(movie_file_name):
        print(f"Downloading {movie_file_name}...")
        movie_content = requests.get(
            "https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
samples/moviedata.zip"
        )
        movie_zip = ZipFile(BytesIO(movie_content.content))
        movie_zip.extractall()

    try:
        with open(movie_file_name) as movie_file:
            movie_data = json.load(movie_file, parse_float=Decimal)
    except FileNotFoundError:
        print(
            f"File {movie_file_name} not found. You must first download the file
to "
            "run this demo. See the README for instructions."
        )
        raise
    else:
```

```
# The sample file lists over 4000 movies, return only the first 250.  
return movie_data[:250]
```

Run an interactive scenario to create the table and perform actions on it.

```
def run_scenario(table_name, movie_file_name, dyn_resource):  
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
    print("-" * 88)  
    print("Welcome to the Amazon DynamoDB getting started demo.")  
    print("-" * 88)  
  
    movies = Movies(dyn_resource)  
    movies_exists = movies.exists(table_name)  
    if not movies_exists:  
        print(f"\nCreating table {table_name}...")  
        movies.create_table(table_name)  
        print(f"\nCreated table {movies.table.name}.")  
  
    my_movie = Question.ask_questions(  
        [  
            Question(  
                "title", "Enter the title of a movie you want to add to the  
table: "  
            ),  
            Question("year", "What year was it released? ", Question.is_int),  
            Question(  
                "rating",  
                "On a scale of 1 - 10, how do you rate it? ",  
                Question.is_float,  
                Question.in_range(1, 10),  
            ),  
            Question("plot", "Summarize the plot for me: "),  
        ]  
    )  
    movies.add_movie(**my_movie)  
    print(f"\nAdded '{my_movie['title']}' to '{movies.table.name}'")  
    print("-" * 88)  
  
    movie_update = Question.ask_questions(  
        [  
            Question("rating", "How would you rate the movie now? ",  
                Question.is_float,  
                Question.in_range(1, 10),  
            ),  
            Question("plot", "Summarize the plot for me: "),  
        ]  
    )  
    movies.update_movie(**movie_update)
```

```
[  
    Question(  
        "rating",  
        f"\nLet's update your movie.\nYou rated it {my_movie['rating']},  
what new "  
        f"rating would you give it? ",  
        Question.is_float,  
        Question.in_range(1, 10),  
    ),  
    Question(  
        "plot",  
        f"You summarized the plot as '{my_movie['plot']}'.\nWhat would  
you say now? ",  
    ),  
]  
)  
my_movie.update(movie_update)  
updated = movies.update_movie(**my_movie)  
print(f"\nUpdated '{my_movie['title']}' with new attributes:")  
pprint(updated)  
print("-" * 88)  
  
if not movies_exists:  
    movie_data = get_sample_movie_data(movie_file_name)  
    print(f"\nReading data from '{movie_file_name}' into your table.")  
    movies.write_batch(movie_data)  
    print(f"\nWrote {len(movie_data)} movies into {movies.table.name}.")  
print("-" * 88)  
  
title = "The Lord of the Rings: The Fellowship of the Ring"  
if Question.ask_question(  
    f"Let's move on...do you want to get info about '{title}'? (y/n) ",  
    Question.is_yesno,  
):  
    movie = movies.get_movie(title, 2001)  
    print("\nHere's what I found:")  
    pprint(movie)  
print("-" * 88)  
  
ask_for_year = True  
while ask_for_year:  
    release_year = Question.ask_question(  
        f"\nLet's get a list of movies released in a given year. Enter a year  
between "
```

```
f"1972 and 2018: ",  
    Question.is_int,  
    Question.in_range(1972, 2018),  
)  
releases = movies.query_movies(release_year)  
if releases:  
    print(f"There were {len(releases)} movies released in  
{release_year}:")  
    for release in releases:  
        print(f"\t{release['title']}")  
    ask_for_year = False  
else:  
    print(f"I don't know about any movies released in {release_year}!")  
    ask_for_year = Question.ask_question(  
        "Try another year? (y/n) ", Question.is_yesno  
)  
print("-" * 88)  
  
years = Question.ask_questions(  
    [  
        Question(  
            "first",  
            f"\nNow let's scan for movies released in a range of years. Enter  
a year: ",  
            Question.is_int,  
            Question.in_range(1972, 2018),  
,  
        Question(  
            "second",  
            "Now enter another year: ",  
            Question.is_int,  
            Question.in_range(1972, 2018),  
,  
    ]  
)  
releases = movies.scan_movies(years)  
if releases:  
    count = Question.ask_question(  
        f"\nFound {len(releases)} movies. How many do you want to see? ",  
        Question.is_int,  
        Question.in_range(1, len(releases)),  
)  
    print(f"\nHere are your {count} movies:\n")  
    pprint(releases[:count])
```

```
else:
    print(
        f"I don't know about any movies released between {years['first']} "
        f"and {years['second']}."
    )
print("-" * 88)

if Question.ask_question(
    f"\nLet's remove your movie from the table. Do you want to remove "
    f"'{my_movie['title']}'? (y/n)",
    Question.is_yesno,
):
    movies.delete_movie(my_movie["title"], my_movie["year"])
    print(f"\nRemoved '{my_movie['title']}' from the table.")
print("-" * 88)

if Question.ask_question(f"\nDelete the table? (y/n) ", Question.is_yesno):
    movies.delete_table()
    print(f"Deleted {table_name}.")
else:
    print(
        "Don't forget to delete the table when you're done or you might incur "
        "charges on your account."
    )

print("\nThanks for watching!")
print("-" * 88)

if __name__ == "__main__":
    try:
        run_scenario(
            "doc-example-table-movies", "moviedata.json",
            boto3.resource("dynamodb")
        )
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

This scenario uses the following helper class to ask questions at a command prompt.

```
class Question:
```

```
"""
A helper class to ask questions at a command prompt and validate and convert
the answers.
"""

def __init__(self, key, question, *validators):
    """
    :param key: The key that is used for storing the answer in a dict, when
               multiple questions are asked in a set.
    :param question: The question to ask.
    :param validators: The answer is passed through the list of validators
    until
               one fails or they all pass. Validators may also
    convert the
               answer to another form, such as from a str to an int.
    """
    self.key = key
    self.question = question
    self.validators = Question.non_empty, *validators

    @staticmethod
    def ask_questions(questions):
        """
        Asks a set of questions and stores the answers in a dict.

        :param questions: The list of questions to ask.
        :return: A dict of answers.
        """
        answers = {}
        for question in questions:
            answers[question.key] = Question.ask_question(
                question.question, *question.validators
            )
        return answers

    @staticmethod
    def ask_question(question, *validators):
        """
        Asks a single question and validates it against a list of validators.
        When an answer fails validation, the complaint is printed and the
        question
        is asked again.

        :param question: The question to ask.
        """
```

```
:param validators: The list of validators that the answer must pass.
:return: The answer, converted to its final form by the validators.
"""

answer = None
while answer is None:
    answer = input(question)
    for validator in validators:
        answer, complaint = validator(answer)
        if answer is None:
            print(complaint)
            break
return answer

@staticmethod
def non_empty(answer):
    """
    Validates that the answer is not empty.
    :return: The non-empty answer, or None.
    """
    return answer if answer != "" else None, "I need an answer. Please?"

@staticmethod
def is_yesno(answer):
    """
    Validates a yes/no answer.
    :return: True when the answer is 'y'; otherwise, False.
    """
    return answer.lower() == "y", ""

@staticmethod
def is_int(answer):
    """
    Validates that the answer can be converted to an int.
    :return: The int answer; otherwise, None.
    """
    try:
        int_answer = int(answer)
    except ValueError:
        int_answer = None
    return int_answer, f"{answer} must be a valid integer."

@staticmethod
def is_letter(answer):
    """
```

```
    Validates that the answer is a letter.
    :return The letter answer, converted to uppercase; otherwise, None.
    """
    return (
        answer.upper() if answer.isalpha() else None,
        f"{answer} must be a single letter.",
    )

@staticmethod
def is_float(answer):
    """
    Validate that the answer can be converted to a float.
    :return The float answer; otherwise, None.
    """
    try:
        float_answer = float(answer)
    except ValueError:
        float_answer = None
    return float_answer, f"{answer} must be a valid float."

@staticmethod
def in_range(lower, upper):
    """
    Validate that the answer is within a range. The answer must be of a type
    that can
    be compared to the lower and upper bounds.
    :return: The answer, if it is within the range; otherwise, None.
    """

    def _validate(answer):
        return (
            answer if lower <= answer <= upper else None,
            f"{answer} must be between {lower} and {upper}.",
        )

    return _validate
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)

- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

Ruby

SDK for Ruby

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a class that encapsulates a DynamoDB table.

```
# Creates an Amazon DynamoDB table that can be used to store movie data.  
# The table uses the release year of the movie as the partition key and the  
# title as the sort key.  
  
#  
# @param table_name [String] The name of the table to create.  
# @return [Aws::DynamoDB::Table] The newly created table.  
def create_table(table_name)  
    @table = @dynamo_resource.create_table(  
        table_name: table_name,  
        key_schema: [  
            {attribute_name: "year", key_type: "HASH"}, # Partition key  
            {attribute_name: "title", key_type: "RANGE"} # Sort key  
        ],  
        attribute_definitions: [  
            {attribute_name: "year", attribute_type: "N"},  
            {attribute_name: "title", attribute_type: "S"}  
        ],
```

```
provisioned_throughput: {read_capacity_units: 10, write_capacity_units: 10})  
  @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)  
  @table  
rescue Aws::DynamoDB::Errors::ServiceError => e  
  @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")  
  raise  
end
```

Create a helper function to download and extract the sample JSON file.

```
# Gets sample movie data, either from a local file or by first downloading it  
from  
# the Amazon DynamoDB Developer Guide.  
#  
# @param movie_file_name [String] The local file name where the movie data is  
stored in JSON format.  
# @return [Hash] The movie data as a Hash.  
def fetch_movie_data(movie_file_name)  
  if !File.file?(movie_file_name)  
    @logger.debug("Downloading #{movie_file_name}...")  
    movie_content = URI.open(  
      "https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/  
samples/moviedata.zip"  
    )  
    movie_json = ""  
    Zip::File.open_buffer(movie_content) do |zip|  
      zip.each do |entry|  
        movie_json = entry.get_input_stream.read  
      end  
    end  
  else  
    movie_json = File.read(movie_file_name)  
  end  
  movie_data = JSON.parse(movie_json)  
  # The sample file lists over 4000 movies. This returns only the first 250.  
  movie_data.slice(0, 250)  
rescue StandardError => e  
  puts("Failure downloading movie data:\n#{e}")  
  raise  
end
```

Run an interactive scenario to create the table and perform actions on it.

```
table_name = "doc-example-table-movies-#{rand(10**4)}"
scaffold = Scaffold.new(table_name)
dynamodb_wrapper = DynamoDBBasics.new(table_name)

new_step(1, "Create a new DynamoDB table if none already exists.")
unless scaffold.exists?(table_name)
  puts("\nNo such table: #{table_name}. Creating it...")
  scaffold.create_table(table_name)
  print "Done!\n".green
end

new_step(2, "Add a new record to the DynamoDB table.")
my_movie = {}
my_movie[:title] = CLI::UI::Prompt.ask("Enter the title of a movie to add to the table. E.g. The Matrix")
my_movie[:year] = CLI::UI::Prompt.ask("What year was it released? E.g. 1989").to_i
my_movie[:rating] = CLI::UI::Prompt.ask("On a scale of 1 - 10, how do you rate it? E.g. 7").to_i
my_movie[:plot] = CLI::UI::Prompt.ask("Enter a brief summary of the plot. E.g. A man awakens to a new reality.")
dynamodb_wrapper.add_item(my_movie)
puts("\nNew record added:")
puts JSON.pretty_generate(my_movie).green
print "Done!\n".green

new_step(3, "Update a record in the DynamoDB table.")
my_movie[:rating] = CLI::UI::Prompt.ask("Let's update the movie you added with a new rating, e.g. 3:").to_i
response = dynamodb_wrapper.update_item(my_movie)
puts("Updated '#{my_movie[:title]}' with new attributes:")
puts JSON.pretty_generate(response).green
print "Done!\n".green

new_step(4, "Get a record from the DynamoDB table.")
puts("Searching for #{my_movie[:title]} (#{my_movie[:year]})...")
response = dynamodb_wrapper.get_item(my_movie[:title], my_movie[:year])
puts JSON.pretty_generate(response).green
print "Done!\n".green

new_step(5, "Write a batch of items into the DynamoDB table.")
download_file = "moviedata.json"
```

```
puts("Downloading movie database to #{download_file}...")
movie_data = scaffold.fetch_movie_data(download_file)
puts("Writing movie data from #{download_file} into your table...")
scaffold.write_batch(movie_data)
puts("Records added: #{movie_data.length}.")
print "Done!\n".green

new_step(5, "Query for a batch of items by key.")
loop do
  release_year = CLI::UI::Prompt.ask("Enter a year between 1972 and 2018, e.g. 1999:").to_i
  results = dynamodb_wrapper.query_items(release_year)
  if results.any?
    puts("There were #{results.length} movies released in #{release_year}:")
    results.each do |movie|
      print "\t #{movie["title"]}\n".green
    end
    break
  else
    continue = CLI::UI::Prompt.ask("Found no movies released in #{release_year}! Try another year? (y/n)")
    break if !continue.eql?("y")
  end
end
print "\nDone!\n".green

new_step(6, "Scan for a batch of items using a filter expression.")
years = []
years[:start] = CLI::UI::Prompt.ask("Enter a starting year between 1972 and 2018:")
years[:end] = CLI::UI::Prompt.ask("Enter an ending year between 1972 and 2018:")
releases = dynamodb_wrapper.scan_items(years)
if !releases.empty?
  puts("Found #{releases.length} movies.")
  count = Question.ask(
    "How many do you want to see?", method(:is_int), in_range(1, releases.length))
  puts("Here are your #{count} movies:")
  releases.take(count).each do |release|
    puts("\t#{release["title"]}\n")
  end
else
  puts("I don't know about any movies released between #{years[:start]} " \
```

```
        "and #{years[:end]}")  
    end  
    print "\nDone!\n".green  
  
    new_step(7, "Delete an item from the DynamoDB table.")  
    answer = CLI::UI::Prompt.ask("Do you want to remove '#{my_movie[:title]}'? (y/n)")  
    if answer.eql?("y")  
        dynamodb_wrapper.delete_item(my_movie[:title], my_movie[:year])  
        puts("Removed '#{my_movie[:title]}' from the table.")  
        print "\nDone!\n".green  
    end  
  
    new_step(8, "Delete the DynamoDB table.")  
    answer = CLI::UI::Prompt.ask("Delete the table? (y/n)")  
    if answer.eql?("y")  
        scaffold.delete_table  
        puts("Deleted #{table_name}.")  
    else  
        puts("Don't forget to delete the table when you're done!")  
    end  
    print "\nThanks for watching!\n".green  
rescue Aws::Errors::ServiceError  
    puts("Something went wrong with the demo.")  
rescue Errno::ENOENT  
    true  
end
```

- For API details, see the following topics in *AWS SDK for Ruby API Reference*.

- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)

- [UpdateItem](#)

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
" Create an Amazon Dynamo DB table.

TRY.
  DATA(lo_session) = /aws1/cl_rt_session_awss=>create( cv_pfl ).
  DATA(lo_dyn) = /aws1/cl_dyn_factory=>create( lo_session ).
  DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
    ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                         iv_keytype = 'HASH' ) )
    ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                         iv_keytype = 'RANGE' ) ) ).
  DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
    ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                         iv_attributetype = 'N' ) )
    ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                         iv_attributetype = 'S' ) ) ).

  " Adjust read/write capacities as desired.
  DATA(lo_dynprovthroughput) = NEW /aws1/cl_dynprovthroughput(
    iv_readcapacityunits = 5
    iv_writecapacityunits = 5 ).
  DATA(oo_result) = lo_dyn->createtable(
    it_keyschema = lt_keyschema
    iv_tablename = iv_table_name
    it_attributedefinitions = lt_attributedefinitions
    io_provisionedthroughput = lo_dynprovthroughput ).

  " Table creation can take some time. Wait till table exists before
  returning.
  lo_dyn->get_waiter( )->tableexists(
```

```
        iv_max_wait_time = 200
        iv_tablename      = iv_table_name .
MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
" It throws exception if the table already exists.
CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
    DATA(lv_error) = |" { lo_resourceinuseex->av_err_code }" -
{ lo_resourceinuseex->av_err_msg }|.
    MESSAGE lv_error TYPE 'E'.
ENDTRY.

" Describe table
TRY.
    DATA(lo_table) = lo_dyn->describable( iv_tablename = iv_table_name ).
    DATA(lv_tablename) = lo_table->get_table( )->ask_tablename( ).
MESSAGE 'The table name is ' && lv_tablename TYPE 'I'.
CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table does not exist' TYPE 'E'.
ENDTRY.

" Put items into the table.
TRY.
    DATA(lo_resp_putitem) = lo_dyn->putitem(
        iv_tablename = iv_table_name
        it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
        ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
'Jaws' ) ) )
        ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1975' }| ) ) )
        ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '7.5' }| ) ) )
        ) .
    lo_resp_putitem = lo_dyn->putitem(
        iv_tablename = iv_table_name
        it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
        ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s = 'Star
Wars' ) ) )
        ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
```

```

        key = 'year' value = NEW /aws1/cl_dynamodbattributevalue( iv_n = |
{ '1978' }| ) ) )
        ( VALUE /aws1/cl_dynamodbattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'rating' value = NEW /aws1/cl_dynamodbattributevalue( iv_n = |
{ '8.1' }| ) ) )
    ) ).

    lo_resp_putitem = lo_dyn->putitem(
        iv_tablename = iv_table_name
        it_item      = VALUE /aws1/
cl_dynamodbattribute=>tt_putiteminputattributemap(
        ( VALUE /aws1/cl_dynamodbattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'title' value = NEW /aws1/cl_dynamodbattributevalue( iv_s =
'Speed' ) ) )
        ( VALUE /aws1/cl_dynamodbattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'year' value = NEW /aws1/cl_dynamodbattributevalue( iv_n = |
{ '1994' }| ) ) )
        ( VALUE /aws1/cl_dynamodbattributevalue=>ts_putiteminputattrmap_maprow(
            key = 'rating' value = NEW /aws1/cl_dynamodbattributevalue( iv_n = |
{ '7.9' }| ) ) )
    ) ).

    " TYPE REF TO ZCL_AWS1_dyn_PUT_ITEM_OUTPUT
    MESSAGE '3 rows inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
    CATCH /aws1/cx_dyncondalcheckfaile00.
    MESSAGE 'A condition specified in the operation could not be evaluated.' TYPE 'E'.
    CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table or index does not exist' TYPE 'E'.
    CATCH /aws1/cx_dyntransactconflictex.
    MESSAGE 'Another transaction is using the item' TYPE 'E'.
ENDTRY.

" Get item from table.
TRY.
    DATA(lo_resp_getitem) = lo_dyn->getitem(
        iv_tablename           = iv_table_name
        it_key                 = VALUE /aws1/cl_dynamodbattributevalue=>tt_key(
            ( VALUE /aws1/cl_dynamodbattributevalue=>ts_key_maprow(
                key = 'title' value = NEW /aws1/cl_dynamodbattributevalue( iv_s =
'Jaws' ) ) )
            ( VALUE /aws1/cl_dynamodbattributevalue=>ts_key_maprow(
                key = 'year' value = NEW /aws1/cl_dynamodbattributevalue( iv_n =
'1975' ) ) )
        ) .
    DATA(lt_attr) = lo_resp_getitem->get_item( ).
```

```
        DATA(lo_title) = lt_attr[ key = 'title' ]-value.
        DATA(lo_year) = lt_attr[ key = 'year' ]-value.
        DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
        CATCH /aws1/cx_dynresourcenotfoundex.
            MESSAGE 'The table or index does not exist' TYPE 'E'.
        ENDTRY.

    " Query item from table.
    TRY.
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = '1975' ) ) ).
        DATA(lt_keyconditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
            ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
                key = 'year'
                value = NEW /aws1/cl_dyncondition(
                    it_attributevaluelist = lt_attributelist
                    iv_comparisonoperator = |EQ|
                ) ) ) .
        DATA(lo_query_result) = lo_dyn->query(
            iv_tablename = iv_table_name
            it_keyconditions = lt_keyconditions ).
        DATA(lt_items) = lo_query_result->get_items( ).
        READ TABLE lo_query_result->get_items( ) INTO DATA(lt_item) INDEX 1.
        lo_title = lt_item[ key = 'title' ]-value.
        lo_year = lt_item[ key = 'year' ]-value.
        lo_rating = lt_item[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
        CATCH /aws1/cx_dynresourcenotfoundex.
            MESSAGE 'The table or index does not exist' TYPE 'E'.
        ENDTRY.

    " Scan items from table.
    TRY.
        DATA(lo_scan_result) = lo_dyn->scan( iv_tablename = iv_table_name ).
        lt_items = lo_scan_result->get_items( ).
        " Read the first item and display the attributes.
        READ TABLE lo_query_result->get_items( ) INTO lt_item INDEX 1.
        lo_title = lt_item[ key = 'title' ]-value.
```

```
    lo_year = lt_item[ key = 'year' ]-value.
    lo_rating = lt_item[ key = 'rating' ]-value.
    MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
    MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
    MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
    CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Update items from table.
    TRY.
        DATA(lt_attributeupdates) = VALUE /aws1/
cl_dynattrvalueupdate=>tt_attributeupdates(
            ( VALUE /aws1/cl_dynattrvalueupdate=>ts_attributeupdates_maprow(
                key = 'rating' value = NEW /aws1/cl_dynattrvalueupdate(
                    io_value = NEW /aws1/cl_dynattributevalue( iv_n = '7.6' )
                    iv_action = |PUT| ) ) ) .
        DATA(lt_key) = VALUE /aws1/cl_dynattributevalue=>tt_key(
            ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
                key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n =
'1975' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
                key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
'1980' ) ) ) .
        DATA(lo_resp) = lo_dyn->updateitem(
            iv tablename      = iv_table_name
            it key           = lt_key
            it attributeupdates = lt_attributeupdates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
        CATCH /aws1/cx_dyncondalcheckfaile00.
            MESSAGE 'A condition specified in the operation could not be evaluated.' TYPE 'E'.
        CATCH /aws1/cx_dynresourcenotfoundex.
            MESSAGE 'The table or index does not exist' TYPE 'E'.
        CATCH /aws1/cx_dyntransactconflictex.
            MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.

    " Delete table.
    TRY.
        lo_dyn->deletetable( iv tablename = iv_table_name ).
        lo_dyn->get_waiter( )->tablenotexists(
            iv max wait time = 200
            iv tablename      = iv_table_name ).
```

```
MESSAGE 'DynamoDB Table deleted.' TYPE 'I'.
CATCH /aws1/cx_dynresourcenotfoundex.
MESSAGE 'The table or index does not exist' TYPE 'E'.
CATCH /aws1/cx_dynresourceinuseex.
MESSAGE 'The table cannot be deleted as it is in use' TYPE 'E'.
ENDTRY.
```

- For API details, see the following topics in *AWS SDK for SAP ABAP API reference*.

- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

Swift

SDK for Swift

 **Note**

This is prerelease documentation for an SDK in preview release. It is subject to change.

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

A Swift class that handles DynamoDB calls to the SDK for Swift.

```
import Foundation
import AWSDynamoDB

/// An enumeration of error codes representing issues that can arise when using
/// the `MovieTable` class.
enum MoviesError: Error {
    /// The specified table wasn't found or couldn't be created.
    case TableNotFound
    /// The specified item wasn't found or couldn't be created.
    case ItemNotFound
    /// The Amazon DynamoDB client is not properly initialized.
    case UninitializedClient
    /// The table status reported by Amazon DynamoDB is not recognized.
    case StatusUnknown
    /// One or more specified attribute values are invalid or missing.
    case InvalidAttributes
}

/// A class representing an Amazon DynamoDB table containing movie
/// information.
public class MovieTable {
    var ddbClient: DynamoDBClient? = nil
    let tableName: String

    /// Create an object representing a movie table in an Amazon DynamoDB
    /// database.
    ///
    /// - Parameters:
    ///   - region: The Amazon Region to create the database in.
    ///   - tableName: The name to assign to the table. If not specified, a
    ///     random table name is generated automatically.
    ///
    /// > Note: The table is not necessarily available when this function
    /// returns. Use `tableExists()` to check for its availability, or
    /// `awaitTableActive()` to wait until the table's status is reported as
    /// ready to use by Amazon DynamoDB.
    ///
    init(region: String = "us-east-2", tableName: String) async throws {
        ddbClient = try DynamoDBClient(region: region)
        self.tableName = tableName
    }
}
```

```
        try await self.createTable()
    }

    /**
     * Create a movie table in the Amazon DynamoDB data store.
     */
    private func createTable() async throws {
        guard let client = self.ddbClient else {
            throw MoviesError.UninitializedClient
        }

        let input = CreateTableInput(
            attributeDefinitions: [
                DynamoDBClientTypes.AttributeDefinition(attributeName: "year",
attributeType: .n),
                DynamoDBClientTypes.AttributeDefinition(attributeName: "title",
attributeType: .s),
            ],
            keySchema: [
                DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
            ],
            provisionedThroughput: DynamoDBClientTypes.ProvisionedThroughput(
                readCapacityUnits: 10,
                writeCapacityUnits: 10
            ),
            tableName: self.tableName
        )
        let output = try await client.createTable(input: input)
        if output.tableDescription == nil {
            throw MoviesError.TableNotFound
        }
    }

    /**
     * - Returns: `true` if the table exists, or `false` if not.
     */
    func tableExists() async throws -> Bool {
        guard let client = self.ddbClient else {
            throw MoviesError.UninitializedClient
    }
```

```
let input = DescribeTableInput(
    tableName: tableName
)
let output = try await client.describeTable(input: input)
guard let description = output.table else {
    throw MoviesError.TableNotFound
}

return (description.tableName == self.tableName)
}

////
/// Waits for the table to exist and for its status to be active.
///
func awaitTableActive() async throws {
    while (try await tableExists() == false) {
        Thread.sleep(forTimeInterval: 0.25)
    }

    while (try await getTableStatus() != .active) {
        Thread.sleep(forTimeInterval: 0.25)
    }
}

////
/// Deletes the table from Amazon DynamoDB.
///
func deleteTable() async throws {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = DeleteTableInput(
        tableName: self.tableName
    )
    _ = try await client.deleteTable(input: input)
}

/// Get the table's status.
///
/// - Returns: The table status, as defined by the
///   `DynamoDBClientTypes.TableStatus` enum.
///
```

```
func getTableStatus() async throws -> DynamoDBClientTypes.TableStatus {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = DescribeTableInput(
        tableName: self.tableName
    )
    let output = try await client.describeTable(input: input)
    guard let description = output.table else {
        throw MoviesError.TableNotFound
    }
    guard let status = description.tableStatus else {
        throw MoviesError.StatusUnknown
    }
    return status
}

/// Populate the movie database from the specified JSON file.
///
/// - Parameter jsonPath: Path to a JSON file containing movie data.
///
func populate(jsonPath: String) async throws {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    // Create a Swift `URL` and use it to load the file into a `Data` object. Then decode the JSON into an array of `Movie` objects.

    let fileUrl = URL(fileURLWithPath: jsonPath)
    let jsonData = try Data(contentsOf: fileUrl)

    var movieList = try JSONDecoder().decode([Movie].self, from: jsonData)

    // Truncate the list to the first 200 entries or so for this example.

    if movieList.count > 200 {
        movieList = Array(movieList[...199])
    }

    // Before sending records to the database, break the movie list into
    // 25-entry chunks, which is the maximum size of a batch item request.
```

```
let count = movieList.count
let chunks = stride(from: 0, to: count, by: 25).map {
    Array(movieList[$0 ..< Swift.min($0 + 25, count)])
}

// For each chunk, create a list of write request records and populate
// them with `PutRequest` requests, each specifying one movie from the
// chunk. Once the chunk's items are all in the `PutRequest` list,
// send them to Amazon DynamoDB using the
// `DynamoDBClient.batchWriteItem()` function.

for chunk in chunks {
    var requestList: [DynamoDBClientTypes.WriteRequest] = []

    for movie in chunk {
        let item = try await movie.getAsItem()
        let request = DynamoDBClientTypes.WriteRequest(
            putRequest: .init(
                item: item
            )
        )
        requestList.append(request)
    }

    let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
    _ = try await client.batchWriteItem(input: input)
}
}

/// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
/// table.
///
/// - Parameter movie: The `Movie` to add to the table.
///
func add(movie: Movie) async throws {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
}

// Get a DynamoDB item containing the movie data.
let item = try await movie.getAsItem()

// Send the `PutItem` request to Amazon DynamoDB.
```

```
        let input = PutItemInput(
            item: item,
            tableName: self.tableName
        )
        _ = try await client.putItem(input: input)
    }

    /// Given a movie's details, add a movie to the Amazon DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title as a `String`.
    ///   - year: The release year of the movie (`Int`).
    ///   - rating: The movie's rating if available (`Double`; default is `nil`).
    ///   - plot: A summary of the movie's plot (`String`; default is `nil`, indicating no plot summary is available).
    ///
    func add(title: String, year: Int, rating: Double? = nil,
             plot: String? = nil) async throws {
        let movie = Movie(title: title, year: year, rating: rating, plot: plot)
        try await self.add(movie: movie)
    }

    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        guard let client = self.ddbClient else {
            throw MoviesError.UninitializedClient
        }

        let input = GetItemInput(
            key: [
                "year": .n(String(year)),
                "title": .s(title)
            ],

```

```
        tableName: self.tableName
    )
let output = try await client.getItem(input: input)
guard let item = output.item else {
    throw MoviesError.ItemNotFound
}

let movie = try Movie(withItem: item)
return movie
}

/// Get all the movies released in the specified year.
///
/// - Parameter year: The release year of the movies to return.
///
/// - Returns: An array of `Movie` objects describing each matching movie.
///
func getMovies(fromYear year: Int) async throws -> [Movie] {
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = QueryInput(
        expressionAttributeNames: [
            "#y": "year"
        ],
        expressionAttributeValues: [
            ":y": .n(String(year))
        ],
        keyConditionExpression: "#y = :y",
        tableName: self.tableName
    )
    let output = try await client.query(input: input)

    guard let items = output.items else {
        throw MoviesError.ItemNotFound
    }

    // Convert the found movies into `Movie` objects and return an array
    // of them.

    var movieList: [Movie] = []
    for item in items {
        let movie = try Movie(withItem: item)
```

```
        movieList.append(movie)
    }
    return movieList
}

/// Return an array of `Movie` objects released in the specified range of
/// years.
///
/// - Parameters:
///   - firstYear: The first year of movies to return.
///   - lastYear: The last year of movies to return.
///   - startKey: A starting point to resume processing; always use `nil`.
///
/// - Returns: An array of `Movie` objects describing the matching movies.
///
/// > Note: The `startKey` parameter is used by this function when
/// recursively calling itself, and should always be `nil` when calling
/// directly.
///
func getMovies(firstYear: Int, lastYear: Int,
               startKey: [Swift.String:DynamoDBClientTypes.AttributeValue]? = nil)
    async throws -> [Movie] {
    var movieList: [Movie] = []

    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }

    let input = ScanInput(
        consistentRead: true,
        exclusiveStartKey: startKey,
        expressionAttributeNames: [
            "#y": "year"           // `year` is a reserved word, so use `#y`
instead.
        ],
        expressionAttributeValues: [
            ":y1": .n(String(firstYear)),
            ":y2": .n(String(lastYear))
        ],
        filterExpression: "#y BETWEEN :y1 AND :y2",
        tableName: self.tableName
    )
}
```

```
let output = try await client.scan(input: input)

guard let items = output.items else {
    return movieList
}

// Build an array of `Movie` objects for the returned items.

for item in items {
    let movie = try Movie(withItem: item)
    movieList.append(movie)
}

// Call this function recursively to continue collecting matching
// movies, if necessary.

if output.lastEvaluatedKey != nil {
    let movies = try await self.getMovies(firstYear: firstYear, lastYear:
lastYear,
                                         startKey: output.lastEvaluatedKey)
    movieList += movies
}
return movieList
}

/// Update the specified movie with new `rating` and `plot` information.
///
/// - Parameters:
///   - title: The title of the movie to update.
///   - year: The release year of the movie to update.
///   - rating: The new rating for the movie.
///   - plot: The new plot summary string for the movie.
///
/// - Returns: An array of mappings of attribute names to their new
/// listing each item actually changed. Items that didn't need to change
/// aren't included in this list. `nil` if no changes were made.
///
func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
{
    -> [Swift.String:DynamoDBClientTypes.AttributeValue]?
    guard let client = self.ddbClient else {
        throw MoviesError.UninitializedClient
    }
}
```

```
// Build the update expression and the list of expression attribute
// values. Include only the information that's changed.

var expressionParts: [String] = []
var attrValues: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]

if rating != nil {
    expressionParts.append("info.rating=:r")
    attrValues[":r"] = .n(String(rating!))
}
if plot != nil {
    expressionParts.append("info.plot=:p")
    attrValues[":p"] = .s(plot!)
}
let expression: String = "set \\" + (expressionParts.joined(separator: ", ")) + "\\"

let input = UpdateItemInput(
    // Create substitution tokens for the attribute values, to ensure
    // no conflicts in expression syntax.
    expressionAttributeValues: attrValues,
    // The key identifying the movie to update consists of the release
    // year and title.
    key: [
        "year": .n(String(year)),
        "title": .s(title)
    ],
    returnValues: .updatedNew,
    tableName: self.tableName,
    updateExpression: expression
)
let output = try await client.updateItem(input: input)

guard let attributes: [Swift.String:DynamoDBClientTypes.AttributeValue] =
output.attributes else {
    throw MoviesError.InvalidAttributes
}
return attributes
}

/// Delete a movie, given its title and release year.
///
/// - Parameters:
///   - title: The movie's title.
///   - year: The movie's release year.
```

```
///  
func delete(title: String, year: Int) async throws {  
    guard let client = self.ddbClient else {  
        throw MoviesError.UninitializedClient  
    }  
  
    let input = DeleteItemInput(  
        key: [  
            "year": .n(String(year)),  
            "title": .s(title)  
        ],  
        tableName: self.tableName  
    )  
    _ = try await client.deleteItem(input: input)  
}  
}
```

The structures used by the MovieTable class to represent movies.

```
import Foundation  
import AWSDynamoDB  
  
/// The optional details about a movie.  
public struct Details: Codable {  
    /// The movie's rating, if available.  
    var rating: Double?  
    /// The movie's plot, if available.  
    var plot: String?  
}  
  
/// A structure describing a movie. The `year` and `title` properties are  
/// required and are used as the key for Amazon DynamoDB operations. The  
/// `info` sub-structure's two properties, `rating` and `plot`, are optional.  
public struct Movie: Codable {  
    /// The year in which the movie was released.  
    var year: Int  
    /// The movie's title.  
    var title: String  
    /// A `Details` object providing the optional movie rating and plot  
    /// information.  
    var info: Details
```

```
/// Create a `Movie` object representing a movie, given the movie's
/// details.
///
/// - Parameters:
///   - title: The movie's title (`String`).
///   - year: The year in which the movie was released (`Int`).
///   - rating: The movie's rating (optional `Double`).
///   - plot: The movie's plot (optional `String`)
init(title: String, year: Int, rating: Double? = nil, plot: String? = nil) {
    self.title = title
    self.year = year

    self.info = Details(rating: rating, plot: plot)
}

/// Create a `Movie` object representing a movie, given the movie's
/// details.
///
/// - Parameters:
///   - title: The movie's title (`String`).
///   - year: The year in which the movie was released (`Int`).
///   - info: The optional rating and plot information for the movie in a
///         `Details` object.
init(title: String, year: Int, info: Details?){
    self.title = title
    self.year = year

    if info != nil {
        self.info = info!
    } else {
        self.info = Details(rating: nil, plot: nil)
    }
}

///
/// Return a new `MovieTable` object, given an array mapping string to Amazon
/// DynamoDB attribute values.
///
/// - Parameter item: The item information provided to the form used by
/// DynamoDB. This is an array of strings mapped to
/// `DynamoDBClientTypes.AttributeValue` values.
init(withItem item: [Swift.String:DynamoDBClientTypes.AttributeValue]) throws
{
    // Read the attributes.
```

```
guard let titleAttr = item["title"],
       let yearAttr = item["year"] else {
    throw MoviesError.ItemNotFound
}
let infoAttr = item["info"] ?? nil

// Extract the values of the title and year attributes.

if case .s(let titleVal) = titleAttr {
    self.title = titleVal
} else {
    throw MoviesError.InvalidAttributes
}

if case .n(let yearVal) = yearAttr {
    self.year = Int(yearVal)!
} else {
    throw MoviesError.InvalidAttributes
}

// Extract the rating and/or plot from the `info` attribute, if
// they're present.

var rating: Double? = nil
var plot: String? = nil

if infoAttr != nil, case .m(let infoVal) = infoAttr {
    let ratingAttr = infoVal["rating"] ?? nil
    let plotAttr = infoVal["plot"] ?? nil

    if ratingAttr != nil, case .n(let ratingVal) = ratingAttr {
        rating = Double(ratingVal) ?? nil
    }
    if plotAttr != nil, case .s(let plotVal) = plotAttr {
        plot = plotVal
    }
}

self.info = Details(rating: rating, plot: plot)
}

/// 
/// Return an array mapping attribute names to Amazon DynamoDB attribute
```

```
/// values, representing the contents of the `Movie` record as a DynamoDB
/// item.
///
/// - Returns: The movie item as an array of type
///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
///
func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue] {
    // Build the item record, starting with the year and title, which are
    // always present.

    var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
        "year": .n(String(self.year)),
        "title": .s(self.title)
    ]

    // Add the `info` field with the rating and/or plot if they're
    // available.

    var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
    if (self.info.rating != nil || self.info.plot != nil) {
        if self.info.rating != nil {
            details["rating"] = .n(String(self.info.rating!))
        }
        if self.info.plot != nil {
            details["plot"] = .s(self.info.plot!)
        }
    }
    item["info"] = .m(details)

    return item
}
```

A program that uses the MovieTable class to access a DynamoDB database.

```
import Foundation
import ArgumentParser
import AWSDynamoDB
import ClientRuntime

@testable import MovieList
```

```
struct ExampleCommand: ParsableCommand {
    @Argument(help: "The path of the sample movie data JSON file.")
    var jsonPath: String = "../../../../../resources/sample_files/movies.json"

    @Option(help: "The AWS Region to run AWS API calls in.")
    var awsRegion = "us-east-2"

    @Option(
        help: ArgumentHelp("The level of logging for the Swift SDK to perform."),
        completion: .list([
            "critical",
            "debug",
            "error",
            "info",
            "notice",
            "trace",
            "warning"
        ])
    )
    var logLevel: String = "error"

    /// Configuration details for the command.
    static var configuration = CommandConfiguration(
        commandName: "basics",
        abstract: "A basic scenario demonstrating the usage of Amazon DynamoDB.",
        discussion: """
        An example showing how to use Amazon DynamoDB to perform a series of
        common database activities on a simple movie database.
        """
    )

    /// Called by ``main()`` to asynchronously run the AWS example.
    func runAsync() async throws {
        print("Welcome to the AWS SDK for Swift basic scenario for Amazon
DynamoDB!")
        SDKLoggingSystem.initialize(logLevel: .error)

        //=====
        // 1. Create the table. The Amazon DynamoDB table is represented by
        //     the `MovieTable` class.
        //=====

        let tableName = "ddb-movies-sample-\(Int.random(in: 1...Int.max))"
    }
}
```

```
//let tableName = String.uniqueName(withPrefix: "ddb-movies-sample",
maxDigits: 8)

print("Creating table \"\$(tableName)\"...")

let movieDatabase = try await MovieTable(region: awsRegion,
                                         tableName: tableName)

print("\nWaiting for table to be ready to use...")
try await movieDatabase.awaitTableActive()

//=====
// 2. Add a movie to the table.
//=====

print("\nAdding a movie...")
try await movieDatabase.add(title: "Avatar: The Way of Water", year:
2022)
try await movieDatabase.add(title: "Not a Real Movie", year: 2023)

//=====
// 3. Update the plot and rating of the movie using an update
//    expression.
//=====

print("\nAdding details to the added movie...")
_ = try await movieDatabase.update(title: "Avatar: The Way of Water",
year: 2022,
                                    rating: 9.2, plot: "It's a sequel.")

//=====
// 4. Populate the table from the JSON file.
//=====

print("\nPopulating the movie database from JSON...")
try await movieDatabase.populate(jsonPath: jsonPath)

//=====
// 5. Get a specific movie by key. In this example, the key is a
//    combination of `title` and `year`.
//=====

print("\nLooking for a movie in the table...")
```

```
let gotMovie = try await movieDatabase.get(title: "This Is the End",
year: 2013)

    print("Found the movie \"\$(gotMovie.title)\", released in
\$(gotMovie.year).")
    print("Rating: \$(gotMovie.info.rating ?? 0.0).")
    print("Plot summary: \$(gotMovie.info.plot ?? "None.")")

//=====
// 6. Delete a movie.
//=====

print("\nDeleting the added movie...")
try await movieDatabase.delete(title: "Avatar: The Way of Water", year:
2022)

//=====
// 7. Use a query with a key condition expression to return all movies
//     released in a given year.
//=====

print("\nGetting movies released in 1994...")
let movieList = try await movieDatabase.getMovies(fromYear: 1994)
for movie in movieList {
    print("    \$(movie.title)")
}

//=====
// 8. Use `scan()` to return movies released in a range of years.
//=====

print("\nGetting movies released between 1993 and 1997...")
let scannedMovies = try await movieDatabase.getMovies(firstYear: 1993,
lastYear: 1997)
for movie in scannedMovies {
    print("    \$(movie.title) (\$(movie.year))")
}

//=====
// 9. Delete the table.
//=====

print("\nDeleting the table...")
try await movieDatabase.deleteTable()
```

```
    }
}

@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.runAsync()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Swift API reference*.

- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table by using batches of PartiQL statements and an AWS SDK

The following code examples show how to:

- Get a batch of items by running multiple SELECT statements.
- Add a batch of items by running multiple INSERT statements.
- Update a batch of items by running multiple UPDATE statements.
- Delete a batch of items by running multiple DELETE statements.

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Before you run this example, download 'movies.json' from
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
GettingStarted.Js.02.html,
// and put it in the same folder as the example.

// Separator for the console display.
var SepBar = new string('-', 80);
const string tableName = "movie_table";
const string movieFileName = "moviedata.json";

DisplayInstructions();

// Create the table and wait for it to be active.
Console.WriteLine($"Creating the movie table: {tableName}");

var success = await DynamoDBMethods.CreateMovieTableAsync(tableName);
if (success)
{
```

```
        Console.WriteLine($"Successfully created table: {tableName}.");
    }

    WaitForEnter();

    // Add movie information to the table from moviedata.json. See the
    // instructions at the top of this file to download the JSON file.
    Console.WriteLine($"Inserting movies into the new table. Please wait...\"");
    success = await PartiQLBatchMethods.InsertMovies(tableName, movieFileName);
    if (success)
    {
        Console.WriteLine("Movies successfully added to the table.");
    }
    else
    {
        Console.WriteLine("Movies could not be added to the table.");
    }

    WaitForEnter();

    // Update multiple movies by using the BatchExecute statement.
    var title1 = "Star Wars";
    var year1 = 1977;
    var title2 = "Wizard of Oz";
    var year2 = 1939;

    Console.WriteLine($"Updating two movies with producer information: {title1} and
        {title2}.");
    success = await PartiQLBatchMethods.GetBatch(tableName, title1, title2, year1,
        year2);
    if (success)
    {
        Console.WriteLine($"Successfully retrieved {title1} and {title2}.");
    }
    else
    {
        Console.WriteLine("Select statement failed.");
    }

    WaitForEnter();

    // Update multiple movies by using the BatchExecute statement.
    var producer1 = "LucasFilm";
    var producer2 = "MGM";
```

```
Console.WriteLine($"Updating two movies with producer information: {title1} and {title2}.");
success = await PartiQLBatchMethods.UpdateBatch(tableName, producer1, title1,
year1, producer2, title2, year2);
if (success)
{
    Console.WriteLine($"Successfully updated {title1} and {title2}.");
}
else
{
    Console.WriteLine("Update failed.");
}

WaitForEnter();

// Delete multiple movies by using the BatchExecute statement.
Console.WriteLine($"Now we will delete {title1} and {title2} from the table.");
success = await PartiQLBatchMethods.DeleteBatch(tableName, title1, year1, title2,
year2);

if (success)
{
    Console.WriteLine($"Deleted {title1} and {title2}");
}
else
{
    Console.WriteLine($"could not delete {title1} or {title2}");
}

WaitForEnter();

// DNow that the PartiQL Batch scenario is complete, delete the movie table.
success = await DynamoDBMethods.DeleteTableAsync(tableName);

if (success)
{
    Console.WriteLine($"Successfully deleted {tableName}");
}
else
{
    Console.WriteLine($"Could not delete {tableName}");
}
```

```
/// <summary>
/// Displays the description of the application on the console.
/// </summary>
void DisplayInstructions()
{
    Console.Clear();
    Console.WriteLine();
    Console.Write(new string(' ', 24));
    Console.WriteLine("DynamoDB PartiQL Basics Example");
    Console.WriteLine(SepBar);
    Console.WriteLine("This demo application shows the basics of using Amazon
DynamoDB with the AWS SDK for");
    Console.WriteLine(".NET version 3.7 and .NET 6.");
    Console.WriteLine(SepBar);
    Console.WriteLine("Creates a table by using the CreateTable method.");
    Console.WriteLine("Gets multiple movies by using a PartiQL SELECT
statement.");
    Console.WriteLine("Updates multiple movies by using the ExecuteBatch
method.");
    Console.WriteLine("Deletes multiple movies by using a PartiQL DELETE
statement.");
    Console.WriteLine("Cleans up the resources created for the demo by deleting
the table.");
    Console.WriteLine(SepBar);

    WaitForEnter();
}

/// <summary>
/// Simple method to wait for the <Enter> key to be pressed.
/// </summary>
void WaitForEnter()
{
    Console.WriteLine("\nPress <Enter> to continue.");
    Console.Write(SepBar);
    _ = Console.ReadLine();
}

/// <summary>
/// Gets movies from the movie table by
/// using an Amazon DynamoDB PartiQL SELECT statement.
/// </summary>
/// <param name="tableName">The name of the table.</param>
```

```
/// <param name="title1">The title of the first movie.</param>
/// <param name="title2">The title of the second movie.</param>
/// <param name="year1">The year of the first movie.</param>
/// <param name="year2">The year of the second movie.</param>
/// <returns>True if successful.</returns>
public static async Task<bool> GetBatch(
    string tableName,
    string title1,
    string title2,
    int year1,
    int year2)
{
    var getBatch = $"SELECT FROM {tableName} WHERE title = ? AND year
= ?";
    var statements = new List<BatchStatementRequest>
    {
        new BatchStatementRequest
        {
            Statement = getBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = title1 },
                new AttributeValue { N = year1.ToString() },
            },
        },
        new BatchStatementRequest
        {
            Statement = getBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = title2 },
                new AttributeValue { N = year2.ToString() },
            },
        }
    };
    var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
{
    Statements = statements,
});
if (response.Responses.Count > 0)
```

```
{  
    response.Responses.ForEach(r =>  
    {  
        Console.WriteLine($"{r.Item["title"]}\t{r.Item["year"]});  
    });  
    return true;  
}  
else  
{  
    Console.WriteLine($"Couldn't find either {title1} or {title2}.");  
    return false;  
}  
  
}  
  
/// <summary>  
/// Inserts movies imported from a JSON file into the movie table by  
/// using an Amazon DynamoDB PartiQL INSERT statement.  
/// </summary>  
/// <param name="tableName">The name of the table into which the movie  
/// information will be inserted.</param>  
/// <param name="movieFileName">The name of the JSON file that contains  
/// movie information.</param>  
/// <returns>A Boolean value that indicates the success or failure of  
/// the insert operation.</returns>  
public static async Task<bool> InsertMovies(string tableName, string  
movieFileName)  
{  
    // Get the list of movies from the JSON file.  
    var movies = ImportMovies(movieFileName);  
  
    var success = false;  
  
    if (movies is not null)  
    {  
        // Insert the movies in a batch using PartiQL. Because the  
        // batch can contain a maximum of 25 items, insert 25 movies  
        // at a time.  
        string insertBatch = $"INSERT INTO {tableName} VALUE  
{{'title': ?, 'year': ?}}";  
        var statements = new List<BatchStatementRequest>();  
  
        try  
        {  
            foreach (var movie in movies)  
            {  
                statements.Add(new BatchStatementRequest  
                {  
                    Statement = insertBatch,  
                    Parameters = new Dictionary<string, object>()  
                    {  
                        {"title", movie.Title},  
                        {"year", movie.Year}  
                    }  
                });  
                if (statements.Count == 25)  
                {  
                    await client.BatchWriteItemAsync(tableName, statements);  
                    statements.Clear();  
                }  
            }  
            if (statements.Count > 0)  
            {  
                await client.BatchWriteItemAsync(tableName, statements);  
            }  
            success = true;  
        }  
        catch (AmazonDynamoDBException ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}
```

```
        for (var indexOffset = 0; indexOffset < 250; indexOffset +=  
25)  
    {  
        for (var i = indexOffset; i < indexOffset + 25; i++)  
        {  
            statements.Add(new BatchStatementRequest  
            {  
                Statement = insertBatch,  
                Parameters = new List<AttributeValue>  
                {  
                    new AttributeValue { S = movies[i].Title },  
                    new AttributeValue { N =  
movies[i].Year.ToString() },  
                },  
            });  
        }  
  
        var response = await  
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest  
{  
    Statements = statements,  
});  
  
// Wait between batches for movies to be successfully  
added.  
System.Threading.Thread.Sleep(3000);  
  
success = response.StatusCode ==  
System.Net.HttpStatusCode.OK;  
  
// Clear the list of statements for the next batch.  
statements.Clear();  
}  
}  
catch (AmazonDynamoDBException ex)  
{  
    Console.WriteLine(ex.Message);  
}  
}  
  
return success;  
}  
  
/// <summary>
```

```
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The full path to the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public static List<Movie> ImportMovies(string movieFileName)
{
    if (!File.Exists(movieFileName))
    {
        return null!;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

    if (allMovies is not null)
    {
        // Return the first 250 entries.
        return allMovies.GetRange(0, 250);
    }
    else
    {
        return null!;
    }
}

/// <summary>
/// Updates information for multiple movies.
/// </summary>
/// <param name="tableName">The name of the table containing the
/// movies to be updated.</param>
/// <param name="producer1">The producer name for the first movie
/// to update.</param>
/// <param name="title1">The title of the first movie.</param>
/// <param name="year1">The year that the first movie was released.</
param>
/// <param name="producer2">The producer name for the second
/// movie to update.</param>
/// <param name="title2">The title of the second movie.</param>
/// <param name="year2">The year that the second movie was released.</
param>
/// <returns>A Boolean value that indicates the success of the update.</
returns>
```

```
public static async Task<bool> UpdateBatch(
    string tableName,
    string producer1,
    string title1,
    int year1,
    string producer2,
    string title2,
    int year2)
{
    string updateBatch = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";
    var statements = new List<BatchStatementRequest>
    {
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = producer1 },
                new AttributeValue { S = title1 },
                new AttributeValue { N = year1.ToString() },
            },
        },
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = producer2 },
                new AttributeValue { S = title2 },
                new AttributeValue { N = year2.ToString() },
            },
        }
    };
    var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
{
    Statements = statements,
});
return response.StatusCode == System.Net.HttpStatusCode.OK;
```

```
}

/// <summary>
/// Deletes multiple movies using a PartiQL BatchExecuteAsync
/// statement.
/// </summary>
/// <param name="tableName">The name of the table containing the
/// moves that will be deleted.</param>
/// <param name="title1">The title of the first movie.</param>
/// <param name="year1">The year the first movie was released.</param>
/// <param name="title2">The title of the second movie.</param>
/// <param name="year2">The year the second movie was released.</param>
/// <returns>A Boolean value indicating the success of the operation.</returns>
public static async Task<bool> DeleteBatch(
    string tableName,
    string title1,
    int year1,
    string title2,
    int year2)
{
    string updateBatch = $"DELETE FROM {tableName} WHERE title = ? AND
year = ?";
    var statements = new List<BatchStatementRequest>
    {
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = title1 },
                new AttributeValue { N = year1.ToString() },
            },
        },
        new BatchStatementRequest
        {
            Statement = updateBatch,
            Parameters = new List<AttributeValue>
            {
                new AttributeValue { S = title2 },
                new AttributeValue { N = year2.ToString() },
            },
        },
    };
}
```

```
        }

        var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
{
    Statements = statements,
});

return response.StatusCode == System.Net.HttpStatusCode.OK;
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
Aws::Client::ClientConfiguration clientConfig;
// 1. Create a table. (CreateTable)
if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

    AwsDoc::DynamoDB::partiqlBatchExecuteScenario(clientConfig);

    // 7. Delete the table. (DeleteTable)
    AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
}

//! Scenario to modify and query a DynamoDB table using PartiQL batch statements.
/*! 
 \sa partiqlBatchExecuteScenario()
 \param clientConfiguration: AWS client configuration.
 \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::partiqlBatchExecuteScenario(
```

```
const Aws::Client::ClientConfiguration &clientConfiguration) {

    // 2. Add multiple movies using "Insert" statements. (BatchExecuteStatement)
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::vector<Aws::String> titles;
    std::vector<float> ratings;
    std::vector<int> years;
    std::vector<Aws::String> plots;
    Aws::String doAgain = "n";
    do {
        Aws::String aTitle = askQuestion(
            "Enter the title of a movie you want to add to the table: ");
        titles.push_back(aTitle);
        int aYear = askQuestionForInt("What year was it released? ");
        years.push_back(aYear);
        float aRating = askQuestionForFloatRange(
            "On a scale of 1 - 10, how do you rate it? ",
            1, 10);
        ratings.push_back(aRating);
        Aws::String aPlot = askQuestion("Summarize the plot for me: ");
        plots.push_back(aPlot);

        doAgain = askQuestion(Aws::String("Would you like to add more movies? (y/n) "));
    } while (doAgain == "y");

    std::cout << "Adding " << titles.size()
        << (titles.size() == 1 ? " movie " : " movies ")
        << "to the table using a batch \"INSERT\" statement." << std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
            titles.size());

        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \""
            << MOVIE_TABLE_NAME << "\" VALUE {\""
            << TITLE_KEY << "': ?, '\" << YEAR_KEY << "': ?, '\""
            << INFO_KEY << "': ?\"}";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
        }
    }
}
```

```
Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
attributes.push_back(
    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));

// Create attribute for the info map.
Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute
= Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
    ALLOCATION_TAG.c_str());
ratingAttribute->SetN(ratings[i]);
infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
    ALLOCATION_TAG.c_str());
plotAttribute->SetS(plots[i]);
infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
attributes.push_back(infoMapAttribute);
statements[i].SetParameters(attributes);
}

Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

request.SetStatements(statements);

Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
    request);
if (!outcome.IsSuccess()) {
    std::cerr << "Failed to add the movies: " <<
outcome.GetError().GetMessage()
    << std::endl;
    return false;
}
}

std::cout << "Retrieving the movie data with a batch \"SELECT\" statement."
    << std::endl;
```

```
// 3. Get the data for multiple movies using "Select" statements.
(BatchExecuteStatement)
{
    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
        titles.size());
    std::stringstream sqlStream;
    sqlStream << "SELECT * FROM `"
        << MOVIE_TABLE_NAME << "` WHERE "
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";

    std::string sql(sqlStream.str());

    for (size_t i = 0; i < statements.size(); ++i) {
        statements[i].SetStatement(sql);
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(
            Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
        statements[i].SetParameters(attributes);
    }

    Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

    request.SetStatements(statements);

    Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
        request);
    if (outcome.IsSuccess()) {
        const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();

        const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

        for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
            const Aws::Map< Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

            printMovieInfo(item);
        }
    }
    else {

```

```
        std::cerr << "Failed to retrieve the movie information: "
        << outcome.GetError().GetMessage() << std::endl;
    return false;
}

}

// 4. Update the data for multiple movies using "Update" statements.
(BatchExecuteStatement)

for (size_t i = 0; i < titles.size(); ++i) {
    ratings[i] = askQuestionForFloatRange(
        Aws::String("\nLet's update your the movie, \\"") + titles[i] +
        ".\nYou rated it " + std::to_string(ratings[i]) +
        ", what new rating would you give it? ", 1, 10);
}

std::cout << "Updating the movie with a batch \"UPDATE\" statement." <<
std::endl;

{

    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
        titles.size());

    std::stringstream sqlStream;
    sqlStream << "UPDATE \"<< MOVIE_TABLE_NAME << \" SET "
    << INFO_KEY << "." << RATING_KEY << "=? WHERE "
    << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";

    std::string sql(sqlStream.str());

    for (size_t i = 0; i < statements.size(); ++i) {
        statements[i].SetStatement(sql);

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(
            Aws::DynamoDB::Model::AttributeValue().SetN(ratings[i]));
        attributes.push_back(
            Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
        statements[i].SetParameters(attributes);
    }
}
```

```
Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

request.SetStatements(statements);
Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
    request);
if (!outcome.IsSuccess()) {
    std::cerr << "Failed to update movie information: "
        << outcome.GetError().GetMessage() << std::endl;
    return false;
}
}

std::cout << "Retrieving the updated movie data with a batch \"SELECT\""
statement."
<< std::endl;

// 5. Get the updated data for multiple movies using "Select" statements.
(BatchExecuteStatement)
{
    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
        titles.size());
    std::stringstream sqlStream;
    sqlStream << "SELECT * FROM \\""
        << MOVIE_TABLE_NAME << "\\" WHERE "
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";

    std::string sql(sqlStream.str());

    for (size_t i = 0; i < statements.size(); ++i) {
        statements[i].SetStatement(sql);
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(
            Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
        statements[i].SetParameters(attributes);
    }

    Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

    request.SetStatements(statements);

    Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
```

```
        request);
    if (outcome.IsSuccess()) {
        const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
        outcome.GetResult();

        const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

        for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
            const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

            printMovieInfo(item);
        }
    }
    else {
        std::cerr << "Failed to retrieve the movies information: "
        << outcome.GetError().GetMessage() << std::endl;
        return false;
    }
}

std::cout << "Deleting the movie data with a batch \"DELETE\" statement."
<< std::endl;

// 6. Delete multiple movies using "Delete" statements.
(BatchExecuteStatement)
{
    Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
        titles.size());
    std::stringstream sqlStream;
    sqlStream << "DELETE FROM  \"\" " << MOVIE_TABLE_NAME << "\" WHERE "
    << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";

    std::string sql(sqlStream.str());

    for (size_t i = 0; i < statements.size(); ++i) {
        statements[i].SetStatement(sql);
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(
            Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
    }
}
```

```
        statements[i].SetParameters(attributes);
    }

    Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

    request.SetStatements(statements);

    Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
        request);

    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to delete the movies: "
        << outcome.GetError().GetMessage() << std::endl;
        return false;
    }
}

return true;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*! \sa createMoviesDynamoDBTable()
 * \param clientConfiguration: AWS client configuration.
 * \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

bool movieTableAlreadyExisted = false;

{
    Aws::DynamoDB::Model::CreateTableRequest request;

    Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
    yearAttributeDefinition.SetAttributeName(YEAR_KEY);
    yearAttributeDefinition.SetAttributeType(
        Aws::DynamoDB::Model::ScalarAttributeType::N);
    request.AddAttributeDefinitions(yearAttributeDefinition);

    Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
    yearAttributeDefinition.SetAttributeName(TITLE_KEY);
```

```
yearAttributeDefinition.SetAttributeType(
    Aws::DynamoDB::Model::ScalarAttributeType::S);
request.AddAttributeDefinitions(yearAttributeDefinition);

Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
    Aws::DynamoDB::Model::KeyType::HASH);
request.AddKeySchema(yearKeySchema);

Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
    Aws::DynamoDB::Model::KeyType::RANGE);
request.AddKeySchema(titleKeySchema);

Aws::DynamoDB::Model::ProvisionedThroughput throughput;
throughput.WithReadCapacityUnits(
    PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
    PROVISIONED_THROUGHPUT_UNITS);
request.SetProvisionedThroughput(throughput);
request.SetTableName(MOVIE_TABLE_NAME);

std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
    request);
if (!result.IsSuccess()) {
    if (result.GetError().GetErrorCode() ==
        Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
        std::cout << "Table already exists." << std::endl;
        movieTableAlreadyExisted = true;
    }
    else {
        std::cerr << "Failed to create table: "
            << result.GetError().GetMessage();
        return false;
    }
}
}

// Wait for table to become active.
if (!movieTableAlreadyExisted) {
    std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
        << "' to become active...." << std::endl;
```

```
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
            << std::endl;
    }

    return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
 \sa deleteMoviesDynamoDBTable()
 \param clientConfiguration: AWS client configuration.
 \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
dynamoClient.DeleteTable(
        request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
            << result.GetResult().GetTableDescription().GetTableName()
            << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
            << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*! \sa waitTableActive()
 \param waitTableActive: The DynamoDB table's name.
```

```
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                         const Aws::Client::ClientConfiguration
                                         &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
dynamoClient.DescribeTable(
            request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                  << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for C++ API Reference*.

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run a scenario that creates a table and runs batches of PartiQL queries.

```
// RunPartiQLBatchScenario shows you how to use the AWS SDK for Go
// to run batches of PartiQL statements to query a table that stores data about
// movies.
//
//   - Use batches of PartiQL statements to add, get, update, and delete data for
//     individual movies.
//
// This example creates an Amazon DynamoDB service client from the specified
// sdkConfig so that
// you can replace it with a mocked or stubbed config for unit testing.
//
// This example creates and deletes a DynamoDB table to use during the scenario.
func RunPartiQLBatchScenario(sdkConfig aws.Config, tableName string) {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("Something went wrong with the demo.")
        }
    }()
}

log.Println(strings.Repeat("-", 88))
log.Println("Welcome to the Amazon DynamoDB PartiQL batch demo.")
log.Println(strings.Repeat("-", 88))

tableBasics := actions.TableBasics{
    DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
    TableName:       tableName,
}
runner := actions.PartiQLRunner{
    DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
    TableName:       tableName,
```

```
}

exists, err := tableBasics.TableExists()
if err != nil {
    panic(err)
}
if !exists {
    log.Printf("Creating table %v...\n", tableName)
    _, err = tableBasics.CreateMovieTable()
    if err != nil {
        panic(err)
    } else {
        log.Printf("Created table %v.\n", tableName)
    }
} else {
    log.Printf("Table %v already exists.\n", tableName)
}
log.Println(strings.Repeat("-", 88))

currentYear, _, _ := time.Now().Date()
customMovies := []actions.Movie{
    Title: "House PartiQL",
    Year: currentYear - 5,
    Info: map[string]interface{}{
        "plot": "Wacky high jinks result from querying a mysterious database.",
        "rating": 8.5},
    Title: "House PartiQL 2",
    Year: currentYear - 3,
    Info: map[string]interface{}{
        "plot": "Moderate high jinks result from querying another mysterious
database.",
        "rating": 6.5},
    Title: "House PartiQL 3",
    Year: currentYear - 1,
    Info: map[string]interface{}{
        "plot": "Tepid high jinks result from querying yet another mysterious
database.",
        "rating": 2.5},
},
}

log.Printf("Inserting a batch of movies into table '%v'.\n", tableName)
err = runner.AddMovieBatch(customMovies)
if err == nil {
```

```
    log.Printf("Added %v movies to the table.\n", len(customMovies))
}

log.Println(strings.Repeat("-", 88))

log.Println("Getting data for a batch of movies.")
movies, err := runner.GetMovieBatch(customMovies)
if err == nil {
    for _, movie := range movies {
        log.Println(movie)
    }
}
log.Println(strings.Repeat("-", 88))

newRatings := []float64{7.7, 4.4, 1.1}
log.Println("Updating a batch of movies with new ratings.")
err = runner.UpdateMovieBatch(customMovies, newRatings)
if err == nil {
    log.Printf("Updated %v movies with new ratings.\n", len(customMovies))
}
log.Println(strings.Repeat("-", 88))

log.Println("Getting projected data from the table to verify our update.")
log.Println("Using a page size of 2 to demonstrate paging.")
projections, err := runner.GetAllMovies(2)
if err == nil {
    log.Println("All movies:")
    for _, projection := range projections {
        log.Println(projection)
    }
}
log.Println(strings.Repeat("-", 88))

log.Println("Deleting a batch of movies.")
err = runner.DeleteMovieBatch(customMovies)
if err == nil {
    log.Printf("Deleted %v movies.\n", len(customMovies))
}

err = tableBasics.DeleteTable()
if err == nil {
    log.Printf("Deleted table %v.\n", tableBasics.TableName)
}

log.Println(strings.Repeat("-", 88))
```

```
    log.Println("Thanks for watching!")
    log.Println(strings.Repeat("-", 88))
}
```

Define a Movie struct that is used in this example.

```
// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string           `dynamodbav:"title"`
    Year  int              `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

Create a struct and methods that run PartiQL statements.

```
// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
// the
// specified table.
type PartiQLRunner struct {
    DynamoDbClient *dynamodb.Client
    TableName       string
}

// AddMovieBatch runs a batch of PartiQL INSERT statements to add multiple movies
// to the
// DynamoDB table.
func (runner PartiQLRunner) AddMovieBatch(movies []Movie) error {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{movie.Title,
            movie.Year, movie.Info})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(fmt.Sprintf(
                "INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
                runner.TableName)),
            Parameters: params,
        }
    }

    _, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
        &dynamodb.BatchExecuteStatementInput{
            Statements: statementRequests,
        })
    if err != nil {
        log.Printf("Couldn't insert a batch of items with PartiQL. Here's why: %v\n",
            err)
    }
    return err
}
```

```
// GetMovieBatch runs a batch of PartiQL SELECT statements to get multiple movies
// from
// the DynamoDB table by title and year.
func (runner PartiQLRunner) GetMovieBatch(movies []Movie) ([]Movie, error) {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{movie.Title,
            movie.Year})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(
                fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?", runner.TableName)),
            Parameters: params,
        }
    }

    output, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
        &dynamodb.BatchExecuteStatementInput{
            Statements: statementRequests,
        })
    var outMovies []Movie
    if err != nil {
        log.Printf("Couldn't get a batch of items with PartiQL. Here's why: %v\n", err)
    } else {
        for _, response := range output.Responses {
            var movie Movie
            err = attributevalue.UnmarshalMap(response.Item, &movie)
            if err != nil {
                log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
            } else {
                outMovies = append(outMovies, movie)
            }
        }
    }
    return outMovies, err
}
```

```
// GetAllMovies runs a PartiQL SELECT statement to get all movies from the
// DynamoDB table.
// pageSize is not typically required and is used to show how to paginate the
// results.
// The results are projected to return only the title and rating of each movie.
func (runner PartiQLRunner) GetAllMovies(pageSize int32)
([]map[string]interface{}, error) {
var output []map[string]interface{}
var response *dynamodb.ExecuteStatementOutput
var err error
var nextToken *string
for moreData := true; moreData; {
    response, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
&dynamodb.ExecuteStatementInput{
    Statement: aws.String(
        fmt.Sprintf("SELECT title, info.rating FROM \"%v\"", runner.TableName)),
    Limit:     aws.Int32(pageSize),
    NextToken: nextToken,
})
    if err != nil {
        log.Printf("Couldn't get movies. Here's why: %v\n", err)
        moreData = false
    } else {
        var pageOutput []map[string]interface{}
        err = attributevalue.UnmarshalListOfMaps(response.Items, &pageOutput)
        if err != nil {
            log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
        } else {
            log.Printf("Got a page of length %v.\n", len(response.Items))
            output = append(output, pageOutput...)
        }
        nextToken = response.NextToken
        moreData = nextToken != nil
    }
}
return output, err
}

// UpdateMovieBatch runs a batch of PartiQL UPDATE statements to update the
// rating of
// multiple movies that already exist in the DynamoDB table.
```

```
func (runner PartiQLRunner) UpdateMovieBatch(movies []Movie, ratings []float64) error {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{ratings[index],
            movie.Title, movie.Year})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(
                fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
                    runner.TableName)),
            Parameters: params,
        }
    }

    _, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
        &dynamodb.BatchExecuteStatementInput{
            Statements: statementRequests,
        })
    if err != nil {
        log.Printf("Couldn't update the batch of movies. Here's why: %v\n", err)
    }
    return err
}

// DeleteMovieBatch runs a batch of PartiQL DELETE statements to remove multiple
// movies
// from the DynamoDB table.
func (runner PartiQLRunner) DeleteMovieBatch(movies []Movie) error {
    statementRequests := make([]types.BatchStatementRequest, len(movies))
    for index, movie := range movies {
        params, err := attributevalue.MarshalList([]interface{}{movie.Title,
            movie.Year})
        if err != nil {
            panic(err)
        }
        statementRequests[index] = types.BatchStatementRequest{
            Statement: aws.String(
                fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
                    runner.TableName)),
    }
}
```

```
    Parameters: params,
}
}

_, err := runner.DynamoDbClient.BatchExecuteStatement(context.TODO(),
&dynamodb.BatchExecuteStatementInput{
    Statements: statementRequests,
})
if err != nil {
    log.Printf("Couldn't delete the batch of movies. Here's why: %v\n", err)
}
return err
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public class ScenarioPartiQLBatch {
    public static void main(String[] args) throws IOException {
        String tableName = "MoviesPartiQBatch";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println("***** Creating an Amazon DynamoDB table
named " + tableName
                    + " with a key named year and a sort key named
title.");
        createTable(ddb, tableName);
    }
}
```

```
        System.out.println("***** Adding multiple records into the " +
tableName
                + " table using a batch command.");
putRecordBatch(ddb);

        System.out.println("***** Updating multiple records using a
batch command.");
updateTableItemBatch(ddb);

        System.out.println("***** Deleting multiple records using a
batch command.");
deleteItemBatch(ddb);

        System.out.println("***** Deleting the Amazon DynamoDB
table.");
deleteDynamoDBTable(ddb, tableName);
ddb.close();
}

public static void createTable(DynamoDbClient ddb, String tableName) {
    DynamoDbWaiter dbWaiter = ddb.waiter();
    ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<>();

    // Define attributes.
    attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("year")
            .attributeType("N")
            .build());

    attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

    ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
    KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

    KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE) // Sort
```

```
        .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)

        .provisionedThroughput(ProvisionedThroughput.builder()
            .readCapacityUnits(new Long(10))
            .writeCapacityUnits(new Long(10))
            .build())
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

    try {
        CreateTableResponse response = ddb.createTable(request);
        DescribeTableRequest tableRequest =
        DescribeTableRequest.builder()
            .tableName(tableName)
            .build();

        // Wait until the Amazon DynamoDB table is created.
        WaiterResponse<DescribeTableResponse> waiterResponse =
        dbWaiter
            .waitUntilTableExists(tableRequest);

        waiterResponse.matched().response().ifPresent(System.out::println);
        String newTable =
        response.tableDescription().tableName();
        System.out.println("The " + newTable + " was successfully
created.");

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

public static void putRecordBatch(DynamoDbClient ddb) {
    String sqlStatement = "INSERT INTO MoviesPartiQBatch VALUE
{'year':?, 'title' : ?, 'info' : ?}";
```

```
try {
    // Create three movies to add to the Amazon DynamoDB
    table.

    // Set data for Movie 1.
    List<AttributeValue> parameters = new ArrayList<>();

    AttributeValue att1 = AttributeValue.builder()
        .n(String.valueOf("2022"))
        .build();

    AttributeValue att2 = AttributeValue.builder()
        .s("My Movie 1")
        .build();

    AttributeValue att3 = AttributeValue.builder()
        .s("No Information")
        .build();

    parameters.add(att1);
    parameters.add(att2);
    parameters.add(att3);

    BatchStatementRequest statementRequestMovie1 =
BatchStatementRequest.builder()
    .statement(sqlStatement)
    .parameters(parameters)
    .build();

    // Set data for Movie 2.
    List<AttributeValue> parametersMovie2 = new
ArrayList<>();
    AttributeValue attMovie2 = AttributeValue.builder()
        .n(String.valueOf("2022"))
        .build();

    AttributeValue attMovie2A = AttributeValue.builder()
        .s("My Movie 2")
        .build();

    AttributeValue attMovie2B = AttributeValue.builder()
        .s("No Information")
        .build();

    parametersMovie2.add(attMovie2);
```

```
parametersMovie2.add(attMovie2A);
parametersMovie2.add(attMovie2B);

BatchStatementRequest statementRequestMovie2 =
BatchStatementRequest.builder()
    .statement(sqlStatement)
    .parameters(parametersMovie2)
    .build();

// Set data for Movie 3.
List<AttributeValue> parametersMovie3 = new
ArrayList<>();
AttributeValue attMovie3 = AttributeValue.builder()
    .n(String.valueOf("2022"))
    .build();

AttributeValue attMovie3A = AttributeValue.builder()
    .s("My Movie 3")
    .build();

AttributeValue attMovie3B = AttributeValue.builder()
    .s("No Information")
    .build();

parametersMovie3.add(attMovie3);
parametersMovie3.add(attMovie3A);
parametersMovie3.add(attMovie3B);

BatchStatementRequest statementRequestMovie3 =
BatchStatementRequest.builder()
    .statement(sqlStatement)
    .parameters(parametersMovie3)
    .build();

// Add all three movies to the list.
List<BatchStatementRequest> myBatchStatementList = new
ArrayList<>();
myBatchStatementList.add(statementRequestMovie1);
myBatchStatementList.add(statementRequestMovie2);
myBatchStatementList.add(statementRequestMovie3);

BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
    .statements(myBatchStatementList)
```

```
        .build();

        BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchRequest);
        System.out.println("ExecuteStatement successful: " +
response.toString());
        System.out.println("Added new movies using a batch
command.");

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

public static void updateTableItemBatch(DynamoDbClient ddb) {
    String sqlStatement = "UPDATE MoviesPartiQBatch SET info =
'directors\":[\"Merian C. Cooper\",\"Ernest B. Schoedsack' where year=? and
title=?";
    List<AttributeValue> parametersRec1 = new ArrayList<>();

    // Update three records.
    AttributeValue att1 = AttributeValue.builder()
        .n(String.valueOf("2022"))
        .build();

    AttributeValue att2 = AttributeValue.builder()
        .s("My Movie 1")
        .build();

    parametersRec1.add(att1);
    parametersRec1.add(att2);

    BatchStatementRequest statementRequestRec1 =
BatchStatementRequest.builder()
        .statement(sqlStatement)
        .parameters(parametersRec1)
        .build();

    // Update record 2.
    List<AttributeValue> parametersRec2 = new ArrayList<>();
    AttributeValue attRec2 = AttributeValue.builder()
        .n(String.valueOf("2022"))
        .build();
```

```
AttributeValue attRec2a = AttributeValue.builder()
    .s("My Movie 2")
    .build();

parametersRec2.add(attRec2);
parametersRec2.add(attRec2a);
BatchStatementRequest statementRequestRec2 =
BatchStatementRequest.builder()
    .statement(sqlStatement)
    .parameters(parametersRec2)
    .build();

// Update record 3.
List<AttributeValue> parametersRec3 = new ArrayList<>();
AttributeValue attRec3 = AttributeValue.builder()
    .n(String.valueOf("2022"))
    .build();

AttributeValue attRec3a = AttributeValue.builder()
    .s("My Movie 3")
    .build();

parametersRec3.add(attRec3);
parametersRec3.add(attRec3a);
BatchStatementRequest statementRequestRec3 =
BatchStatementRequest.builder()
    .statement(sqlStatement)
    .parameters(parametersRec3)
    .build();

// Add all three movies to the list.
List<BatchStatementRequest> myBatchStatementList = new
ArrayList<>();
myBatchStatementList.add(statementRequestRec1);
myBatchStatementList.add(statementRequestRec2);
myBatchStatementList.add(statementRequestRec3);

BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
    .statements(myBatchStatementList)
    .build();

try {
```

```
        BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchRequest);
        System.out.println("ExecuteStatement successful: " +
response.toString());
        System.out.println("Updated three movies using a batch
command.");
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    System.out.println("Item was updated!");
}

public static void deleteItemBatch(DynamoDbClient ddb) {
    String sqlStatement = "DELETE FROM MoviesPartiQBatch WHERE year
= ? and title=?";
    List<AttributeValue> parametersRec1 = new ArrayList<>();

    // Specify three records to delete.
    AttributeValue att1 = AttributeValue.builder()
        .n(String.valueOf("2022"))
        .build();

    AttributeValue att2 = AttributeValue.builder()
        .s("My Movie 1")
        .build();

    parametersRec1.add(att1);
    parametersRec1.add(att2);

    BatchStatementRequest statementRequestRec1 =
BatchStatementRequest.builder()
        .statement(sqlStatement)
        .parameters(parametersRec1)
        .build();

    // Specify record 2.
    List<AttributeValue> parametersRec2 = new ArrayList<>();
    AttributeValue attRec2 = AttributeValue.builder()
        .n(String.valueOf("2022"))
        .build();

    AttributeValue attRec2a = AttributeValue.builder()
```

```
        .s("My Movie 2")
        .build();

    parametersRec2.add(attRec2);
    parametersRec2.add(attRec2a);
    BatchStatementRequest statementRequestRec2 =
BatchStatementRequest.builder()
                    .statement(sqlStatement)
                    .parameters(parametersRec2)
                    .build();

    // Specify record 3.
    List<AttributeValue> parametersRec3 = new ArrayList<>();
    AttributeValue attRec3 = AttributeValue.builder()
                    .n(String.valueOf("2022"))
                    .build();

    AttributeValue attRec3a = AttributeValue.builder()
                    .s("My Movie 3")
                    .build();

    parametersRec3.add(attRec3);
    parametersRec3.add(attRec3a);

    BatchStatementRequest statementRequestRec3 =
BatchStatementRequest.builder()
                    .statement(sqlStatement)
                    .parameters(parametersRec3)
                    .build();

    // Add all three movies to the list.
    List<BatchStatementRequest> myBatchStatementList = new
ArrayList<>();
    myBatchStatementList.add(statementRequestRec1);
    myBatchStatementList.add(statementRequestRec2);
    myBatchStatementList.add(statementRequestRec3);

    BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
                    .statements(myBatchStatementList)
                    .build();

    try {
        ddb.batchExecuteStatement(batchRequest);
```

```
        System.out.println("Deleted three movies using a batch  
command.");  
  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    }  
}  
  
public static void deleteDynamoDBTable(DynamoDbClient ddb, String  
tableName) {  
    DeleteTableRequest request = DeleteTableRequest.builder()  
        .tableName(tableName)  
        .build();  
  
    try {  
        ddb.deleteTable(request);  
  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    }  
    System.out.println(tableName + " was successfully deleted!");  
}  
  
private static ExecuteStatementResponse  
executeStatementRequest(DynamoDbClient ddb, String statement,  
    List<AttributeValue> parameters) {  
    ExecuteStatementRequest request =  
    ExecuteStatementRequest.builder()  
        .statement(statement)  
        .parameters(parameters)  
        .build();  
  
    return ddb.executeStatement(request);  
}  
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Execute batch PartiQL statements.

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "Cities";

export const main = async () => {
  /**
   * Create a table.
   */

  log("Creating a table.");
  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
  });
}
```

```
AttributeDefinitions: [
  {
    AttributeName: "name",
    // 'S' is a data type descriptor that represents a number type.
    // For a list of all data type descriptors, see the following link.
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeType: "S",
  },
],
// The KeySchema defines the primary key. The primary key can be
// a partition key, or a combination of a partition key and a sort key.
// Key schema design is important. For more info, see
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
KeySchema: [{ AttributeName: "name", KeyType: "HASH" }],
});
await client.send(createTableCommand);
log(`Table created: ${tableName}.`);

/**
 * Wait until the table is active.
 */

// This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
// You can't write to a table before it's active.
log("Waiting for the table to be active.");
await waitUntilTableExists({ client }, { TableName: tableName });
log("Table active.");

/**
 * Insert items.
 */

log("Inserting cities into the table.");
const addItemsStatementCommand = new BatchExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
  Statements: [
    {
      Statement: `INSERT INTO ${tableName} value {'name':?:, 'population':?}`,
      Parameters: ["Alachua", 10712],
    },
    {

```

```
Statement: `INSERT INTO ${tableName} value {'name':?:, 'population':?}`,
Parameters: ["High Springs", 6415],
},
],
});
await docClient.send(addItemsStatementCommand);
log(`Cities inserted.`);

/***
 * Select items.
 */
log("Selecting cities from the table.");
const selectItemsStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-reference.select.html
    Statements: [
        {
            Statement: `SELECT * FROM ${tableName} WHERE name=?`,
            Parameters: ["Alachua"],
        },
        {
            Statement: `SELECT * FROM ${tableName} WHERE name=?`,
            Parameters: ["High Springs"],
        },
    ],
});
const selectItemResponse = await docClient.send(selectItemsStatementCommand);
log(
    `Got cities: ${selectItemResponse.Responses.map(
        (r) => `${r.Item.name} (${r.Item.population})`,
    ).join(", ")}`,
);

/***
 * Update items.
 */
log("Modifying the populations.");
const updateItemStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-reference.update.html
    Statements: [
        {
```

```
Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
Parameters: [10, "Alachua"],
},
{
  Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
  Parameters: [5, "High Springs"],
},
],
});
await docClient.send(updateItemStatementCommand);
log(`Updated cities.`);

/**
 * Delete the items.
 */

log("Deleting the cities.");
const deleteItemStatementCommand = new BatchExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-reference.delete.html
  Statements: [
    {
      Statement: `DELETE FROM ${tableName} WHERE name=?`,
      Parameters: ["Alachua"],
    },
    {
      Statement: `DELETE FROM ${tableName} WHERE name=?`,
      Parameters: ["High Springs"],
    },
  ],
});
await docClient.send(deleteItemStatementCommand);
log("Cities deleted.");

/**
 * Delete the table.
 */

log("Deleting the table.");
const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
await client.send(deleteTableCommand);
log("Table deleted.");
};
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun main() {  
    val ddb = DynamoDbClient { region = "us-east-1" }  
    val tableName = "MoviesPartiQLBatch"  
    println("Creating an Amazon DynamoDB table named $tableName with a key named id and a sort key named title.")  
    createTablePartiQLBatch(ddb, tableName, "year")  
    putRecordBatch(ddb)  
    updateTableItemBatchBatch(ddb)  
    deleteItemsBatch(ddb)  
    deleteTablePartiQLBatch(tableName)  
}  
  
suspend fun createTablePartiQLBatch(ddb: DynamoDbClient, tableNameVal: String,  
key: String) {  
    val attDef = AttributeDefinition {  
        attributeName = key  
        attributeType = ScalarAttributeType.N  
    }  
  
    val attDef1 = AttributeDefinition {  
        attributeName = "title"  
        attributeType = ScalarAttributeType.S  
    }  
  
    val keySchemaVal = KeySchemaElement {  
        attributeName = key  
        keyType = KeyType.Hash  
    }
```

```
val keySchemaVal1 = KeySchemaElement {  
    attributeName = "title"  
    keyType = KeyType.Range  
}  
  
val provisionedVal = ProvisionedThroughput {  
    readCapacityUnits = 10  
    writeCapacityUnits = 10  
}  
  
val request = CreateTableRequest {  
    attributeDefinitions = listOf(attDef, attDef1)  
    keySchema = listOf(keySchemaVal, keySchemaVal1)  
    provisionedThroughput = provisionedVal  
    tableName = tableNameVal  
}  
  
val response = ddb.createTable(request)  
ddb.waitUntilTableExists { // suspend call  
    tableName = tableNameVal  
}  
    println("The table was successfully created  
${response.tableDescription?.tableArn}")  
}  
  
suspend fun putRecordBatch(ddb: DynamoDbClient) {  
    val sqlStatement = "INSERT INTO MoviesPartiQBatch VALUE {'year':?,  
'title' : ?, 'info' : ?}"  
  
    // Create three movies to add to the Amazon DynamoDB table.  
    val parametersMovie1 = mutableListOf<AttributeValue>()  
    parametersMovie1.add(AttributeValue.N("2022"))  
    parametersMovie1.add(AttributeValue.S("My Movie 1"))  
    parametersMovie1.add(AttributeValue.S("No Information"))  
  
    val statementRequestMovie1 = BatchStatementRequest {  
        statement = sqlStatement  
        parameters = parametersMovie1  
    }  
  
    // Set data for Movie 2.  
    val parametersMovie2 = mutableListOf<AttributeValue>()  
    parametersMovie2.add(AttributeValue.N("2022"))
```

```
parametersMovie2.add(AttributeValue.S("My Movie 2"))
parametersMovie2.add(AttributeValue.S("No Information"))

val statementRequestMovie2 = BatchStatementRequest {
    statement = sqlStatement
    parameters = parametersMovie2
}

// Set data for Movie 3.
val parametersMovie3 = mutableListOf<AttributeValue>()
parametersMovie3.add(AttributeValue.N("2022"))
parametersMovie3.add(AttributeValue.S("My Movie 3"))
parametersMovie3.add(AttributeValue.S("No Information"))

val statementRequestMovie3 = BatchStatementRequest {
    statement = sqlStatement
    parameters = parametersMovie3
}

// Add all three movies to the list.
val myBatchStatementList = mutableListOf<BatchStatementRequest>()
myBatchStatementList.add(statementRequestMovie1)
myBatchStatementList.add(statementRequestMovie2)
myBatchStatementList.add(statementRequestMovie3)

val batchRequest = BatchExecuteStatementRequest {
    statements = myBatchStatementList
}
val response = ddb.batchExecuteStatement(batchRequest)
println("ExecuteStatement successful: " + response.toString())
println("Added new movies using a batch command.")
}

suspend fun updateTableItemBatchBatch(ddb: DynamoDbClient) {
    val sqlStatement =
        "UPDATE MoviesPartiQBatch SET info = 'directors\":[\"Merian C. Cooper\",
\"Ernest B. Schoedsack\" where year=? and title=?"
    val parametersRec1 = mutableListOf<AttributeValue>()
    parametersRec1.add(AttributeValue.N("2022"))
    parametersRec1.add(AttributeValue.S("My Movie 1"))
    val statementRequestRec1 = BatchStatementRequest {
        statement = sqlStatement
        parameters = parametersRec1
    }
}
```

```
// Update record 2.
val parametersRec2 = mutableListOf<AttributeValue>()
parametersRec2.add(AttributeValue.N("2022"))
parametersRec2.add(AttributeValue.S("My Movie 2"))
val statementRequestRec2 = BatchStatementRequest {
    statement = sqlStatement
    parameters = parametersRec2
}

// Update record 3.
val parametersRec3 = mutableListOf<AttributeValue>()
parametersRec3.add(AttributeValue.N("2022"))
parametersRec3.add(AttributeValue.S("My Movie 3"))
val statementRequestRec3 = BatchStatementRequest {
    statement = sqlStatement
    parameters = parametersRec3
}

// Add all three movies to the list.
val myBatchStatementList = mutableListOf<BatchStatementRequest>()
myBatchStatementList.add(statementRequestRec1)
myBatchStatementList.add(statementRequestRec2)
myBatchStatementList.add(statementRequestRec3)

val batchRequest = BatchExecuteStatementRequest {
    statements = myBatchStatementList
}

val response = ddb.batchExecuteStatement(batchRequest)
println("ExecuteStatement successful: $response")
println("Updated three movies using a batch command.")
println("Items were updated!")
}

suspend fun deleteItemsBatch(ddb: DynamoDbClient) {
    // Specify three records to delete.
    val sqlStatement = "DELETE FROM MoviesPartiQBatch WHERE year = ? and title=?"
    val parametersRec1 = mutableListOf<AttributeValue>()
    parametersRec1.add(AttributeValue.N("2022"))
    parametersRec1.add(AttributeValue.S("My Movie 1"))

    val statementRequestRec1 = BatchStatementRequest {
        statement = sqlStatement
    }
}
```

```
    parameters = parametersRec1
}

// Specify record 2.
val parametersRec2 = mutableListOf<AttributeValue>()
parametersRec2.add(AttributeValue.N("2022"))
parametersRec2.add(AttributeValue.S("My Movie 2"))
val statementRequestRec2 = BatchStatementRequest {
    statement = sqlStatement
    parameters = parametersRec2
}

// Specify record 3.
val parametersRec3 = mutableListOf<AttributeValue>()
parametersRec3.add(AttributeValue.N("2022"))
parametersRec3.add(AttributeValue.S("My Movie 3"))
val statementRequestRec3 = BatchStatementRequest {
    statement = sqlStatement
    parameters = parametersRec3
}

// Add all three movies to the list.
val myBatchStatementList = mutableListOf<BatchStatementRequest>()
myBatchStatementList.add(statementRequestRec1)
myBatchStatementList.add(statementRequestRec2)
myBatchStatementList.add(statementRequestRec3)

val batchRequest = BatchExecuteStatementRequest {
    statements = myBatchStatementList
}

ddb.batchExecuteStatement(batchRequest)
println("Deleted three movies using a batch command.")
}

suspend fun deleteTablePartiQLBatch(tableNameVal: String) {
    val request = DeleteTableRequest {
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}
```

```
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
namespace DynamoDb\PartiQL_Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;

use function AwsUtilities\loadMovieData;
use function AwsUtilities\testable_readline;

class GettingStartedWithPartiQLBatch
{
    public function run()
    {
        echo("\n");
        echo("-----\n");
        print("Welcome to the Amazon DynamoDB - PartiQL getting started demo
using PHP!\n");
        echo("-----\n");

        $uuid = uniqid();
        $service = new DynamoDb\DynamoDBService();

        $tableName = "partiql_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
```

```
        new DynamoDBAttribute('title', 'S', 'RANGE')
    ]
);

echo "Waiting for table...";
$service->dynamoDbClient->waitForTable("TableExists", ['TableName' =>
$tableName]);
echo "table $tableName found!\n";

echo "What's the name of the last movie you watched?\n";
while (empty($movieName)) {
    $movieName = testable_readline("Movie name: ");
}
echo "And what year was it released?\n";
$movieYear = "year";
while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
    $movieYear = testable_readline("Year released: ");
}
$key = [
    'Item' => [
        'year' => [
            'N' => "$movieYear",
        ],
        'title' => [
            'S' => $movieName,
        ],
    ],
];
list($statement, $parameters) = $service-
>buildStatementAndParameters("INSERT", $tableName, $key);
$service->insertItemByPartiQLBatch($statement, $parameters);

echo "How would you rate the movie from 1-10?\n";
$rating = 0;
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
    $rating = testable_readline("Rating (1-10): ");
}
echo "What was the movie about?\n";
while (empty($plot)) {
    $plot = testable_readline("Plot summary: ");
}
$attributes = [
    new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
```

```
        new DynamoDBAttribute('plot', 'S', 'RANGE', $plot),
    ];

    list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
    $service->updateItemByPartiQLBatch($statement, $parameters);
    echo "Movie added and updated.\n";

    $batch = json_decode(loadMovieData());

    $service->writeBatch($tableName, $batch);

    $movie = $service->getItemByPartiQLBatch($tableName, [$key]);
    echo "\nThe movie {$movie['Responses'][0]['Item']['title']]['S']"
    was released in {$movie['Responses'][0]['Item']['year']]['N'].\n";
    echo "What rating would you like to give {$movie['Responses'][0]['Item']
['title']]['S']}?\n";
    $rating = 0;
    while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
        $rating = testable_readline("Rating (1-10): ");
    }
    $attributes = [
        new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
        new DynamoDBAttribute('plot', 'S', 'RANGE', $plot)
    ];
    list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
    $service->updateItemByPartiQLBatch($statement, $parameters);

    $movie = $service->getItemByPartiQLBatch($tableName, [$key]);
    echo "Okay, you have rated {$movie['Responses'][0]['Item']['title']}
['S']"
    as a {$movie['Responses'][0]['Item']['rating']]['N']\n";

    $service->deleteItemByPartiQLBatch($statement, $parameters);
    echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";

    echo "That's okay though. The book was better. Now, for something
lighter, in what year were you born?\n";
    $birthYear = "not a number";
    while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
        $birthYear = testable_readline("Birth year: ");
```

```
        }
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were born:\n";
        $oops = "Oops! There were no movies released in that year (that we know of).\n";
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;

        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}
```

```
public function insertItemByPartiQLBatch(string $statement, array $parameters)
{
    $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => [
            [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ],
        ],
    ]);
}

public function getItemByPartiQLBatch(string $tableName, array $keys): Result
{
    $statements = [];
    foreach ($keys as $key) {
        list($statement, $parameters) = $this->buildStatementAndParameters("SELECT", $tableName, $key['Item']);
        $statements[] = [
            'Statement' => "$statement",
            'Parameters' => $parameters,
        ];
    }

    return $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => $statements,
    ]);
}

public function updateItemByPartiQLBatch(string $statement, array $parameters)
{
    $this->dynamoDbClient->batchExecuteStatement([
        'Statements' => [
            [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ],
        ],
    ]);
}
```

```
public function deleteItemByPartiQLBatch(string $statement, array  
$parameters)  
{  
    $this->dynamoDbClient->batchExecuteStatement([  
        'Statements' => [  
            [  
                'Statement' => "$statement",  
                'Parameters' => $parameters,  
            ],  
            [],  
        ]);  
}  
}
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a class that can run batches of PartiQL statements.

```
from datetime import datetime  
from decimal import Decimal  
import logging  
from pprint import pprint  
  
import boto3  
from botocore.exceptions import ClientError  
  
from scaffold import Scaffold  
  
logger = logging.getLogger(__name__)  
  
class PartiQLBatchWrapper:  
    """
```

```
Encapsulates a DynamoDB resource to run PartiQL statements.  
"""\n\n    def __init__(self, dyn_resource):\n        """\n            :param dyn_resource: A Boto3 DynamoDB resource.\n        """\n\n            self.dyn_resource = dyn_resource\n\n\n    def run_partiql(self, statements, param_list):\n        """\n            Runs a PartiQL statement. A Boto3 resource is used even though\n            `execute_statement` is called on the underlying `client` object because\n            the\n                resource transforms input and output from plain old Python objects\n            (POPOs) to\n                the DynamoDB format. If you create the client directly, you must do these\n            transforms yourself.\n\n            :param statements: The batch of PartiQL statements.\n            :param param_list: The batch of PartiQL parameters that are associated\n            with\n                each statement. This list must be in the same order as\n            the\n                statements.\n            :return: The responses returned from running the statements, if any.\n        """\n\n        try:\n            output = self.dyn_resource.meta.client.batch_execute_statement(\n                Statements=[\n                    {"Statement": statement, "Parameters": params}\n                    for statement, params in zip(statements, param_list)\n                ]\n            )\n        except ClientError as err:\n            if err.response["Error"]["Code"] == "ResourceNotFoundException":\n                logger.error(\n                    "Couldn't execute batch of PartiQL statements because the\n                    table \"\n                        \"does not exist.\"\n                )\n            else:\n                logger.error(\n
```

```
        "Couldn't execute batch of PartiQL statements. Here's why:  
%s: %s",  
        err.response["Error"]["Code"],  
        err.response["Error"]["Message"],  
    )  
    raise  
else:  
    return output
```

Run a scenario that creates a table and runs PartiQL queries in batches.

```
def run_scenario(scaffold, wrapper, table_name):  
    logging.basicConfig(level=logging.INFO, format"%(levelname)s: %(message)s")  
  
    print("-" * 88)  
    print("Welcome to the Amazon DynamoDB PartiQL batch statement demo.")  
    print("-" * 88)  
  
    print(f"Creating table '{table_name}' for the demo...")  
    scaffold.create_table(table_name)  
    print("-" * 88)  
  
    movie_data = [  
        {  
            "title": f"House PartiQL",  
            "year": datetime.now().year - 5,  
            "info": {  
                "plot": "Wacky high jinks result from querying a mysterious  
database.",  
                "rating": Decimal("8.5"),  
            },  
        },  
        {  
            "title": f"House PartiQL 2",  
            "year": datetime.now().year - 3,  
            "info": {  
                "plot": "Moderate high jinks result from querying another  
mysterious database.",  
                "rating": Decimal("6.5"),  
            },  
        },  
    ]
```

```
        },
    ],
    {
        "title": f"House PartiQL 3",
        "year": datetime.now().year - 1,
        "info": {
            "plot": "Tepid high jinks result from querying yet another
mysterious database.",
            "rating": Decimal("2.5"),
        },
    },
]
]

print(f"Inserting a batch of movies into table '{table_name}'")
statements = [
    f'INSERT INTO "{table_name}" ' f"VALUE {{'title': ?, 'year': ?,
'info': ?}}"
] * len(movie_data)
params = [list(movie.values()) for movie in movie_data]
wrapper.run_partiql(statements, params)
print("Success!")
print("-" * 88)

print(f"Getting data for a batch of movies.")
statements = [f'SELECT * FROM "{table_name}" WHERE title=? AND year=?'] *
len(
    movie_data
)
params = [[movie["title"], movie["year"]] for movie in movie_data]
output = wrapper.run_partiql(statements, params)
for item in output["Responses"]:
    print(f"\n{item['Item']['title']}, {item['Item']['year']}")
    pprint(item["Item"])
print("-" * 88)

ratings = [Decimal("7.7"), Decimal("5.5"), Decimal("1.3")]
print(f"Updating a batch of movies with new ratings.")
statements = [
    f'UPDATE "{table_name}" SET info.rating=? ' f"WHERE title=? AND year=?"
] * len(movie_data)
params = [
    [rating, movie["title"], movie["year"]]
    for rating, movie in zip(ratings, movie_data)
]
```

```
wrapper.run_partiql(statements, params)
print("Success!")
print("-" * 88)

print(f"Getting projected data from the table to verify our update.")
output = wrapper.dyn_resource.meta.client.execute_statement(
    Statement=f'SELECT title, info.rating FROM "{table_name}"'
)
pprint(output["Items"])
print("-" * 88)

print(f"Deleting a batch of movies from the table.")
statements = [f'DELETE FROM "{table_name}" WHERE title=? AND year=?'] * len(
    movie_data
)
params = [[movie["title"], movie["year"]] for movie in movie_data]
wrapper.run_partiql(statements, params)
print("Success!")
print("-" * 88)

print(f"Deleting table '{table_name}'...")
scaffold.delete_table()
print("-" * 88)

print("\nThanks for watching!")
print("-" * 88)

if __name__ == "__main__":
    try:
        dyn_res = boto3.resource("dynamodb")
        scaffold = Scaffold(dyn_res)
        movies = PartiQLBatchWrapper(dyn_res)
        run_scenario(scaffold, movies, "doc-example-table-partiql-movies")
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run a scenario that creates a table and runs batch PartiQL queries.

```
table_name = "doc-example-table-movies-partiql-#{rand(10**4)}"
scaffold = Scaffold.new(table_name)
sdk = DynamoDBPartiQLBatch.new(table_name)

new_step(1, "Create a new DynamoDB table if none already exists.")
unless scaffold.exists?(table_name)
    puts("\nNo such table: #{table_name}. Creating it...")
    scaffold.create_table(table_name)
    print "Done!\n".green
end

new_step(2, "Populate DynamoDB table with movie data.")
download_file = "moviedata.json"
puts("Downloading movie database to #{download_file}...")
movie_data = scaffold.fetch_movie_data(download_file)
puts("Writing movie data from #{download_file} into your table...")
scaffold.write_batch(movie_data)
puts("Records added: #{movie_data.length}.")
print "Done!\n".green

new_step(3, "Select a batch of items from the movies table.")
puts "Let's select some popular movies for side-by-side comparison."
response = sdk.batch_execute_select([["Mean Girls", 2004], ["Goodfellas", 1977], ["The Prancing of the Lambs", 2005]])
puts("Items selected: #{response['responses'].length}\n")
print "\nDone!\n".green

new_step(4, "Delete a batch of items from the movies table.")
sdk.batch_execute_write([["Mean Girls", 2004], ["Goodfellas", 1977], ["The Prancing of the Lambs", 2005]])
print "\nDone!\n".green
```

```
new_step(5, "Delete the table.")
if scaffold.exists?(table_name)
  scaffold.delete_table
end
end
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table using PartiQL and an AWS SDK

The following code examples show how to:

- Get an item by running a SELECT statement.
- Add an item by running an INSERT statement.
- Update an item by running an UPDATE statement.
- Delete an item by running a DELETE statement.

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
namespace PartiQL_Basics_Scenario
{
    public class PartiQLMethods
    {
        private static readonly AmazonDynamoDBClient Client = new
            AmazonDynamoDBClient();
```

```
/// <summary>
/// Inserts movies imported from a JSON file into the movie table by
/// using an Amazon DynamoDB PartiQL INSERT statement.
/// </summary>
/// <param name="tableName">The name of the table where the movie
/// information will be inserted.</param>
/// <param name="movieFileName">The name of the JSON file that contains
/// movie information.</param>
/// <returns>A Boolean value that indicates the success or failure of
/// the insert operation.</returns>
public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
{
    // Get the list of movies from the JSON file.
    var movies = ImportMovies(movieFileName);

    var success = false;

    if (movies is not null)
    {
        // Insert the movies in a batch using PartiQL. Because the
        // batch can contain a maximum of 25 items, insert 25 movies
        // at a time.
        string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
        var statements = new List<BatchStatementRequest>();

        try
        {
            for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
            {
                for (var i = indexOffset; i < indexOffset + 25; i++)
                {
                    statements.Add(new BatchStatementRequest
                    {
                        Statement = insertBatch,
                        Parameters = new List<AttributeValue>
                        {
                            new AttributeValue { S = movies[i].Title },
                            new AttributeValue { N =
movies[i].Year.ToString() },
                        }
                    });
                }
            }
        }
    }
}
```

```
        },
    });
}

var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
{
    Statements = statements,
});

// Wait between batches for movies to be successfully
added.
System.Threading.Thread.Sleep(3000);

success = response.StatusCode ==
System.Net.HttpStatusCode.OK;

// Clear the list of statements for the next batch.
statements.Clear();
}
}
catch (AmazonDynamoDBException ex)
{
    Console.WriteLine(ex.Message);
}
}

return success;
}

/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The full path to the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public static List<Movie> ImportMovies(string movieFileName)
{
    if (!File.Exists(movieFileName))
    {
        return null!;
    }

    using var sr = new StreamReader(movieFileName);
```

```
        string json = sr.ReadToEnd();
        var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

        if (allMovies is not null)
        {
            // Return the first 250 entries.
            return allMovies.GetRange(0, 250);
        }
        else
        {
            return null!;
        }
    }

    /// <summary>
    /// Uses a PartiQL SELECT statement to retrieve a single movie from the
    /// movie database.
    /// </summary>
    /// <param name="tableName">The name of the movie table.</param>
    /// <param name="movieTitle">The title of the movie to retrieve.</param>
    /// <returns>A list of movie data. If no movie matches the supplied
    /// title, the list is empty.</returns>
    public static async Task<List<Dictionary<string,AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
{
    string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
    var parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
    };

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = selectSingle,
    Parameters = parameters,
});

    return response.Items;
}
```

```
    /// <summary>
    /// Retrieve multiple movies by year using a SELECT statement.
    /// </summary>
    /// <param name="tableName">The name of the movie table.</param>
    /// <param name="year">The year the movies were released.</param>
    /// <returns></returns>
    public static async Task<List<Dictionary<string, AttributeValue>>>
GetMovies(string tableName, int year)
{
    string selectSingle = $"SELECT * FROM {tableName} WHERE year = ?";
    var parameters = new List<AttributeValue>
    {
        new AttributeValue { N = year.ToString() },
    };

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
    {
        Statement = selectSingle,
        Parameters = parameters,
    });

    return response.Items;
}

    /// <summary>
    /// Inserts a single movie into the movies table.
    /// </summary>
    /// <param name="tableName">The name of the table.</param>
    /// <param name="movieTitle">The title of the movie to insert.</param>
    /// <param name="year">The year that the movie was released.</param>
    /// <returns>A Boolean value that indicates the success or failure of
    /// the INSERT operation.</returns>
    public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
{
    string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?, 'year': ?}}";
    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
    {
```

```
        Statement = insertBatch,
        Parameters = new List<AttributeValue>
        {
            new AttributeValue { S = movieTitle },
            new AttributeValue { N = year.ToString() },
        },
    });

    return response.StatusCode == System.Net.HttpStatusCode.OK;
}

/// <summary>
/// Updates a single movie in the table, adding information for the
/// producer.
/// </summary>
/// <param name="tableName">the name of the table.</param>
/// <param name="producer">The name of the producer.</param>
/// <param name="movieTitle">The movie title.</param>
/// <param name="year">The year the movie was released.</param>
/// <returns>A Boolean value that indicates the success of the
/// UPDATE operation.</returns>
public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
{
    string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = insertSingle,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = producer },
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});

    return response.StatusCode == System.Net.HttpStatusCode.OK;
}
```

```
/// <summary>
/// Deletes a single movie from the table.
/// </summary>
/// <param name="tableName">The name of the table.</param>
/// <param name="movieTitle">The title of the movie to delete.</param>
/// <param name="year">The year that the movie was released.</param>
/// <returns>A Boolean value that indicates the success of the
/// DELETE operation.</returns>
public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
{
    var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = deleteSingle,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});

    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}

/// <summary>
/// Displays the list of movies returned from a database query.
/// </summary>
/// <param name="items">The list of movie information to display.</param>
private static void DisplayMovies(List<Dictionary<string,
AttributeValue>> items)
{
    if (items.Count > 0)
    {
        Console.WriteLine($"Found {items.Count} movies.");
        items.ForEach(item =>
Console.WriteLine($"{item["year"].N}\t{item["title"].S}"));
    }
}
```

```
        else
        {
            Console.WriteLine($"Didn't find a movie that matched the supplied
criteria.");
        }
    }

}

/// <summary>
/// Uses a PartiQL SELECT statement to retrieve a single movie from the
/// movie database.
/// </summary>
/// <param name="tableName">The name of the movie table.</param>
/// <param name="movieTitle">The title of the movie to retrieve.</param>
/// <returns>A list of movie data. If no movie matches the supplied
/// title, the list is empty.</returns>
public static async Task<List<Dictionary<string, AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
{
    string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
    var parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
    };

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
    {
        Statement = selectSingle,
        Parameters = parameters,
    });
}

return response.Items;
}

/// <summary>
/// Inserts a single movie into the movies table.

```

```
/// </summary>
/// <param name="tableName">The name of the table.</param>
/// <param name="movieTitle">The title of the movie to insert.</param>
/// <param name="year">The year that the movie was released.</param>
/// <returns>A Boolean value that indicates the success or failure of
/// the INSERT operation.</returns>
public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
{
    string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?, 'year': ?}}";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = insertBatch,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});
}

return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

```
/// <summary>
/// Updates a single movie in the table, adding information for the
/// producer.
/// </summary>
/// <param name="tableName">the name of the table.</param>
/// <param name="producer">The name of the producer.</param>
/// <param name="movieTitle">The movie title.</param>
/// <param name="year">The year the movie was released.</param>
/// <returns>A Boolean value that indicates the success of the
/// UPDATE operation.</returns>
public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
{
    string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";
```

```
        var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = insertSingle,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = producer },
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

/// <summary>
/// Deletes a single movie from the table.
/// </summary>
/// <param name="tableName">The name of the table.</param>
/// <param name="movieTitle">The title of the movie to delete.</param>
/// <param name="year">The year that the movie was released.</param>
/// <returns>A Boolean value that indicates the success of the
/// DELETE operation.</returns>
public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
{
    var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

    var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
{
    Statement = deleteSingle,
    Parameters = new List<AttributeValue>
    {
        new AttributeValue { S = movieTitle },
        new AttributeValue { N = year.ToString() },
    },
});

    return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 1. Create a table. (CreateTable)
if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

    AwsDoc::DynamoDB::partiqlExecuteScenario(clientConfig);

    // 7. Delete the table. (DeleteTable)
    AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
}

//! Scenario to modify and query a DynamoDB table using single PartiQL
// statements.
/*
\sa partiqlExecuteScenario()
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/
bool
AwsDoc::DynamoDB::partiqlExecuteScenario(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie using an "Insert" statement. (ExecuteStatement)
    Aws::String title;
    float rating;
    int year;
    Aws::String plot;
{
```

```
title = askQuestion(
    "Enter the title of a movie you want to add to the table: ");
year = askQuestionForInt("What year was it released? ");
rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                1, 10);
plot = askQuestion("Summarize the plot for me: ");

Aws::DynamoDB::Model::ExecuteStatementRequest request;
std::stringstream sqlStream;
sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {""
<< TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '""
<< INFO_KEY << "': ?}";

request.SetStatement(sqlStream.str());

// Create the parameter attributes.
Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
    ALLOCATION_TAG.c_str());
ratingAttribute->SetN(rating);
infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
    ALLOCATION_TAG.c_str());
plotAttribute->SetS(plot);
infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
attributes.push_back(infoMapAttribute);
request.SetParameters(attributes);

Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
    request);

if (!outcome.IsSuccess()) {
    std::cerr << "Failed to add a movie: " <<
outcome.GetError().GetMessage()
```

```
        << std::endl;
    return false;
}

std::cout << "\nAdded '" << title << "' to '" << MOVIE_TABLE_NAME << "'."
    << std::endl;

// 3. Get the data for the movie using a "Select" statement.
(ExecuteStatement)
{
    Aws::DynamoDB::Model::ExecuteStatementRequest request;
    std::stringstream sqlStream;
    sqlStream << "SELECT * FROM \\" << MOVIE_TABLE_NAME << "\\ WHERE "
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";

    request.SetStatement(sqlStream.str());

    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
    request.SetParameters(attributes);

    Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
    request);

    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to retrieve movie information: "
            << outcome.GetError().GetMessage() << std::endl;
        return false;
    }
    else {
        // Print the retrieved movie information.
        const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

        const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

        if (items.size() == 1) {
            printMovieInfo(items[0]);
        }
        else {
```

```
        std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                           << " There should be only one movie." << std::endl;
    }
}
}

// 4. Update the data for the movie using an "Update" statement.
(ExecuteStatement)
{
    rating = askQuestionForFloatRange(
        Aws::String("\nLet's update your movie.\nYou rated it  ") +
        std::to_string(rating)
        + ", what new rating would you give it? ", 1, 10);

    Aws::DynamoDB::Model::ExecuteStatementRequest request;
    std::stringstream sqlStream;
    sqlStream << "UPDATE \""
    << MOVIE_TABLE_NAME << "\" SET "
        << INFO_KEY << "."
        << RATING_KEY << "=? WHERE "
        << TITLE_KEY << "=?"
        << AND "
        << YEAR_KEY << "=?";

    request.SetStatement(sqlStream.str());

    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;

    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(rating));
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

    request.SetParameters(attributes);

    Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
    request);

    if (!outcome.IsSuccess()) {
        std::cerr << "Failed to update a movie: "
                           << outcome.GetError().GetMessage();
        return false;
    }
}

std::cout << "\nUpdated '" << title << "' with new attributes:" << std::endl;
```

```
// 5. Get the updated data for the movie using a "Select" statement.  
(ExecuteStatement)  
{  
    Aws::DynamoDB::Model::ExecuteStatementRequest request;  
    std::stringstream sqlStream;  
    sqlStream << "SELECT * FROM \\" << MOVIE_TABLE_NAME << "\\ WHERE "  
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";  
  
    request.SetStatement(sqlStream.str());  
  
    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;  
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().Sets(title));  
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));  
    request.SetParameters(attributes);  
  
    Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =  
    dynamoClient.ExecuteStatement(  
        request);  
    if (!outcome.IsSuccess()) {  
        std::cerr << "Failed to retrieve the movie information: "  
            << outcome.GetError().GetMessage() << std::endl;  
        return false;  
    }  
    else {  
        const Aws::DynamoDB::Model::ExecuteStatementResult &result =  
        outcome.GetResult();  
  
        const Aws::Vector<Aws::Map<Aws::String,  
        Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();  
  
        if (items.size() == 1) {  
            printMovieInfo(items[0]);  
        }  
        else {  
            std::cerr << "Error: " << items.size() << " movies were  
retrieved. "  
                << " There should be only one movie." << std::endl;  
        }  
    }  
  
    std::cout << "Deleting the movie" << std::endl;  
  
    // 6. Delete the movie using a "Delete" statement. (ExecuteStatement)
```

```
{  
    Aws::DynamoDB::Model::ExecuteStatementRequest request;  
    std::stringstream sqlStream;  
    sqlStream << "DELETE FROM \\" << MOVIE_TABLE_NAME << "\\" WHERE "  
        << TITLE_KEY << "=?" and " << YEAR_KEY << "=?";  
  
    request.SetStatement(sqlStream.str());  
  
    Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;  
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));  
    attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));  
    request.SetParameters(attributes);  
  
    Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =  
    dynamoClient.ExecuteStatement(  
        request);  
    if (!outcome.IsSuccess()) {  
        std::cerr << "Failed to delete the movie: "  
            << outcome.GetError().GetMessage() << std::endl;  
        return false;  
    }  
}  
  
std::cout << "Movie successfully deleted." << std::endl;  
return true;  
}  
  
//! Create a DynamoDB table to be used in sample code scenarios.  
/*!  
 \sa createMoviesDynamoDBTable()  
 \param clientConfiguration: AWS client configuration.  
 \return bool: Function succeeded.  
 */  
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(  
    const Aws::Client::ClientConfiguration &clientConfiguration) {  
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);  
  
    bool movieTableAlreadyExisted = false;  
  
    {  
        Aws::DynamoDB::Model::CreateTableRequest request;  
  
        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;  
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
```

```
yearAttributeDefinition.SetAttributeType(
    Aws::DynamoDB::Model::ScalarAttributeType::N);
request.AddAttributeDefinitions(yearAttributeDefinition);

Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
yearAttributeDefinition.SetAttributeName(TITLE_KEY);
yearAttributeDefinition.SetAttributeType(
    Aws::DynamoDB::Model::ScalarAttributeType::S);
request.AddAttributeDefinitions(yearAttributeDefinition);

Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
    Aws::DynamoDB::Model::KeyType::HASH);
request.AddKeySchema(yearKeySchema);

Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
    Aws::DynamoDB::Model::KeyType::RANGE);
request.AddKeySchema(titleKeySchema);

Aws::DynamoDB::Model::ProvisionedThroughput throughput;
throughput.WithReadCapacityUnits(
    PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
    PROVISIONED_THROUGHPUT_UNITS);
request.SetProvisionedThroughput(throughput);
request.SetTableName(MOVIE_TABLE_NAME);

std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
    request);
if (!result.IsSuccess()) {
    if (result.GetError().GetErrorCode() ==
        Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
        std::cout << "Table already exists." << std::endl;
        movieTableAlreadyExisted = true;
    }
    else {
        std::cerr << "Failed to create table: "
            << result.GetError().GetMessage();
        return false;
    }
}
```

```
}

// Wait for table to become active.
if (!movieTableAlreadyExisted) {
    std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
        << "' to become active...." << std::endl;
    if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
clientConfiguration)) {
        return false;
    }
    std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
        << std::endl;
}

return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
\sa deleteMoviesDynamoDBTable()
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
    const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
dynamoClient.DeleteTable(
        request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
            << result.GetResult().GetTableDescription().GetTableName()
            << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
            << std::endl;
    }

    return result.IsSuccess();
}
```

```
}

//! Query a newly created DynamoDB table until it is active.
<太后!
\sa waitTableActive()
\param waitTableActive: The DynamoDB table's name.
\param clientConfiguration: AWS client configuration.
\return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                         const Aws::Client::ClientConfiguration
&clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
dynamoClient.DescribeTable(
            request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                  << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run a scenario that creates a table and runs PartiQL queries.

```
// RunPartiQLSingleScenario shows you how to use the AWS SDK for Go
// to use PartiQL to query a table that stores data about movies.
//
// * Use PartiQL statements to add, get, update, and delete data for individual
//   movies.
//
// This example creates an Amazon DynamoDB service client from the specified
// sdkConfig so that
// you can replace it with a mocked or stubbed config for unit testing.
//
// This example creates and deletes a DynamoDB table to use during the scenario.
func RunPartiQLSingleScenario(sdkConfig aws.Config, tableName string) {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("Something went wrong with the demo.")
        }
    }()
}

log.Println(strings.Repeat("-", 88))
log.Println("Welcome to the Amazon DynamoDB PartiQL single action demo.")
log.Println(strings.Repeat("-", 88))

tableBasics := actions.TableBasics{
    DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
    TableName:      tableName,
```

```
}

runner := actions.PartiQLRunner{
    DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
    TableName:      tableName,
}

exists, err := tableBasics.TableExists()
if err != nil {
    panic(err)
}
if !exists {
    log.Printf("Creating table %v...\n", tableName)
    _, err = tableBasics.CreateMovieTable()
    if err != nil {
        panic(err)
    } else {
        log.Printf("Created table %v.\n", tableName)
    }
} else {
    log.Printf("Table %v already exists.\n", tableName)
}
log.Println(strings.Repeat("-", 88))

currentYear, _, _ := time.Now().Date()
customMovie := actions.Movie{
    Title: "24 Hour PartiQL People",
    Year:  currentYear,
    Info: map[string]interface{}{
        "plot":   "A group of data developers discover a new query language they can't stop using.",
        "rating": 9.9,
    },
}

log.Printf("Inserting movie '%v' released in %v.", customMovie.Title,
customMovie.Year)
err = runner.AddMovie(customMovie)
if err == nil {
    log.Printf("Added %v to the movie table.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting data for movie '%v' released in %v.", customMovie.Title,
customMovie.Year)
```

```
movie, err := runner.GetMovie(customMovie.Title, customMovie.Year)
if err == nil {
    log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

newRating := 6.6
log.Printf("Updating movie '%v' with a rating of %v.", customMovie.Title,
newRating)
err = runner.UpdateMovie(customMovie, newRating)
if err == nil {
    log.Printf("Updated %v with a new rating.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting data again to verify the update.")
movie, err = runner.GetMovie(customMovie.Title, customMovie.Year)
if err == nil {
    log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Deleting movie '%v'.\n", customMovie.Title)
err = runner.DeleteMovie(customMovie)
if err == nil {
    log.Printf("Deleted %v.\n", customMovie.Title)
}

err = tableBasics.DeleteTable()
if err == nil {
    log.Printf("Deleted table %v.\n", tableBasics.TableName)
}

log.Println(strings.Repeat("-", 88))
log.Println("Thanks for watching!")
log.Println(strings.Repeat("-", 88))
}
```

Define a Movie struct that is used in this example.

```
// Movie encapsulates data about a movie. Title and Year are the composite
// primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
// key,
// and Info is additional data.
type Movie struct {
    Title string            `dynamodbav:"title"`
    Year  int               `dynamodbav:"year"`
    Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
    title, err := attributevalue.Marshal(movie.Title)
    if err != nil {
        panic(err)
    }
    year, err := attributevalue.Marshal(movie.Year)
    if err != nil {
        panic(err)
    }
    return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
// example.
func (movie Movie) String() string {
    return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
        movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

Create a struct and methods that run PartiQL statements.

```
// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
// the
// specified table.
type PartiQLRunner struct {
    DynamoDbClient *dynamodb.Client
```

```
TableName      string
}

// AddMovie runs a PartiQL INSERT statement to add a movie to the DynamoDB table.
func (runner PartiQLRunner) AddMovie(movie Movie) error {
    params, err := attributevalue.MarshalList([]interface{}{movie.Title, movie.Year,
    movie.Info})
    if err != nil {
        panic(err)
    }
    _, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
    &dynamodb.ExecuteStatementInput{
        Statement: aws.String(
            fmt.Sprintf("INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
            runner.TableName)),
        Parameters: params,
    })
    if err != nil {
        log.Printf("Couldn't insert an item with PartiQL. Here's why: %v\n", err)
    }
    return err
}

// GetMovie runs a PartiQL SELECT statement to get a movie from the DynamoDB
// table by
// title and year.
func (runner PartiQLRunner) GetMovie(title string, year int) (Movie, error) {
    var movie Movie
    params, err := attributevalue.MarshalList([]interface{}{title, year})
    if err != nil {
        panic(err)
    }
    response, err := runner.DynamoDbClient.ExecuteStatement(context.TODO(),
    &dynamodb.ExecuteStatementInput{
        Statement: aws.String(
            fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
            runner.TableName)),
        Parameters: params,
    })
    if err != nil {
```

```
    log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
} else {
    err = attributevalue.UnmarshalMap(response.Items[0], &movie)
    if err != nil {
        log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
    }
}
return movie, err
}

// UpdateMovie runs a PartiQL UPDATE statement to update the rating of a movie
// that
// already exists in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovie(movie Movie, rating float64) error {
    params, err := attributevalue.MarshalList([]interface{}{rating, movie.Title,
        movie.Year})
    if err != nil {
        panic(err)
    }
    _, err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
        &dynamodb.ExecuteStatementInput{
            Statement: aws.String(
                fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
                    runner.TableName)),
            Parameters: params,
        })
    if err != nil {
        log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
    }
    return err
}

// DeleteMovie runs a PartiQL DELETE statement to remove a movie from the
// DynamoDB table.
func (runner PartiQLRunner) DeleteMovie(movie Movie) error {
    params, err := attributevalue.MarshalList([]interface{}{movie.Title,
        movie.Year})
    if err != nil {
        panic(err)
    }
}
```

```
_ , err = runner.DynamoDbClient.ExecuteStatement(context.TODO(),
&dynamodb.ExecuteStatementInput{
    Statement: aws.String(
        fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
            runner.TableName)),
    Parameters: params,
})
if err != nil {
    log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
    err)
}
return err
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public class ScenarioPartiQ {
    public static void main(String[] args) throws IOException {
        final String usage = """
        Usage:
        <fileName>
        Where:
        fileName - The path to the moviedata.json file that you can
        download from the Amazon DynamoDB Developer Guide.
        """;
        if (args.length != 1) {
            System.out.println(usage);
        }
    }
}
```

```
        System.exit(1);
    }

    String fileName = args[0];
    String tableName = "MoviesPartiQ";
    Region region = Region.US_EAST_1;
    DynamoDbClient ddb = DynamoDbClient.builder()
        .region(region)
        .build();

    System.out.println(
        "***** Creating an Amazon DynamoDB table named MoviesPartiQ
with a key named year and a sort key named title.");
    createTable(ddb, tableName);

    System.out.println("***** Loading data into the MoviesPartiQ table.");
    loadData(ddb, fileName);

    System.out.println("***** Getting data from the MoviesPartiQ table.");
    getItem(ddb);

    System.out.println("***** Putting a record into the MoviesPartiQ
table.");
    putRecord(ddb);

    System.out.println("***** Updating a record.");
    updateTableItem(ddb);

    System.out.println("***** Querying the movies released in 2013.");
    queryTable(ddb);

    System.out.println("***** Deleting the Amazon DynamoDB table.");
    deleteDynamoDBTable(ddb, tableName);
    ddb.close();
}

public static void createTable(DynamoDbClient ddb, String tableName) {
    DynamoDbWaiter dbWaiter = ddb.waiter();
    ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

    // Define attributes.
    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("year")
        .attributeType("N")
```

```
.build());  
  
attributeDefinitions.add(AttributeDefinition.builder()  
    .attributeName("title")  
    .attributeType("S")  
    .build());  
  
ArrayList<KeySchemaElement> tableKey = new ArrayList<>();  
KeySchemaElement key = KeySchemaElement.builder()  
    .attributeName("year")  
    .keyType(KeyType.HASH)  
    .build();  
  
KeySchemaElement key2 = KeySchemaElement.builder()  
    .attributeName("title")  
    .keyType(KeyType.RANGE) // Sort  
    .build();  
  
// Add KeySchemaElement objects to the list.  
tableKey.add(key);  
tableKey.add(key2);  
  
CreateTableRequest request = CreateTableRequest.builder()  
    .keySchema(tableKey)  
    .provisionedThroughput(ProvisionedThroughput.builder()  
        .readCapacityUnits(new Long(10))  
        .writeCapacityUnits(new Long(10))  
        .build())  
    .attributeDefinitions(attributeDefinitions)  
    .tableName(tableName)  
    .build();  
  
try {  
    CreateTableResponse response = ddb.createTable(request);  
    DescribeTableRequest tableRequest = DescribeTableRequest.builder()  
        .tableName(tableName)  
        .build();  
  
    // Wait until the Amazon DynamoDB table is created.  
    WaiterResponse<DescribeTableResponse> waiterResponse =  
    dbWaiter.waitUntilTableExists(tableRequest);  
    waiterResponse.matched().response().ifPresent(System.out::println);  
    String newTable = response.tableDescription().tableName();  
    System.out.println("The " + newTable + " was successfully created.");
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String fileName) throws
IOException {

        String sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?:,
'title' : ?, 'info' : ?}";
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
        ObjectNode currentNode;
        int t = 0;
        List<AttributeValue> parameters = new ArrayList<>();
        while (iter.hasNext()) {

            // Add 200 movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();

            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf(year))
                .build();

            AttributeValue att2 = AttributeValue.builder()
                .s(title)
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s(info)
                .build();

            parameters.add(att1);
```

```
        parameters.add(att2);
        parameters.add(att3);

        // Insert the movie into the Amazon DynamoDB table.
        executeStatementRequest(ddb, sqlStatement, parameters);
        System.out.println("Added Movie " + title);

        parameters.remove(att1);
        parameters.remove(att2);
        parameters.remove(att3);
        t++;
    }
}

public static void getItem(DynamoDbClient ddb) {

    String sqlStatement = "SELECT * FROM MoviesPartiQ where year=? and
title=?";
    List<AttributeValue> parameters = new ArrayList<>();
    AttributeValue att1 = AttributeValue.builder()
        .n("2012")
        .build();

    AttributeValue att2 = AttributeValue.builder()
        .s("The Perks of Being a Wallflower")
        .build();

    parameters.add(att1);
    parameters.add(att2);

    try {
        ExecuteStatementResponse response = executeStatementRequest(ddb,
sqlStatement, parameters);
        System.out.println("ExecuteStatement successful: " +
response.toString());

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

public static void putRecord(DynamoDbClient ddb) {
```

```
String sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?,  
'title' : ?, 'info' : ?}";  
try {  
    List<AttributeValue> parameters = new ArrayList<>();  
  
    AttributeValue att1 = AttributeValue.builder()  
        .n(String.valueOf("2020"))  
        .build();  
  
    AttributeValue att2 = AttributeValue.builder()  
        .s("My Movie")  
        .build();  
  
    AttributeValue att3 = AttributeValue.builder()  
        .s("No Information")  
        .build();  
  
    parameters.add(att1);  
    parameters.add(att2);  
    parameters.add(att3);  
  
    executeStatementRequest(ddb, sqlStatement, parameters);  
    System.out.println("Added new movie.");  
  
} catch (DynamoDbException e) {  
    System.err.println(e.getMessage());  
    System.exit(1);  
}  
}  
  
public static void updateTableItem(DynamoDbClient ddb) {  
  
    String sqlStatement = "UPDATE MoviesPartiQ SET info = 'directors\\":  
    ["Merian C. Cooper","Ernest B. Schoedsack'] where year=? and title=?";  
    List<AttributeValue> parameters = new ArrayList<>();  
    AttributeValue att1 = AttributeValue.builder()  
        .n(String.valueOf("2013"))  
        .build();  
  
    AttributeValue att2 = AttributeValue.builder()  
        .s("The East")  
        .build();  
  
    parameters.add(att1);
```

```
parameters.add(att2);

try {
    executeStatementRequest(ddb, sqlStatement, parameters);

} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
System.out.println("Item was updated!");
}

// Query the table where the year is 2013.
public static void queryTable(DynamoDbClient ddb) {
    String sqlStatement = "SELECT * FROM MoviesPartiQ where year = ? ORDER BY
year";
    try {

        List<AttributeValue> parameters = new ArrayList<>();
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2013"))
            .build();
        parameters.add(att1);

        // Get items in the table and write out the ID value.
        ExecuteStatementResponse response = executeStatementRequest(ddb,
sqlStatement, parameters);
        System.out.println("ExecuteStatement successful: " +
response.toString());

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{
    DeleteTableRequest request = DeleteTableRequest.builder()
        .tableName(tableName)
        .build();

    try {
```

```
        ddb.deleteTable(request);

    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    System.out.println(tableName + " was successfully deleted!");
}

private static ExecuteStatementResponse
executeStatementRequest(DynamoDbClient ddb, String statement,
    List<AttributeValue> parameters) {
    ExecuteStatementRequest request = ExecuteStatementRequest.builder()
        .statement(statement)
        .parameters(parameters)
        .build();

    return ddb.executeStatement(request);
}

private static void processResults(ExecuteStatementResponse
executeStatementResult) {
    System.out.println("ExecuteStatement successful: " +
executeStatementResult.toString());
}
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Execute single PartiQL statements.

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "SingleOriginCoffees";

export const main = async () => {
  /**
   * Create a table.
   */

  log("Creating a table.");
  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "varietal",
        // 'S' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "S",
      },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
  });
}
```

```
// Key schema design is important. For more info, see
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [{ AttributeName: "varieta", KeyType: "HASH" }],
});
await client.send(createTableCommand);
log(`Table created: ${tableName}.`);

/**
 * Wait until the table is active.
 */

// This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
// You can't write to a table before it's active.
log("Waiting for the table to be active.");
await waitUntilTableExists({ client }, { TableName: tableName });
log("Table active.");

/**
 * Insert an item.
 */

log("Inserting a coffee into the table.");
const addItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
    Statement: `INSERT INTO ${tableName} value {'varieta':?:, 'profile':?:}`,
    Parameters: ["arabica", ["chocolate", "floral"]],
});
await client.send(addItemStatementCommand);
log(`Coffee inserted.`);

/**
 * Select an item.
 */

log("Selecting the coffee from the table.");
const selectItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
    Statement: `SELECT * FROM ${tableName} WHERE varieta=?`,
    Parameters: ["arabica"],
});
const selectItemResponse = await docClient.send(selectItemStatementCommand);
```

```
log(`Got coffee: ${JSON.stringify(selectItemResponse.Items[0])}`);

/**
 * Update the item.
 */

log("Add a flavor profile to the coffee.");
const updateItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
    Statement: `UPDATE ${tableName} SET profile=list_append(profile, ?) WHERE
varietal=?`,
    Parameters: [["fruity"], "arabica"],
});
await client.send(updateItemStatementCommand);
log(`Updated coffee`);

/**
 * Delete the item.
 */

log("Deleting the coffee.");
const deleteItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
    Statement: `DELETE FROM ${tableName} WHERE varietal=?`,
    Parameters: ["arabica"],
});
await docClient.send(deleteItemStatementCommand);
log("Coffee deleted.");

/**
 * Delete the table.
 */

log("Deleting the table.");
const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
await client.send(deleteTableCommand);
log("Table deleted.");
});
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun main(args: Array<String>) {  
    val usage = """  
        Usage:  
        <fileName>  
  
        Where:  
        fileName - The path to the moviedata.json you can download from the  
        Amazon DynamoDB Developer Guide.  
        """  
  
    if (args.size != 1) {  
        println(usage)  
        exitProcess(1)  
    }  
  
    val ddb = DynamoDbClient { region = "us-east-1" }  
    val tableName = "MoviesPartiQ"  
  
    // Get the moviedata.json from the Amazon DynamoDB Developer Guide.  
    val fileName = args[0]  
    println("Creating an Amazon DynamoDB table named MoviesPartiQ with a key  
    named id and a sort key named title.")  
    createTablePartiQL(ddb, tableName, "year")  
    loadDataPartiQL(ddb, fileName)  
  
    println("***** Getting data from the MoviesPartiQ table.")  
    getMoviePartiQL(ddb)  
  
    println("***** Putting a record into the MoviesPartiQ table.")  
    putRecordPartiQL(ddb)  
  
    println("***** Updating a record.")
```

```
updateTableItemPartiQL(ddb)

    println("***** Querying the movies released in 2013.")
    queryTablePartiQL(ddb)

    println("***** Deleting the MoviesPartiQ table.")
    deleteTablePartiQL(tableName)
}

suspend fun createTablePartiQL(ddb: DynamoDbClient, tableNameVal: String, key: String) {
    val attDef = AttributeDefinition {
        attributeName = key
        attributeType = ScalarAttributeType.N
    }

    val attDef1 = AttributeDefinition {
        attributeName = "title"
        attributeType = ScalarAttributeType.S
    }

    val keySchemaVal = KeySchemaElement {
        attributeName = key
        keyType = KeyType.Hash
    }

    val keySchemaVal1 = KeySchemaElement {
        attributeName = "title"
        keyType = KeyType.Range
    }

    val provisionedVal = ProvisionedThroughput {
        readCapacityUnits = 10
        writeCapacityUnits = 10
    }

    val request = CreateTableRequest {
        attributeDefinitions = listOf(attDef, attDef1)
        keySchema = listOf(keySchemaVal, keySchemaVal1)
        provisionedThroughput = provisionedVal
        tableName = tableNameVal
    }

    val response = ddb.createTable(request)
```

```
        ddb.waitUntilTableExists { // suspend call
            tableName = tableNameVal
        }
        println("The table was successfully created
${response.tableDescription?.tableArn}")
    }

suspend fun loadDataPartiQL(ddb: DynamoDbClient, fileName: String) {
    val sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?, 'title' : ?,
'info' : ?}"
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode
    var t = 0

    while (iter.hasNext()) {
        if (t == 200) {
            break
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()

        val parameters: MutableList<AttributeValue> = ArrayList<AttributeValue>()
        parameters.add(AttributeValue.N(year.toString()))
        parameters.add(AttributeValue.S(title))
        parameters.add(AttributeValue.S(info))

        executeStatementPartiQL(ddb, sqlStatement, parameters)
        println("Added Movie $title")
        parameters.clear()
        t++
    }
}

suspend fun getMoviePartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "SELECT * FROM MoviesPartiQ where year=? and title=?"
    val parameters: MutableList<AttributeValue> = ArrayList<AttributeValue>()
    parameters.add(AttributeValue.N("2012"))
    parameters.add(AttributeValue.S("The Perks of Being a Wallflower"))
    val response = executeStatementPartiQL(ddb, sqlStatement, parameters)
```

```
        println("ExecuteStatement successful: $response")
    }

suspend fun putRecordPartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?, 'title' : ?, 'info' : ?}"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2020"))
    parameters.add(AttributeValue.S("My Movie"))
    parameters.add(AttributeValue.S("No Info"))
    executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("Added new movie.")
}

suspend fun updateTableItemPartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "UPDATE MoviesPartiQ SET info = 'directors\":[\"Merian C. Cooper\", \"Ernest B. Schoedsack\"]' where year=? and title=?"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2013"))
    parameters.add(AttributeValue.S("The East"))
    executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("Item was updated!")
}

// Query the table where the year is 2013.
suspend fun queryTablePartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "SELECT * FROM MoviesPartiQ where year = ?"

    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2013"))
    val response = executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("ExecuteStatement successful: $response")
}

suspend fun deleteTablePartiQL(tableNameVal: String) {
    val request = DeleteTableRequest {
        tableName = tableNameVal
    }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}
```

```
suspend fun executeStatementPartiQL(
    ddb: DynamoDbClient,
    statementVal: String,
    parametersVal: List<AttributeValue>
): ExecuteStatementResponse {
    val request = ExecuteStatementRequest {
        statement = statementVal
        parameters = parametersVal
    }

    return ddb.executeStatement(request)
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
namespace DynamoDb\PartiQL_Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;

use function AwsUtilities\testable_readline;
use function AwsUtilities\loadMovieData;

class GettingStartedWithPartiQL
{
    public function run()
    {
        echo("\n");
        echo("-----\n");
```

```
print("Welcome to the Amazon DynamoDB - PartiQL getting started demo\nusing PHP!\n");
echo("-----\n");

$uuid = uniqid();
$service = new DynamoDb\DynamoDBService();

$tableName = "partiql_demo_table_$uuid";
$service->createTable(
    $tableName,
    [
        new DynamoDBAttribute('year', 'N', 'HASH'),
        new DynamoDBAttribute('title', 'S', 'RANGE')
    ]
);

echo "Waiting for table...";
$service->dynamoDbClient->waitForTable($tableName);
echo "table $tableName found!\n";

echo "What's the name of the last movie you watched?\n";
while (empty($movieName)) {
    $movieName = testable_readline("Movie name: ");
}
echo "And what year was it released?\n";
$movieYear = "year";
while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
    $movieYear = testable_readline("Year released: ");
}
$key = [
    'Item' => [
        'year' => [
            'N' => "$movieYear",
        ],
        'title' => [
            'S' => $movieName,
        ],
    ],
];
list($statement, $parameters) = $service-
>buildStatementAndParameters("INSERT", $tableName, $key);
$service->insertItemByPartiQL($statement, $parameters);
```

```
echo "How would you rate the movie from 1-10?\n";
$rating = 0;
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
    $rating = testable_readline("Rating (1-10): ");
}
echo "What was the movie about?\n";
while (empty($plot)) {
    $plot = testable_readline("Plot summary: ");
}
$attributes = [
    new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
    new DynamoDBAttribute('plot', 'S', 'RANGE', $plot),
];
list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
$service->updateItemByPartiQL($statement, $parameters);
echo "Movie added and updated.\n";

$batch = json_decode(loadMovieData());

$service->writeBatch($tableName, $batch);

$movie = $service->getItemByPartiQL($tableName, $key);
echo "\nThe movie {$movie['Items'][0]['title']['S']} was released in
{$movie['Items'][0]['year']['N']}.\n";
echo "What rating would you like to give {$movie['Items'][0]['title']
['S']}?\n";
$rating = 0;
while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
    $rating = testable_readline("Rating (1-10): ");
}
$attributes = [
    new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
    new DynamoDBAttribute('plot', 'S', 'RANGE', $plot)
];
list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
$service->updateItemByPartiQL($statement, $parameters);
```

```
$movie = $service->getItemByPartiQL($tableName, $key);
echo "Okay, you have rated {$movie['Items'][0]['title']['S']} as a
{$movie['Items'][0]['rating']['N']}\n";

$service->deleteItemByPartiQL($statement, $parameters);
echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";

echo "That's okay though. The book was better. Now, for something
lighter, in what year were you born?\n";
$birthYear = "not a number";
while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
    $birthYear = testable_readline("Birth year: ");
}
$birthKey = [
    'Key' => [
        'year' => [
            'N' => "$birthYear",
        ],
    ],
];
$result = $service->query($tableName, $birthKey);
$marshal = new Marshaler();
echo "Here are the movies in our collection released the year you were
born:\n";
$oops = "Oops! There were no movies released in that year (that we know
of).\n";
$display = "";
foreach ($result['Items'] as $movie) {
    $movie = $marshal->unmarshalItem($movie);
    $display .= $movie['title'] . "\n";
}
echo ($display) ?: $oops;

$yearsKey = [
    'Key' => [
        'year' => [
            'N' => [
                'minRange' => 1990,
                'maxRange' => 1999,
            ],
        ],
    ],
];

```

```
$filter = "year between 1990 and 1999";
echo "\nHere's a list of all the movies released in the 90s:\n";
$result = $service->scan($tableName, $yearsKey, $filter);
foreach ($result['Items'] as $movie) {
    $movie = $marshal->unmarshalItem($movie);
    echo $movie['title'] . "\n";
}

echo "\nCleaning up this demo by deleting table $tableName...\n";
$service->deleteTable($tableName);
}

public function insertItemByPartiQL(string $statement, array $parameters)
{
    $this->dynamoDbClient->executeStatement([
        'Statement' => "$statement",
        'Parameters' => $parameters,
    ]);
}

public function getItemByPartiQL(string $tableName, array $key): Result
{
    list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);

    return $this->dynamoDbClient->executeStatement([
        'Parameters' => $parameters,
        'Statement' => $statement,
    ]);
}

public function updateItemByPartiQL(string $statement, array $parameters)
{
    $this->dynamoDbClient->executeStatement([
        'Statement' => $statement,
        'Parameters' => $parameters,
    ]);
}

public function deleteItemByPartiQL(string $statement, array $parameters)
{
    $this->dynamoDbClient->executeStatement([
        'Statement' => $statement,
```

```
    'Parameters' => $parameters,  
]);  
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a class that can run PartiQL statements.

```
from datetime import datetime  
from decimal import Decimal  
import logging  
from pprint import pprint  
  
import boto3  
from botocore.exceptions import ClientError  
  
from scaffold import Scaffold  
  
logger = logging.getLogger(__name__)  
  
class PartiQLWrapper:  
    """  
    Encapsulates a DynamoDB resource to run PartiQL statements.  
    """  
  
    def __init__(self, dyn_resource):  
        """  
        :param dyn_resource: A Boto3 DynamoDB resource.  
        """  
        self.dyn_resource = dyn_resource
```

```
def run_partiql(self, statement, params):
    """
    Runs a PartiQL statement. A Boto3 resource is used even though
    `execute_statement` is called on the underlying `client` object because
    the
        resource transforms input and output from plain old Python objects
    (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
    transforms yourself.

    :param statement: The PartiQL statement.
    :param params: The list of PartiQL parameters. These are applied to the
        statement in the order they are listed.
    :return: The items returned from the statement, if any.
    """
    try:
        output = self.dyn_resource.meta.client.execute_statement(
            Statement=statement, Parameters=params
        )
    except ClientError as err:
        if err.response["Error"]["Code"] == "ResourceNotFoundException":
            logger.error(
                "Couldn't execute PartiQL '%s' because the table does not
exist.",
                statement,
            )
        else:
            logger.error(
                "Couldn't execute PartiQL '%s'. Here's why: %s: %s",
                statement,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
    raise
else:
    return output
```

Run a scenario that creates a table and runs PartiQL queries.

```
def run_scenario(scaffold, wrapper, table_name):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB PartiQL single statement demo.")
    print("-" * 88)

    print(f"Creating table '{table_name}' for the demo...")
    scaffold.create_table(table_name)
    print("-" * 88)

    title = "24 Hour PartiQL People"
    year = datetime.now().year
    plot = "A group of data developers discover a new query language they can't
stop using."
    rating = Decimal("9.9")

    print(f"Inserting movie '{title}' released in {year}.")
    wrapper.run_partiql(
        f"INSERT INTO \\"{table_name}\\" VALUE {{'title': ?, 'year': ?, "
        'info': ?}}",
        [title, year, {"plot": plot, "rating": rating}],
    )
    print("Success!")
    print("-" * 88)

    print(f"Getting data for movie '{title}' released in {year}.")
    output = wrapper.run_partiql(
        f"SELECT * FROM \"{table_name}\" WHERE title=? AND year=?", [title, year]
    )
    for item in output["Items"]:
        print(f"\n{item['title']}, {item['year']}")
    pprint(output["Items"])
    print("-" * 88)

    rating = Decimal("2.4")
    print(f"Updating movie '{title}' with a rating of {float(rating)}.")
    wrapper.run_partiql(
        f"UPDATE \"{table_name}\" SET info.rating=? WHERE title=? AND year=?",
        [rating, title, year],
    )
    print("Success!")
    print("-" * 88)
```

```
print(f"Getting data again to verify our update.")
output = wrapper.run_partiql(
    f'SELECT * FROM "{table_name}" WHERE title=? AND year=?', [title, year]
)
for item in output["Items"]:
    print(f"\n{item['title']}, {item['year']}")
    pprint(output["Items"])
print("-" * 88)

print(f"Deleting movie '{title}' released in {year}.")
wrapper.run_partiql(
    f'DELETE FROM "{table_name}" WHERE title=? AND year=?', [title, year]
)
print("Success!")
print("-" * 88)

print(f"Deleting table '{table_name}'...")
scaffold.delete_table()
print("-" * 88)

print("\nThanks for watching!")
print("-" * 88)

if __name__ == "__main__":
    try:
        dyn_res = boto3.resource("dynamodb")
        scaffold = Scaffold(dyn_res)
        movies = PartiQLWrapper(dyn_res)
        run_scenario(scaffold, movies, "doc-example-table-partiql-movies")
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run a scenario that creates a table and runs PartiQL queries.

```
table_name = "doc-example-table-movies-partiql-#{rand(10**8)}"
scaffold = Scaffold.new(table_name)
sdk = DynamoDBPartiQLSingle.new(table_name)

new_step(1, "Create a new DynamoDB table if none already exists.")
unless scaffold.exists?(table_name)
    puts("\nNo such table: #{table_name}. Creating it...")
    scaffold.create_table(table_name)
    print "Done!\n".green
end

new_step(2, "Populate DynamoDB table with movie data.")
download_file = "moviedata.json"
puts("Downloading movie database to #{download_file}...")
movie_data = scaffold.fetch_movie_data(download_file)
puts("Writing movie data from #{download_file} into your table...")
scaffold.write_batch(movie_data)
puts("Records added: #{movie_data.length}.")
print "Done!\n".green

new_step(3, "Select a single item from the movies table.")
response = sdk.select_item_by_title("Star Wars")
puts("Items selected for title 'Star Wars': #{response.items.length}\n")
print "#{response.items.first}".yellow
print "\n\nDone!\n".green

new_step(4, "Update a single item from the movies table.")
puts "Let's correct the rating on The Big Lebowski to 10.0."
sdk.update_rating_by_title("The Big Lebowski", 1998, 10.0)
print "\nDone!\n".green
```

```
new_step(5, "Delete a single item from the movies table.")
puts "Let's delete The Silence of the Lambs because it's just too scary."
sdk.delete_item_by_title("The Silence of the Lambs", 1991)
print "\nDone!\n".green

new_step(6, "Insert a new item into the movies table.")
puts "Let's create a less-scary movie called The Prancing of the Lambs."
sdk.insert_item("The Prancing of the Lambs", 2005, "A movie about happy
livestock.", 5.0)
print "\nDone!\n".green

new_step(7, "Delete the table.")
if scaffold.exists?(table_name)
  scaffold.delete_table
end
end
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
async fn make_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<(), SdkError<CreateTableError>> {
    let ad = AttributeDefinition::builder()
        .attribute_name(key)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .expect("creating AttributeDefinition");

    let ks = KeySchemaElement::builder()
```

```
.attribute_name(key)
.key_type(KeyType::Hash)
.build()
.expect("creating KeySchemaElement");

let pt = ProvisionedThroughput::builder()
.read_capacity_units(10)
.write_capacity_units(5)
.build()
.expect("creating ProvisionedThroughput");

match client
.create_table()
.table_name(table)
.key_schema(ks)
.attribute_definitions(ad)
.provisioned_throughput(pt)
.send()
.await
{
Ok(_) => Ok(()),
Err(e) => Err(e),
}
}

async fn add_item(client: &Client, item: Item) -> Result<(), SdkError<ExecuteStatementError>> {
match client
.execute_statement()
.statement(format!(
r#"INSERT INTO "{}" VALUE {{"
"{}": ?, "account_type": ?, "age": ?, "first_name": ?, "last_name": ?"
}} "#,
item.table, item.key
))
.set_parameters(Some(vec![
AttributeValue::S(item.utype),
AttributeValue::S(item.age),
AttributeValue::S(item.first_name),
AttributeValue::S(item.last_name),
]))
```

```
        ]))
        .send()
        .await
    {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

async fn query_item(client: &Client, item: Item) -> bool {
    match client
        .execute_statement()
        .statement(format!(
            r#"SELECT * FROM "{}" WHERE "{}" = ?"#
            , item.table, item.key
        ))
        .set_parameters(Some(vec![AttributeValue::S(item.value)])))
        .send()
        .await
    {
        Ok(resp) => {
            if !resp.items().is_empty() {
                println!("Found a matching entry in the table:");
                println!("{}: {:?}", resp.items.unwrap_or_default().pop());
                true
            } else {
                println!("Did not find a match.");
                false
            }
        }
        Err(e) => {
            println!("Got an error querying table:");
            println!("{}: {}", e);
            process::exit(1);
        }
    }
}

async fn remove_item(client: &Client, table: &str, key: &str, value: String) ->
Result<(), Error> {
    client
        .execute_statement()
        .statement(format!(r#"DELETE FROM "{}" WHERE "{}" = ?"#
            , table, key))
        .set_parameters(Some(vec![AttributeValue::S(value)])))
}
```

```
.send()  
.await?;  
  
println!("Deleted item.");  
  
Ok(())  
}  
  
async fn remove_table(client: &Client, table: &str) -> Result<(), Error> {  
    client.delete_table().table_name(table).send().await?;  
  
    Ok(())  
}
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use a document model for DynamoDB using an AWS SDK

The following code example shows how to perform Create, Read, Update, and Delete (CRUD) and batch operations using a document model for DynamoDB and an AWS SDK.

For more information, see [Document model](#).

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Perform CRUD operations using a document model.

```
/// <summary>
/// Performs CRUD operations on an Amazon DynamoDB table.
/// </summary>
public class MidlevelItemCRUD
{
    public static async Task Main()
    {
        var tableName = "ProductCatalog";
        var sampleBookId = 555;

        var client = new AmazonDynamoDBClient();
        var productCatalog = LoadTable(client, tableName);

        await CreateBookItem(productCatalog, sampleBookId);
        RetrieveBook(productCatalog, sampleBookId);

        // Couple of sample updates.
        UpdateMultipleAttributes(productCatalog, sampleBookId);
        UpdateBookPriceConditionally(productCatalog, sampleBookId);

        // Delete.
        await DeleteBook(productCatalog, sampleBookId);
    }

    /// <summary>
    /// Loads the contents of a DynamoDB table.
    /// </summary>
    /// <param name="client">An initialized DynamoDB client object.</param>
    /// <param name="tableName">The name of the table to load.</param>
    /// <returns>A DynamoDB table object.</returns>
    public static Table LoadTable(IAmazonDynamoDB client, string tableName)
    {
        Table productCatalog = Table.LoadTable(client, tableName);
        return productCatalog;
    }

    /// <summary>
    /// Creates an example book item and adds it to the DynamoDB table
    /// ProductCatalog.
    /// </summary>
    /// <param name="productCatalog">A DynamoDB table object.</param>
    /// <param name="sampleBookId">An integer value representing the book's
    ID.</param>
```

```
public static async Task CreateBookItem(Table productCatalog, int sampleBookId)
{
    Console.WriteLine("\n*** Executing CreateBookItem() ***");
    var book = new Document
    {
        ["Id"] = sampleBookId,
        ["Title"] = "Book " + sampleBookId,
        ["Price"] = 19.99,
        ["ISBN"] = "111-1111111111",
        ["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" },
        ["PageCount"] = 500,
        ["Dimensions"] = "8.5x11x.5",
        ["InPublication"] = new DynamoDBBool(true),
        ["InStock"] = new DynamoDBBool(false),
        ["QuantityOnHand"] = 0,
    };

    // Adds the book to the ProductCatalog table.
    await productCatalog.PutItemAsync(book);
}

/// <summary>
/// Retrieves an item, a book, from the DynamoDB ProductCatalog table.
/// </summary>
/// <param name="productCatalog">A DynamoDB table object.</param>
/// <param name="sampleBookId">An integer value representing the book's ID.</param>
public static async void RetrieveBook(
    Table productCatalog,
    int sampleBookId)
{
    Console.WriteLine("\n*** Executing RetrieveBook() ***");

    // Optional configuration.
    var config = new GetItemOperationConfig
    {
        AttributesToGet = new List<string> { "Id", "ISBN", "Title",
        "Authors", "Price" },
        ConsistentRead = true,
    };
}
```

```
        Document document = await productCatalog.GetItemAsync(sampleBookId,
config);
        Console.WriteLine("RetrieveBook: Printing book retrieved...");  
        PrintDocument(document);
    }

    ///<summary>
    /// Updates multiple attributes for a book and writes the changes to the
    /// DynamoDB table ProductCatalog.
    ///</summary>
    ///<param name="productCatalog">A DynamoDB table object.</param>
    ///<param name="sampleBookId">An integer value representing the book's
ID.</param>
    public static async void UpdateMultipleAttributes(
        Table productCatalog,
        int sampleBookId)
{
    Console.WriteLine("\nUpdating multiple attributes....");
    int partitionKey = sampleBookId;

    var book = new Document
    {
        ["Id"] = partitionKey,

        // List of attribute updates.
        // The following replaces the existing authors list.
        ["Authors"] = new List<string> { "Author x", "Author y" },
        ["newAttribute"] = "New Value",
        ["ISBN"] = null, // Remove it.
    };

    // Optional parameters.
    var config = new UpdateItemOperationConfig
    {
        // Gets updated item in response.
        ReturnValues = ReturnValues.AllNewAttributes,
    };

    Document updatedBook = await productCatalog.UpdateItemAsync(book,
config);
    Console.WriteLine("UpdateMultipleAttributes: Printing item after
updates ...");
    PrintDocument(updatedBook);
}
```

```
/// <summary>
/// Updates a book item if it meets the specified criteria.
/// </summary>
/// <param name="productCatalog">A DynamoDB table object.</param>
/// <param name="sampleBookId">An integer value representing the book's
ID.</param>
public static async void UpdateBookPriceConditionally(
    Table productCatalog,
    int sampleBookId)
{
    Console.WriteLine("\n*** Executing UpdateBookPriceConditionally()
***");

    int partitionKey = sampleBookId;

    var book = new Document
    {
        ["Id"] = partitionKey,
        ["Price"] = 29.99,
    };

    // For conditional price update, creating a condition expression.
    var expr = new Expression
    {
        ExpressionStatement = "Price = :val",
    };
    expr.ExpressionAttributeValues[":val"] = 19.00;

    // Optional parameters.
    var config = new UpdateItemOperationConfig
    {
        ConditionalExpression = expr,
        ReturnValue = ReturnValue.AllNewAttributes,
    };

    Document updatedBook = await productCatalog.UpdateItemAsync(book,
config);
    Console.WriteLine("UpdateBookPriceConditionally: Printing item whose
price was conditionally updated");
    PrintDocument(updatedBook);
}

/// <summary>
```

```
/// Deletes the book with the supplied Id value from the DynamoDB table
/// ProductCatalog.
/// </summary>
/// <param name="productCatalog">A DynamoDB table object.</param>
/// <param name="sampleBookId">An integer value representing the book's
ID.</param>
public static async Task DeleteBook(
    Table productCatalog,
    int sampleBookId)
{
    Console.WriteLine("\n*** Executing DeleteBook() ***");

    // Optional configuration.
    var config = new DeleteItemOperationConfig
    {
        // Returns the deleted item.
        ReturnValues = ReturnValues.AllOldAttributes,
    };
    Document document = await
productCatalog.DeleteItemAsync(sampleBookId, config);
    Console.WriteLine("DeleteBook: Printing deleted just deleted...");

    PrintDocument(document);
}

/// <summary>
/// Prints the information for the supplied DynamoDB document.
/// </summary>
/// <param name="updatedDocument">A DynamoDB document object.</param>
public static void PrintDocument(Document updatedDocument)
{
    if (updatedDocument == null)
    {
        return;
    }

    foreach (var attribute in updatedDocument.GetAttributeNames())
    {
        string stringValue = null;
        var value = updatedDocument[attribute];

        if (value == null)
        {
            continue;
        }

        if (value is String)
        {
            stringValue = value.ToString();
        }
        else if (value is Number)
        {
            stringValue = value.ToString("F");
        }
        else if (value is Boolean)
        {
            stringValue = value.ToString().ToLower();
        }
        else if (value is Binary)
        {
            stringValue = Convert.ToBase64String((byte[])value);
        }
        else if (value is Map)
        {
            stringValue = PrintDocument((Map)value);
        }
        else if (value is List)
        {
            stringValue = PrintDocument((List<Object>)value);
        }
        else
        {
            stringValue = value.ToString();
        }

        Console.WriteLine(attribute + ": " + stringValue);
    }
}
```

```
        }

        if (value is Primitive)
        {
            stringValue = value.AsPrimitive().Value.ToString();
        }
        else if (value is PrimitiveList)
        {
            stringValue = string.Join(", ", (from primitive
                in value.AsPrimitiveList().Entries
                select
primitive.Value).ToArray());
        }

        Console.WriteLine($"{attribute} - {stringValue}", attribute,
stringValue);
    }
}
}
```

Perform batch write operations using a document model.

```
/// <summary>
/// Shows how to use mid-level Amazon DynamoDB API calls to perform batch
/// operations.
/// </summary>
public class MidLevelBatchWriteItem
{
    public static async Task Main()
    {
        IAmazonDynamoDB client = new AmazonDynamoDBClient();

        await SingleTableBatchWrite(client);
        await MultiTableBatchWrite(client);
    }

    /// <summary>
    /// Perform a batch operation on a single DynamoDB table.
    /// </summary>
    /// <param name="client">An initialized DynamoDB object.</param>
```

```
public static async Task SingleTableBatchWrite(IAmazonDynamoDB client)
{
    Table productCatalog = Table.LoadTable(client, "ProductCatalog");
    var batchWrite = productCatalog.CreateBatchWrite();

    var book1 = new Document
    {
        ["Id"] = 902,
        ["Title"] = "My book1 in batch write using .NET helper classes",
        ["ISBN"] = "902-11-11-1111",
        ["Price"] = 10,
        ["ProductCategory"] = "Book",
        ["Authors"] = new List<string> { "Author 1", "Author 2", "Author
3" },
        ["Dimensions"] = "8.5x11x.5",
        ["InStock"] = new DynamoDBBool(true),
        ["QuantityOnHand"] = new DynamoDBNull(), // Quantity is unknown
at this time.
    };

    batchWrite.AddDocumentToPut(book1);

    // Specify delete item using overload that takes PK.
    batchWrite.AddKeyToDelete(12345);
    Console.WriteLine("Performing batch write in
SingleTableBatchWrite()");
    await batchWrite.ExecuteAsync();
}

/// <summary>
/// Perform a batch operation involving multiple DynamoDB tables.
/// </summary>
/// <param name="client">An initialized DynamoDB client object.</param>
public static async Task MultiTableBatchWrite(IAmazonDynamoDB client)
{
    // Specify item to add in the Forum table.
    Table forum = Table.LoadTable(client, "Forum");
    var forumBatchWrite = forum.CreateBatchWrite();

    var forum1 = new Document
    {
        ["Name"] = "Test BatchWrite Forum",
        ["Threads"] = 0,
    };
}
```

```
        forumBatchWrite.AddDocumentToPut(forum1);

        // Specify item to add in the Thread table.
        Table thread = Table.LoadTable(client, "Thread");
        var threadBatchWrite = thread.CreateBatchWrite();

        var thread1 = new Document
        {
            ["ForumName"] = "S3 forum",
            ["Subject"] = "My sample question",
            ["Message"] = "Message text",
            ["KeywordTags"] = new List<string> { "S3", "Bucket" },
        };
        threadBatchWrite.AddDocumentToPut(thread1);

        // Specify item to delete from the Thread table.
        threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

        // Create multi-table batch.
        var superBatch = new MultiTableDocumentBatchWrite();
        superBatch.AddBatch(forumBatchWrite);
        superBatch.AddBatch(threadBatchWrite);
        Console.WriteLine("Performing batch write in
MultiTableBatchWrite()");

        // Execute the batch.
        await superBatch.ExecuteAsync();
    }
}
```

Scan a table using a document model.

```
/// <summary>
/// Shows how to use mid-level Amazon DynamoDB API calls to scan a DynamoDB
/// table for values.
/// </summary>
public class MidLevelScanOnly
{
    public static async Task Main()
    {
```

```
IAmazonDynamoDB client = new AmazonDynamoDBClient();

Table productCatalogTable = Table.LoadTable(client,
"ProductCatalog");

await FindProductsWithNegativePrice(productCatalogTable);
await FindProductsWithNegativePriceWithConfig(productCatalogTable);
}

/// <summary>
/// Retrieves any products that have a negative price in a DynamoDB
table.
/// </summary>
/// <param name="productCatalogTable">A DynamoDB table object.</param>
public static async Task FindProductsWithNegativePrice(
    Table productCatalogTable)
{
    // Assume there is a price error. So we scan to find items priced <
    0.
    var scanFilter = new ScanFilter();
    scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

    Search search = productCatalogTable.Scan(scanFilter);

    do
    {
        var documentList = await search.GetNextSetAsync();
        Console.WriteLine("\nFindProductsWithNegativePrice:
printing .....");

        foreach (var document in documentList)
        {
            PrintDocument(document);
        }
    }
    while (!search.IsDone);
}

/// <summary>
/// Finds any items in the ProductCatalog table using a DynamoDB
/// configuration object.
/// </summary>
/// <param name="productCatalogTable">A DynamoDB table object.</param>
public static async Task FindProductsWithNegativePriceWithConfig(
```

```
Table productCatalogTable)
{
    // Assume there is a price error. So we scan to find items priced <
    0.
    var scanFilter = new ScanFilter();
    scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

    var config = new ScanOperationConfig()
    {
        Filter = scanFilter,
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string> { "Title", "Id" },
    };

    Search search = productCatalogTable.Scan(config);

    do
    {
        var documentList = await search.GetNextSetAsync();
        Console.WriteLine("\nFindProductsWithNegativePriceWithConfig:
printing .....");

        foreach (var document in documentList)
        {
            PrintDocument(document);
        }
    }
    while (!search.IsDone);
}

/// <summary>
/// Displays the details of the passed DynamoDB document object on the
/// console.
/// </summary>
/// <param name="document">A DynamoDB document object.</param>
public static void PrintDocument(Document document)
{
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
        if (value is Primitive)
        {
```

```
        stringValue = value.AsPrimitive().Value.ToString();
    }
    else if (value is PrimitiveList)
    {
        stringValue = string.Join(", ", (from primitive
            in value.AsPrimitiveList().Entries
            select
primitive.Value).ToArray());
    }

    Console.WriteLine($"{attribute} - {stringValue}");
}
}
```

Query and scan a table using a document model.

```
/// <summary>
/// Shows how to perform mid-level query procedures on an Amazon DynamoDB
/// table.
/// </summary>
public class MidLevelQueryAndScan
{
    public static async Task Main()
    {
        IAmazonDynamoDB client = new AmazonDynamoDBClient();

        // Query examples.
        Table replyTable = Table.LoadTable(client, "Reply");
        string forumName = "Amazon DynamoDB";
        string threadSubject = "DynamoDB Thread 2";

        await FindRepliesInLast15Days(replyTable);
        await FindRepliesInLast15DaysWithConfig(replyTable, forumName,
threadSubject);
        await FindRepliesPostedWithinTimePeriod(replyTable, forumName,
threadSubject);

        // Get Example.
    }
}
```

```
        Table productCatalogTable = Table.LoadTable(client,
"ProductCatalog");
        int productId = 101;

        await GetProduct(productCatalogTable, productId);
    }

/// <summary>
/// Retrieves information about a product from the DynamoDB table
/// ProductCatalog based on the product ID and displays the information
/// on the console.
/// </summary>
/// <param name="tableName">The name of the table from which to retrieve
/// product information.</param>
/// <param name="productId">The ID of the product to retrieve.</param>
public static async Task GetProduct(Table tableName, int productId)
{
    Console.WriteLine("**** Executing GetProduct() ***");
    Document productDocument = await tableName.GetItemAsync(productId);
    if (productDocument != null)
    {
        PrintDocument(productDocument);
    }
    else
    {
        Console.WriteLine("Error: product " + productId + " does not
exist");
    }
}

/// <summary>
/// Retrieves replies from the passed DynamoDB table object.
/// </summary>
/// <param name="table">The table we want to query.</param>
public static async Task FindRepliesInLast15Days(
    Table table)
{
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
    var filter = new QueryFilter("Id", QueryOperator.Equal, "Id");
    filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

    // Use Query overloads that take the minimum required query
parameters.
```

```
Search search = table.Query(filter);

do
{
    var documentSet = await search.GetNextSetAsync();
    Console.WriteLine("\nFindRepliesInLast15Days:
printing .....");

    foreach (var document in documentSet)
    {
        PrintDocument(document);
    }
}
while (!search.IsDone);
}

/// <summary>
/// Retrieve replies made during a specific time period.
/// </summary>
/// <param name="table">The table we want to query.</param>
/// <param name="forumName">The name of the forum that we're interested
in.</param>
/// <param name="threadSubject">The subject of the thread, which we are
/// searching for replies.</param>
public static async Task FindRepliesPostedWithinTimePeriod(
    Table table,
    string forumName,
    string threadSubject)
{
    DateTime startDate = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0,
0));
    DateTime endDate = DateTime.UtcNow.Subtract(new TimeSpan(1, 0, 0,
0));

    var filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
 "#" + threadSubject);
    filter.AddCondition("ReplyDateTime", QueryOperator.Between,
startDate, endDate);

    var config = new QueryOperationConfig()
    {
        Limit = 2, // 2 items/page.
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string>
```

```
{  
    "Message",  
    "ReplyDateTime",  
    "PostedBy",  
},  
        ConsistentRead = true,  
        Filter = filter,  
};  
  
Search search = table.Query(config);  
  
do  
{  
    var documentList = await search.GetNextSetAsync();  
    Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: printing  
replies posted within dates: {0} and {1} .....", startDate, endDate);  
  
    foreach (var document in documentList)  
    {  
        PrintDocument(document);  
    }  
}  
while (!search.IsDone);  
}  
  
/// <summary>  
/// Perform a query for replies made in the last 15 days using a DynamoDB  
/// QueryOperationConfig object.  
/// </summary>  
/// <param name="table">The table we want to query.</param>  
/// <param name="forumName">The name of the forum that we're interested  
in.</param>  
/// <param name="threadName">The bane of the thread that we are searching  
/// for replies.</param>  
public static async Task FindRepliesInLast15DaysWithConfig(  
    Table table,  
    string forumName,  
    string threadName)  
{  
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);  
    var filter = new QueryFilter("Id", QueryOperator.Equal, forumName +  
    "#" + threadName);  
    filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,  
    twoWeeksAgoDate);  
}
```

```
var config = new QueryOperationConfig()
{
    Filter = filter,

    // Optional parameters.
    Select = SelectValues.SpecificAttributes,
    AttributesToGet = new List<string>
    {
        "Message",
        "ReplyDateTime",
        "PostedBy",
    },
    ConsistentRead = true,
};

Search search = table.Query(config);

do
{
    var documentSet = await search.GetNextSetAsync();
    Console.WriteLine("\nFindRepliesInLast15DaysWithConfig:
printing .....");

    foreach (var document in documentSet)
    {
        PrintDocument(document);
    }
}
while (!search.IsDone);
}

/// <summary>
/// Displays the contents of the passed DynamoDB document on the console.
/// </summary>
/// <param name="document">A DynamoDB document to display.</param>
public static void PrintDocument(Document document)
{
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
```

```
        if (value is Primitive)
        {
            stringValue = value.AsPrimitive().Value.ToString();
        }
        else if (value is PrimitiveList)
        {
            stringValue = string.Join(", ", (from primitive
                in value.AsPrimitiveList().Entries
                select
                    primitive.Value).ToArray());
        }

        Console.WriteLine($"{attribute} - {stringValue}");
    }
}
```

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use a high-level object persistence model for DynamoDB using an AWS SDK

The following code example shows how to perform Create, Read, Update, and Delete (CRUD) and batch operations using an object persistence model for DynamoDB and an AWS SDK.

For more information, see [Object persistence model](#).

.NET

AWS SDK for .NET

 **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Perform CRUD operations using a high-level object persistence model.

```
/// <summary>
/// Shows how to perform high-level CRUD operations on an Amazon DynamoDB
/// table.
/// </summary>
public class HighLevelItemCrud
{
    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();
        DynamoDBContext context = new DynamoDBContext(client);
        await PerformCRUDOperations(context);
    }

    public static async Task PerformCRUDOperations(IDynamoDBContext context)
    {
        int bookId = 1001; // Some unique value.
        Book myBook = new Book
        {
            Id = bookId,
            Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
            Isbn = "111-1111111001",
            BookAuthors = new List<string> { "Author 1", "Author 2" },
        };

        // Save the book to the ProductCatalog table.
        await context.SaveAsync(myBook);

        // Retrieve the book from the ProductCatalog table.
        Book bookRetrieved = await context.LoadAsync<Book>(bookId);

        // Update some properties.
        bookRetrieved.Isbn = "222-2222221001";

        // Update existing authors list with the following values.
        bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author
x" };
        await context.SaveAsync(bookRetrieved);

        // Retrieve the updated book. This time, add the optional
        // ConsistentRead parameter using DynamoDBContextConfig object.
        await context.LoadAsync<Book>(bookId, new DynamoDBContextConfig
```

```
{  
    ConsistentRead = true,  
});  
  
// Delete the book.  
await context.DeleteAsync<Book>(bookId);  
  
// Try to retrieve deleted book. It should return null.  
Book deletedBook = await context.LoadAsync<Book>(bookId, new  
DynamoDBContextConfig  
{  
    ConsistentRead = true,  
});  
  
if (deletedBook == null)  
{  
    Console.WriteLine("Book is deleted");  
}  
}  
}  
}
```

Perform batch write operations using a high-level object persistence model.

```
/// <summary>  
/// Performs high-level batch write operations to an Amazon DynamoDB table.  
/// This example was written using the AWS SDK for .NET version 3.7 and .NET  
/// Core 5.0.  
/// </summary>  
public class HighLevelBatchWriteItem  
{  
    public static async Task SingleTableBatchWrite(IDynamoDBContext context)  
    {  
        Book book1 = new Book  
        {  
            Id = 902,  
            InPublication = true,  
            Isbn = "902-11-11-1111",  
            PageCount = "100",  
            Price = 10,  
            ProductCategory = "Book",  
        };  
        Book book2 = new Book  
        {  
            Id = 903,  
            InPublication = true,  
            Isbn = "903-11-11-1111",  
            PageCount = "200",  
            Price = 20,  
            ProductCategory = "Book",  
        };  
        Book book3 = new Book  
        {  
            Id = 904,  
            InPublication = true,  
            Isbn = "904-11-11-1111",  
            PageCount = "300",  
            Price = 30,  
            ProductCategory = "Book",  
        };  
        await context.BatchWriteAsync(new[] { book1, book2, book3 });  
    }  
}
```

```
        Title = "My book3 in batch write",
    };

    Book book2 = new Book
    {
        Id = 903,
        InPublication = true,
        Isbn = "903-11-11-1111",
        PageCount = "200",
        Price = 10,
        ProductCategory = "Book",
        Title = "My book4 in batch write",
    };

    var bookBatch = context.CreateBatchWrite<Book>();
    bookBatch.AddPutItems(new List<Book> { book1, book2 });

    Console.WriteLine("Adding two books to ProductCatalog table.");
    await bookBatch.ExecuteAsync();
}

public static async Task MultiTableBatchWrite(IDynamoDBContext context)
{
    // New Forum item.
    Forum newForum = new Forum
    {
        Name = "Test BatchWrite Forum",
        Threads = 0,
    };
    var forumBatch = context.CreateBatchWrite<Forum>();
    forumBatch.AddPutItem(newForum);

    // New Thread item.
    Thread newThread = new Thread
    {
        ForumName = "S3 forum",
        Subject = "My sample question",
        KeywordTags = new List<string> { "S3", "Bucket" },
        Message = "Message text",
    };

    DynamoDBOperationConfig config = new DynamoDBOperationConfig();
    config.SkipVersionCheck = true;
    var threadBatch = context.CreateBatchWrite<Thread>(config);
```

```
        threadBatch.AddPutItem(newThread);
        threadBatch.AddDeleteKey("some partition key value", "some sort key
value");

        var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);

        Console.WriteLine("Performing batch write in
MultiTableBatchWrite().");
        await superBatch.ExecuteAsync();
    }

    public static async Task Main()
{
    AmazonDynamoDBClient client = new AmazonDynamoDBClient();
    DynamoDBContext context = new DynamoDBContext(client);

    await SingleTableBatchWrite(context);
    await MultiTableBatchWrite(context);
}
}
```

Map arbitrary data to a table using a high-level object persistence model.

```
/// <summary>
/// Shows how to map arbitrary data to an Amazon DynamoDB table.
/// </summary>
public class HighLevelMappingArbitraryData
{
    /// <summary>
    /// Creates a book, adds it to the DynamoDB ProductCatalog table,
retrieves
    /// the new book from the table, updates the dimensions and writes the
    /// changed item back to the table.
    /// </summary>
    /// <param name="context">The DynamoDB context object used to write and
    /// read data from the table.</param>
    public static async Task AddRetrieveUpdateBook(IDynamoDBContext context)
    {
        // Create a book.
        DimensionType myBookDimensions = new DimensionType()
```

```
        {
            Length = 8M,
            Height = 11M,
            Thickness = 0.5M,
        };

        Book myBook = new Book
        {
            Id = 501,
            Title = "AWS SDK for .NET Object Persistence Model Handling
Arbitrary Data",
            Isbn = "999-9999999999",
            BookAuthors = new List<string> { "Author 1", "Author 2" },
            Dimensions = myBookDimensions,
        };

        // Add the book to the DynamoDB table ProductCatalog.
        await context.SaveAsync(myBook);

        // Retrieve the book.
        Book bookRetrieved = await context.LoadAsync<Book>(501);

        // Update the book dimensions property.
        bookRetrieved.Dimensions.Height += 1;
        bookRetrieved.Dimensions.Length += 1;
        bookRetrieved.Dimensions.Thickness += 0.2M;

        // Write the changed item to the table.
        await context.SaveAsync(bookRetrieved);
    }

    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();
        DynamoDBContext context = new DynamoDBContext(client);
        await AddRetrieveUpdateBook(context);
    }
}
```

Query and scan a table using a high-level object persistence model.

```
/// <summary>
/// Shows how to perform high-level query and scan operations to Amazon
/// DynamoDB tables.
/// </summary>
public class HighLevelQueryAndScan
{
    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();

        DynamoDBContext context = new DynamoDBContext(client);

        // Get an item.
        await GetBook(context, 101);

        // Sample forum and thread to test queries.
        string forumName = "Amazon DynamoDB";
        string threadSubject = "DynamoDB Thread 1";

        // Sample queries.
        await FindRepliesInLast15Days(context, forumName, threadSubject);
        await FindRepliesPostedWithinTimePeriod(context, forumName,
threadSubject);

        // Scan table.
        await FindProductsPricedLessThanZero(context);
    }

    public static async Task GetBook(IDynamoDBContext context, int productId)
    {
        Book bookItem = await context.LoadAsync<Book>(productId);

        Console.WriteLine("\nGetBook: Printing result.....");
        Console.WriteLine($"Title: {bookItem.Title} \n ISBN:{bookItem.Isbn}\n No. of pages: {bookItem.PageCount}");
    }

    /// <summary>
    /// Queries a DynamoDB table to find replies posted within the last 15
    /// days.
    /// </summary>
```

```
    /// <param name="context">The DynamoDB context used to perform the
    query.</param>
    /// <param name="forumName">The name of the forum that we're interested
    in.</param>
    /// <param name="threadSubject">The thread object containing the query
    parameters.</param>
    public static async Task FindRepliesInLast15Days(
        IDynamoDBContext context,
        string forumName,
        string threadSubject)
    {
        string replyId = $"{forumName} #{threadSubject}";
        DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);

        List<object> times = new List<object>();
        times.Add(twoWeeksAgoDate);

        List<ScanCondition> scs = new List<ScanCondition>();
        var sc = new ScanCondition("PostedBy", ScanOperator.GreaterThan,
times.ToArray());
        scs.Add(sc);

        var cfg = new DynamoDBOperationConfig
        {
            QueryFilter = scs,
        };

        AsyncSearch<Reply> response = context.QueryAsync<Reply>(replyId,
cfg);
        IEnumerable<Reply> latestReplies = await
response.GetRemainingAsync();

        Console.WriteLine("\nReplies in last 15 days:");

        foreach (Reply r in latestReplies)
        {

Console.WriteLine($"{r.Id}\t{r.PostedBy}\t{r.Message}\t{r.ReplyDateTime}");
        }
    }

    /// <summary>
    /// Queries for replies posted within a specific time period.
    /// </summary>
```

```
    /// <param name="context">The DynamoDB context used to perform the
    query.</param>
    /// <param name="forumName">The name of the forum that we're interested
    in.</param>
    /// <param name="threadSubject">Information about the subject that we're
    /// interested in.</param>
    public static async Task FindRepliesPostedWithinTimePeriod(
        IDynamoDBContext context,
        string forumName,
        string threadSubject)
    {
        string forumId = forumName + "#" + threadSubject;
        Console.WriteLine("\nReplies posted within time period:");

        DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(30);
        DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(1);

        List<object> times = new List<object>();
        times.Add(startDate);
        times.Add(endDate);

        List<ScanCondition> scs = new List<ScanCondition>();
        var sc = new ScanCondition("LastPostedBy", ScanOperator.Between,
times.ToArray());
        scs.Add(sc);

        var cfg = new DynamoDBOperationConfig
        {
            QueryFilter = scs,
        };

        AsyncSearch<Reply> response = context.QueryAsync<Reply>(forumId,
cfg);
        IEnumerable<Reply> repliesInAPeriod = await
response.GetRemainingAsync();

        foreach (Reply r in repliesInAPeriod)
        {

Console.WriteLine("{r.Id}\t{r.PostedBy}\t{r.Message}\t{r.ReplyDateTime}");
        }
    }

    /// <summary>
```

```
/// Queries the DynamoDB ProductCatalog table for products costing less
/// than zero.
/// </summary>
/// <param name="context">The DynamoDB context object used to perform the
/// query.</param>
public static async Task FindProductsPricedLessThanZero(IDynamoDBContext
context)
{
    int price = 0;

    List<ScanCondition> scs = new List<ScanCondition>();
    var sc1 = new ScanCondition("Price", ScanOperator.LessThan, price);
    var sc2 = new ScanCondition("ProductCategory", ScanOperator.Equal,
    "Book");
    scs.Add(sc1);
    scs.Add(sc2);

    AsyncSearch<Book> response = context.ScanAsync<Book>(scs);

    IEnumerable<Book> itemsWithWrongPrice = await
response.GetRemainingAsync();

    Console.WriteLine("\nFindProductsPricedLessThanZero: Printing
result.....");

    foreach (Book r in itemsWithWrongPrice)
    {
        Console.WriteLine($"{r.Id}\t{r.Title}\t{r.Price}\t{r.Isbn}");
    }
}
```

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Cross-service examples for DynamoDB using AWS SDKs

The following sample applications use AWS SDKs to combine DynamoDB with other AWS services. Each example includes a link to GitHub, where you can find instructions on how to set up and run the application.

Examples

- [Build an application to submit data to a DynamoDB table](#)
- [Create an API Gateway REST API to track COVID-19 data](#)
- [Create a messenger application with Step Functions](#)
- [Create a photo asset management application that lets users manage photos using labels](#)
- [Create a web application to track DynamoDB data](#)
- [Create a websocket chat application with API Gateway](#)
- [Detect PPE in images with Amazon Rekognition using an AWS SDK](#)
- [Invoke a Lambda function from a browser](#)
- [Save EXIF and other image information using an AWS SDK](#)
- [Use API Gateway to invoke a Lambda function](#)
- [Use Step Functions to invoke Lambda functions](#)
- [Use scheduled events to invoke a Lambda function](#)

Build an application to submit data to a DynamoDB table

The following code examples show how to build an application that submits data to an Amazon DynamoDB table and notifies you when a user updates the table.

Java

SDK for Java 2.x

Shows how to create a dynamic web application that submits data using the Amazon DynamoDB Java API and sends a text message using the Amazon Simple Notification Service Java API.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

This example shows how to build an app that enables users to submit data to an Amazon DynamoDB table, and send a text message to the administrator using Amazon Simple Notification Service (Amazon SNS).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- Amazon SNS

Kotlin

SDK for Kotlin

Shows how to create a native Android application that submits data using the Amazon DynamoDB Kotlin API and sends a text message using the Amazon SNS Kotlin API.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Create an API Gateway REST API to track COVID-19 data

The following code example shows how to create a REST API that simulates a system to track daily cases of COVID-19 in the United States, using fictional data.

Python

SDK for Python (Boto3)

Shows how to use AWS Chalice with the AWS SDK for Python (Boto3) to create a serverless REST API that uses Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. The REST API simulates a system that tracks daily cases of COVID-19 in the United States, using fictional data. Learn how to:

- Use AWS Chalice to define routes in Lambda functions that are called to handle REST requests that come through API Gateway.
- Use Lambda functions to retrieve and store data in a DynamoDB table to serve REST requests.
- Define table structure and security role resources in an AWS CloudFormation template.
- Use AWS Chalice and CloudFormation to package and deploy all necessary resources.
- Use CloudFormation to clean up all created resources.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Create a messenger application with Step Functions

The following code example shows how to create an AWS Step Functions messenger application that retrieves message records from a database table.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with AWS Step Functions to create a messenger application that retrieves message records from an Amazon DynamoDB table and sends them with Amazon Simple Queue Service (Amazon SQS). The state machine integrates with an AWS Lambda function to scan the database for unsent messages.

- Create a state machine that retrieves and updates message records from an Amazon DynamoDB table.
- Update the state machine definition to also send messages to Amazon Simple Queue Service (Amazon SQS).
- Start and stop state machine runs.
- Connect to Lambda, DynamoDB, and Amazon SQS from a state machine by using service integrations.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda
- Amazon SQS
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Create a photo asset management application that lets users manage photos using labels

The following code examples show how to create a serverless application that lets users manage photos using labels.

.NET

AWS SDK for .NET

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK for PHP

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK for Rust

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Create a web application to track DynamoDB data

The following code examples show how to create a web application that tracks work items in an Amazon DynamoDB table and uses Amazon Simple Email Service (Amazon SES) to send reports.

.NET

AWS SDK for .NET

Shows how to use the Amazon DynamoDB .NET API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SES

Java

SDK for Java 2.x

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SES

JavaScript

SDK for JavaScript (v3)

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SES

Kotlin

SDK for Kotlin

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SES

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) to create a REST service that tracks work items in Amazon DynamoDB and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses the Flask web framework to handle HTTP routing and integrates with a React webpage to present a fully functional web application.

- Build a Flask REST service that integrates with AWS services.
- Read, write, and update work items that are stored in a DynamoDB table.
- Use Amazon SES to send email reports of work items.

For complete source code and instructions on how to set up and run, see the full example in the [AWS Code Examples Repository](#) on GitHub.

Services used in this example

- DynamoDB

- Amazon SES

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Create a websocket chat application with API Gateway

The following code example shows how to create a chat application that is served by a websocket API built on Amazon API Gateway.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with Amazon API Gateway V2 to create a websocket API that integrates with AWS Lambda and Amazon DynamoDB.

- Create a websocket API served by API Gateway.
- Define a Lambda handler that stores connections in DynamoDB and posts messages to other chat participants.
- Connect to the websocket chat application and send messages with the Websockets package.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Detect PPE in images with Amazon Rekognition using an AWS SDK

The following code examples show how to build an app that uses Amazon Rekognition to detect Personal Protective Equipment (PPE) in images.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function that detects images with Personal Protective Equipment.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

JavaScript

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an application to detect personal protective equipment (PPE) in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app saves the results to an Amazon DynamoDB table, and sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for PPE using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Update a DynamoDB table with results.

- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a browser

The following code example shows how to invoke an AWS Lambda function from a browser.

JavaScript

SDK for JavaScript (v2)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

SDK for JavaScript (v3)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Save EXIF and other image information using an AWS SDK

The following code example shows how to:

- Get EXIF information from a a JPG, JPEG, or PNG file.
- Upload the image file to an Amazon S3 bucket.
- Use Amazon Rekognition to identify the three top attributes (*labels*) in the file.
- Add the EXIF and label information to an Amazon DynamoDB table in the Region.

Rust

SDK for Rust

Get EXIF information from a JPG, JPEG, or PNG file, upload the image file to an Amazon S3 bucket, use Amazon Rekognition to identify the three top attributes (*labels* in Amazon Rekognition) in the file, and add the EXIF and label information to a Amazon DynamoDB table in the Region.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use API Gateway to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by Amazon API Gateway.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification

Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use Step Functions to invoke Lambda functions

The following code examples show how to create an AWS Step Functions state machine that invokes AWS Lambda functions in sequence.

Java

SDK for Java 2.x

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for Java 2.x. Each workflow step is implemented using an AWS Lambda function.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

JavaScript

SDK for JavaScript (v3)

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for JavaScript. Each workflow step is implemented using an AWS Lambda function.

Lambda is a compute service that enables you to run code without provisioning or managing servers. Step Functions is a serverless orchestration service that lets you combine Lambda functions and other AWS services to build business-critical applications.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use scheduled events to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

Java

SDK for Java 2.x

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case.

This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see [Using DynamoDB with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Security and compliance in Amazon DynamoDB

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the [AWS compliance programs](#). To learn about the compliance programs that apply to DynamoDB, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using DynamoDB. The following topics show you how to configure DynamoDB to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your DynamoDB resources.

Topics

- [AWS managed policies for Amazon DynamoDB](#)
- [Using resource-based policies for DynamoDB](#)
- [Data protection in DynamoDB](#)
- [AWS Identity and Access Management \(IAM\)](#)
- [Compliance validation by industry for DynamoDB](#)
- [Resilience and disaster recovery in Amazon DynamoDB](#)
- [Infrastructure security in Amazon DynamoDB](#)
- [AWS PrivateLink for DynamoDB](#)
- [Configuration and vulnerability analysis in Amazon DynamoDB](#)
- [Security best practices for Amazon DynamoDB](#)

AWS managed policies for Amazon DynamoDB

DynamoDB uses AWS managed policies to define a set of permissions the service needs to perform specific actions. DynamoDB maintains and updates its AWS managed policies. You can't change the permissions in AWS managed policies. For more information about AWS managed policies, see [AWS managed policies](#) in the IAM User Guide.

DynamoDB may occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. An AWS managed policy is most likely to be updated when a new feature is launched or when new operations become available. DynamoDB will not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

AWS managed policy: `DynamoDBReplicationServiceRolePolicy`

You can't attach the `DynamoDBReplicationServiceRolePolicy` policy to your IAM entities. This policy is attached to a service-linked role that allows DynamoDB to perform actions on your behalf. For more information, see [Using IAM with global tables](#).

This policy grants permissions that allow the service-linked role to perform data replication between global table replicas. It also grants administrative permissions to manage global table replicas on your behalf.

Permissions details

This policy grants permissions to do the following:

- `dynamodb` – Perform data replication and manage table replicas.
- `application-autoscaling` – Retrieve and manage table AutoScaling settings
- `account` – Retrieve region status for evaluating replica accessibility.
- `iam` – To create the service-linked role for application AutoScaling in the event that the service-linked role does not already exist.

The definition of this managed policy can be found [here](#).

AWS managed policy: `AmazonDynamoDBReadOnlyAccess`

You can attach the `AmazonDynamoDBReadOnlyAccess` policy to your IAM identities.

This policy grants read-only access to Amazon DynamoDB.

Permissions details

This policy includes the following permissions:

- Amazon DynamoDB – Provides read-only access to Amazon DynamoDB.
- Amazon DynamoDB Accelerator (DAX) – Provides read-only access to Amazon DynamoDB Accelerator (DAX).
- Application Auto Scaling – Allows principals to view configurations from Application Auto Scaling. This is required so that users can view automatic scaling policies that are attached to a table.
- CloudWatch – Allows principals to view metric data and alarms configured in CloudWatch. This is required so users can view the billable table size and CloudWatch alarms that have been configured for a table.
- AWS Data Pipeline – Allows principals to view AWS Data Pipeline and associated objects.
- Amazon EC2 – Allows principals to view Amazon EC2 VPCs, subnets, and security groups.
- IAM – Allows principals to view IAM roles.
- AWS KMS – Allows principals to view keys configured in AWS KMS. This is required so users can view AWS KMS keys that they create and manage in their account.
- Amazon SNS – Allows principals to list Amazon SNS topics and and subscriptions by topic.
- AWS Resource Groups – Allows principals to view resource groups and their queries.
- AWS Resource Groups Tagging – Allows principals to list all the tagged or previously tagged resources in a Region.
- Kinesis – Allows principals to view Kinesis data streams descriptions.
- Amazon CloudWatch Contributor Insights – Allow principals to view time series data collected by Contributor Insights rules.

To review the policy in JSON format, see [AmazonDynamoDBReadOnlyAccess](#).

DynamoDB updates to AWS managed policies

This table shows updates to the AWS access management policies for DynamoDB.

Change	Description	Date Changed
AmazonDynamoDBReadOnlyAccess update to an existing policy	AmazonDynamoDBReadOnlyAccess added the permission dynamodb:GetResourcePolicy . This permission provides access to read resource-based policies attached to DynamoDB resources.	March 20, 2024
DynamoDBReplicationServiceRolePolicy update to an existing policy	DynamoDBReplicationServiceRolePolicy added the permission dynamodb:GetResourcePolicy . This permission allows the service-linked role to read resource-based policies attached to DynamoDB resources.	December 15, 2023
DynamoDBReplicationServiceRolePolicy update to an existing policy	DynamoDBReplicationServiceRolePolicy added the permission account:ListRegions . This permission allows the service-linked role to evaluate replica accessibility	May 10, 2023
DynamoDBReplicationServiceRolePolicy added to list of managed policies	Added information about the managed policy DynamoDBReplicationServiceRolePolicy , which is used by the DynamoDB global tables service-linked role.	May 10, 2023
DynamoDB global tables started tracking changes	DynamoDB global tables started tracking changes for its AWS managed policies.	May 10, 2023

Using resource-based policies for DynamoDB

DynamoDB supports resource-based policies for tables, indexes, and streams. Resource-based policies let you define access permissions by specifying who has access to each resource, and the actions they are allowed to perform on each resource.

You can attach a resource-based policy to DynamoDB resources, such as a table or a stream. In this policy, you specify permissions for Identity and Access Management (IAM) [principals](#) that can perform specific actions on these DynamoDB resources. For example, the policy attached to a table will contain permissions for access to the table and its indexes. As a result, resource-based policies can help you simplify access control for your DynamoDB tables, indexes, and streams, by defining permissions at the resource level. The maximum size of a policy you can attach to a DynamoDB resource is 20 KB.

A significant benefit of using resource-based policies is to simplify cross-account access control for providing cross-account access to IAM principals in different AWS accounts. For more information, see [Resource-based policy for cross-account access](#).

Resource-based policies also support integrations with [IAM Access Analyzer](#) external access analyzer and [Block Public Access \(BPA\)](#) capabilities. IAM Access Analyzer reports cross-account access to external entities specified in resource-based policies. It also provides visibility to help you refine permissions and conform to the least privilege principle. BPA helps you prevent public access to your DynamoDB tables, indexes, and streams, and is automatically enabled in the resource-based policies creation and modification workflows.

Topics

- [Create a table with a resource-based policy](#)
- [Attach a policy to an existing table](#)
- [Attach a resource-based policy to a stream](#)
- [Remove a resource-based policy from a table](#)
- [Cross-account access with resource-based policies](#)
- [Blocking public access with resource-based policies](#)
- [IAM actions supported by resource-based policies](#)
- [Authorization with IAM identity-based policies and DynamoDB resource-based policies](#)
- [Resource-based policy examples](#)
- [Resource-based policy considerations](#)

- [Resource-based policy best practices](#)

Create a table with a resource-based policy

You can add a resource-based policy while you create a table by using the DynamoDB console, [CreateTable](#) API, AWS CLI, [AWS SDK](#), or an AWS CloudFormation template.

AWS CLI

The following example creates a table named *MusicCollection* using the `create-table` AWS CLI command. This command also includes the `resource-policy` parameter that adds a resource-based policy to the table. This policy allows the user *John* to perform the [RestoreTableToPointInTime](#), [GetItem](#), and [PutItem](#) API actions on the table.

Remember to replace the *italicized* text with your resource-specific information.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S
    AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
    AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --resource-policy \
    "{"
        \\"Version\\": \\"2012-10-17\",
        \\"Statement\\": [
            {
                \\"Effect\\": \\"Allow\",
                \\"Principal\\": {
                    \\"AWS\\": \\"arn:aws:iam::123456789012:user/John\"
                },
                \\"Action\\": [
                    \\"dynamodb:RestoreTableToPointInTime\",
                    \\"dynamodb:GetItem\",
                    \\"dynamodb:DescribeTable\"
                ],
                \\"Resource\\": \\"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
            }
        ]
    }"
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. On the dashboard, choose **Create table**.
3. In **Table details**, enter the table name, partition key, and sort key details.
4. In **Table settings**, choose **Customize settings**.
5. (Optional) Specify your options for **Table class**, **Capacity calculator**, **Read/write capacity settings**, **Secondary indexes**, **Encryption at rest**, and **Deletion protection**.
6. In **Resource-based policy**, add a policy to define the access permissions for the table and its indexes. In this policy, you specify who has access to these resources, and the actions they are allowed to perform on each resource. To add a policy, do one of the following:
 - Type or paste a JSON policy document. For details about the IAM policy language, see [Creating policies using the JSON editor](#) in the *IAM User Guide*.

 **Tip**

To see examples of resource-based policies in the Amazon DynamoDB Developer Guide, choose **Policy examples**.

- Choose **Add new statement** to add a new statement and enter the information in the provided fields. Repeat this step for as many statements as you would like to add.

 **Important**

Make sure that you resolve any security warnings, errors, or suggestions before you save your policy.

The following IAM policy example allows the user *John* to perform the [RestoreTableToPointInTime](#), [GetItem](#), and [PutItem](#) API actions on the table *MusicCollection*.

Remember to replace the *italicized* text with your resource-specific information.

```
{
```

```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::123456789012:user/John"
        },
        "Action": [
            "dynamodb:RestoreTableToPointInTime",
            "dynamodb:GetItem",
            "dynamodb:PutItem"
        ],
        "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection"
    }
]
}
```

7. (Optional) Choose **Preview external access** in the lower-right corner to preview how your new policy affects public and cross-account access to your resource. Before you save your policy, you can check whether it introduces new IAM Access Analyzer findings or resolves existing findings. If you don't see an active analyzer, choose **Go to Access Analyzer** to [create an account analyzer](#) in IAM Access Analyzer. For more information, see [Preview access](#).
8. Choose **Create table**.

AWS CloudFormation template

Using the AWS::DynamoDB::Table resource

The following CloudFormation template creates a table with a stream using the [AWS::DynamoDB::Table](#) resource. This template also includes resource-based policies that are attached to both the table and the stream.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "MusicCollectionTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "Artist",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
                }
            }
        }
    }
}
```

```
        }
    ],
    "KeySchema": [
        {
            "AttributeName": "Artist",
            "KeyType": "HASH"
        }
    ],
    "BillingMode": "PROVISIONED",
    "ProvisionedThroughput": {
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    },
    "StreamSpecification": {
        "StreamViewType": "OLD_IMAGE",
        "ResourcePolicy": {
            "PolicyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Principal": {
                            "AWS": "arn:aws:iam::111122223333:user/John"
                        },
                        "Effect": "Allow",
                        "Action": [
                            "dynamodb:GetRecords",
                            "dynamodb:GetShardIterator",
                            "dynamodb:DescribeStream"
                        ],
                        "Resource": "*"
                    }
                ]
            }
        }
    },
    "TableName": "MusicCollection",
    "ResourcePolicy": {
        "PolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Principal": {
                        "AWS": [
                            "arn:aws:iam::111122223333:user/John"
                        ]
                    }
                }
            ]
        }
    }
},
```

```
        ]
    },
    "Effect": "Allow",
    "Action": "dynamodb:GetItem",
    "Resource": "*"
}
]
}
}
}
}
}
```

Using the AWS::DynamoDB::GlobalTable resource

The following CloudFormation template creates a table with the [AWS::DynamoDB::GlobalTable](#) resource and attaches a resource-based policy to the table and its stream.

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Resources": {
    "GlobalMusicCollection": {
      "Type": "AWS::DynamoDB::GlobalTable",
      "Properties": {
        "TableName": "MusicCollection",
        "AttributeDefinitions": [
          {
            "AttributeName": "Artist",
            "AttributeType": "S"
          }
        ],
        "KeySchema": [
          {
            "AttributeName": "Artist",
            "KeyType": "HASH"
          }
        ],
        "BillingMode": "PAY_PER_REQUEST",
        "StreamSpecification": {
          "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "Replicas": [
          {
            "Region": "us-east-1",
            "ResourcePolicy": {

```

```
        "PolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Principal": {
                        "AWS": [
                            "arn:aws:iam::111122223333:user/John"
                        ]
                    },
                    "Effect": "Allow",
                    "Action": "dynamodb:GetItem",
                    "Resource": "*"
                }
            ],
            "ReplicaStreamSpecification": {
                "ResourcePolicy": {
                    "PolicyDocument": {
                        "Version": "2012-10-17",
                        "Statement": [
                            {
                                "Principal": {
                                    "AWS": [
                                        "arn:aws:iam::111122223333:user/John"
                                    ],
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:GetRecords",
                                        "dynamodb:GetShardIterator",
                                        "dynamodb:DescribeStream"
                                    ],
                                    "Resource": "*"
                                }
                            }
                        ]
                    }
                }
            }
        }
    }
}
```

Attach a policy to an existing table

You can attach a resource-based policy to an existing table or modify an existing policy by using the DynamoDB console, [PutResourcePolicy](#) API, the AWS CLI, AWS SDK, or an [AWS CloudFormation template](#).

AWS CLI example to attach a new policy

The following IAM policy example uses the `put-resource-policy` AWS CLI command to attach a resource-based policy to an existing table. This example allows the user *John* to perform the [GetItem](#), [PutItem](#), [UpdateItem](#), and [UpdateTable](#) API actions on an existing table named *MusicCollection*.

Remember to replace the *italicized* text with your resource-specific information.

```
aws dynamodb put-resource-policy \
--resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
--policy \
"{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
        {
            \"Effect\": \"Allow\",
            \"Principal\": {
                \"AWS\": \"arn:aws:iam::111122223333:user/John\"
            },
            \"Action\": [
                \"dynamodb:GetItem\",
                \"dynamodb:PutItem\",
                \"dynamodb:UpdateItem\",
                \"dynamodb:UpdateTable\"
            ],
            \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
        }
    ]
}"
```

AWS CLI example to conditionally update an existing policy

To conditionally update an existing resource-based policy of a table, you can use the optional `expected-revision-id` parameter. The following example will only update the policy if it

exists in DynamoDB and its current revision ID matches the provided `expected-revision-id` parameter.

```
aws dynamodb put-resource-policy \
--resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
--expected-revision-id 1709841168699 \
--policy \
{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
        {
            \"Effect\": \"Allow\",
            \"Principal\": {
                \"AWS\": \"arn:aws:iam::111122223333:user/John\"
            },
            \"Action\": [
                \"dynamodb:GetItem\",
                \"dynamodb:UpdateItem\",
                \"dynamodb:UpdateTable\"
            ],
            \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
        }
    ]
}"
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. From the dashboard, choose an existing table.
3. Navigate to the **Permissions** tab, and choose **Create table policy**.
4. In the resource-based policy editor, add the policy you would like to attach and choose **Create policy**.

The following IAM policy example allows the user *John* to perform the [GetItem](#), [PutItem](#), [UpdateItem](#), and [UpdateTable](#) API actions on an existing table named *MusicCollection*.

Remember to replace the *italicized* text with your resource-specific information.

{

```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::111122223333:user/John"
        },
        "Action": [
            "dynamodb:GetItem",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:UpdateTable"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    }
]
```

AWS SDK for Java 2.x

The following IAM policy example uses the `putResourcePolicy` method to attach a resource-based policy to an existing table. This policy allows a user to perform the [GetItem](#) API action on an existing table.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutResourcePolicyRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * Get started with the AWS SDK for Java 2.x
 */
public class PutResourcePolicy {

    public static void main(String[] args) {
        final String usage = """
```

```
Usage:  
    <tableArn> <allowedAWSPrincipal>  
  
Where:  
    tableArn - The Amazon DynamoDB table ARN to attach the policy to.  
For example, arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection.  
    allowedAWSPrincipal - Allowed AWS principal ARN that the example  
policy will give access to. For example, arn:aws:iam::123456789012:user/John.  
    """;  
  
if (args.length != 2) {  
    System.out.println(usage);  
    System.exit(1);  
}  
  
String tableArn = args[0];  
String allowedAWSPrincipal = args[1];  
System.out.println("Attaching a resource-based policy to the Amazon DynamoDB  
table with ARN " +  
    tableArn);  
Region region = Region.US_WEST_2;  
DynamoDbClient ddb = DynamoDbClient.builder()  
    .region(region)  
    .build();  
  
String result = putResourcePolicy(ddb, tableArn, allowedAWSPrincipal);  
System.out.println("Revision ID for the attached policy is " + result);  
ddb.close();  
}  
  
public static String putResourcePolicy(DynamoDbClient ddb, String tableArn, String  
allowedAWSPrincipal) {  
    String policy = generatePolicy(tableArn, allowedAWSPrincipal);  
    PutResourcePolicyRequest request = PutResourcePolicyRequest.builder()  
        .policy(policy)  
        .resourceArn(tableArn)  
        .build();  
  
    try {  
        return ddb.putResourcePolicy(request).revisionId();  
    } catch (DynamoDbException e) {  
        System.err.println(e.getMessage());  
        System.exit(1);  
    }
```

```
        return "";
    }

private static String generatePolicy(String tableArn, String allowedAWSPrincipal) {
    return "{\n" +
        "    \"Version\": \"2012-10-17\",\\n" +
        "    \"Statement\": [\n" +
        "        {\n" +
        "            \"Effect\": \"Allow\",\\n" +
        "            \"Principal\": {\"AWS\":\"" + allowedAWSPrincipal + "\"},\\n" +
        "            \"Action\": [\n" +
        "                \"dynamodb:GetItem\\n\" +
        "            ],\\n" +
        "            \"Resource\": \"\" + tableArn + "\"\\n" +
        "        }\n" +
        "    ]\\n" +
        "}";
}
```

Attach a resource-based policy to a stream

You can attach a resource-based policy to an existing table's stream or modify an existing policy by using the DynamoDB console, [PutResourcePolicy](#) API, the AWS CLI, AWS SDK, or an [AWS CloudFormation template](#).

 **Note**

You can't attach a policy to a stream while creating it using the [CreateTable](#) or [UpdateTable](#) APIs. However, you can modify or delete a policy after a table is deleted. You can also modify or delete the policy of a disabled stream.

AWS CLI

The following IAM policy example uses the `put-resource-policy` AWS CLI command to attach a resource-based policy to the stream of a table named *MusicCollection*. This example allows the user *John* to perform the [GetRecords](#), [GetShardIterator](#), and [DescribeStream](#) API actions on the stream.

Remember to replace the *italicized* text with your resource-specific information.

```
aws dynamodb put-resource-policy \
--resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492 \
--policy \
"{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
        {
            \"Effect\": \"Allow\",
            \"Principal\": {
                \"AWS\": \"arn:aws:iam::111122223333:user/John\"
            },
            \"Action\": [
                \"dynamodb:GetRecords\",
                \"dynamodb:GetShardIterator\",
                \"dynamodb:DescribeStream\"
            ],
            \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection/stream/2024-02-12T18:57:26.492\"
        }
    ]
}"
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. On the DynamoDB console dashboard, choose **Tables** and then select an existing table.

Make sure the table you select has streams turned on. For information about turning on streams for a table, see [Enabling a stream](#).

3. Choose the **Permissions** tab.
4. In **Resource-based policy for active stream**, choose **Create stream policy**.
5. In the **Resource-based policy** editor, add a policy to define the access permissions for the stream. In this policy, you specify who has access to the stream and the actions they are allowed to perform on the stream. To add a policy, do one of the following:

- Type or paste a JSON policy document. For details about the IAM policy language, see [Creating policies using the JSON editor](#) in the *IAM User Guide*.

 **Tip**

To see examples of resource-based policies in the Amazon DynamoDB Developer Guide, choose **Policy examples**.

- Choose **Add new statement** to add a new statement and enter the information in the provided fields. Repeat this step for as many statements as you would like to add.

 **Important**

Make sure that you resolve any security warnings, errors, or suggestions before you save your policy.

6. (Optional) Choose **Preview external access** in the lower-right corner to preview how your new policy affects public and cross-account access to your resource. Before you save your policy, you can check whether it introduces new IAM Access Analyzer findings or resolves existing findings. If you don't see an active analyzer, choose **Go to Access Analyzer** to [create an account analyzer](#) in IAM Access Analyzer. For more information, see [Preview access](#).
7. Choose **Create policy**.

The following IAM policy example attaches a resource-based policy to the stream of a table named *MusicCollection*. This example allows the user *John* to perform the [GetRecords](#), [GetShardIterator](#), and [DescribeStream](#) API actions on the stream.

Remember to replace the *italicized* text with your resource-specific information.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Principal": {  
        "AWS": "arn:aws:iam::111122223333:user/John"  
      },  
      "Action": [  
        "kinesis:DescribeStream",  
        "kinesis:GetRecords",  
        "kinesis:GetShardIterator"  
      ]  
    }  
  ]  
}
```

```
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator",
        "dynamodb:DescribeStream"
    ],
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492"
    ]
}
]
```

Remove a resource-based policy from a table

You can delete a resource-based policy from an existing table by using the DynamoDB console, [DeleteResourcePolicy](#) API, the AWS CLI, AWS SDK, or an AWS CloudFormation template.

AWS CLI

The following example uses the `delete-resource-policy` AWS CLI command to remove a resource-based policy from a table named *MusicCollection*.

Remember to replace the *italicized* text with your resource-specific information.

```
aws dynamodb delete-resource-policy \
--resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. On the DynamoDB console dashboard, choose **Tables** and then select an existing table.
3. Choose **Permissions**.
4. From the **Manage policy** dropdown, choose **Delete policy**.
5. In the **Delete resource-based policy for table** dialog box, type **confirm** to confirm the delete action.
6. Choose **Delete**.

Cross-account access with resource-based policies

Using a resource-based policy, you can provide cross-account access to resources available in different AWS accounts. All cross-account access allowed by the resource-based policies will be reported through IAM Access Analyzer external access findings if you have an analyzer in the same AWS Region as the resource. IAM Access Analyzer runs policy checks to validate your policy against IAM [policy grammar](#) and [best practices](#). These checks generate findings and provide actionable recommendations to help you author policies that are functional and conform to security best practices. You can view the active findings from IAM Access Analyzer in the **Permissions** tab of the [DynamoDB console](#).

For information about validating policies by using IAM Access Analyzer, see [IAM Access Analyzer policy validation](#) in the *IAM User Guide*. To view a list of the warnings, errors, and suggestions that are returned by IAM Access Analyzer, see [IAM Access Analyzer policy check reference](#).

To grant [GetItem](#) permission to a user A in account A for accessing a table B in account B, perform the following steps:

1. Attach a resource-based policy to table B that grants permission to user A for performing the [GetItem](#) action.
2. Attach an identity-based policy to user A that grants it permission to perform the [GetItem](#) action on table B.

Using the **Preview external access** option available in [DynamoDB console](#), you can preview how your new policy affects public and cross-account access to your resource. Before you save your policy, you can check whether it introduces new IAM Access Analyzer findings or resolves existing findings. If you don't see an active analyzer, choose **Go to Access Analyzer** to [create an account analyzer](#) in IAM Access Analyzer. For more information, see [Preview access](#).

The table name parameter in the DynamoDB data plane and control plane APIs accept complete Amazon Resource Name (ARN) of the table to support cross-account operations. If you only provide the table name parameter instead of a complete ARN, the API operation will be performed on the table in the account to which the requestor belongs. For an example of a policy that uses cross-account access, see [Resource-based policy for cross-account access](#).

The resource owner's account will be charged even when a principal from another account is reading from or writing to the DynamoDB table in the owner's account. If the table has provisioned throughput, the sum of all the requests from the owner accounts and the requestors in other

accounts will determine if the request will be throttled (if autoscaling is disabled) or scaled up/down if autoscaling is enabled.

The requests will be logged in the CloudTrail logs of both the owner and the requestor accounts so that each of the two accounts can track which account accessed what data.

 **Note**

The cross-account access of [control plane APIs](#) has a lower transactions per second (TPS) limit of 500 requests.

Blocking public access with resource-based policies

[Block Public Access \(BPA\)](#) is a feature that identifies and prevents the attaching of resource-based policies that grant public access to your DynamoDB tables, indexes, or streams across your [Amazon Web Services \(AWS\)](#) accounts. With BPA, you can prevent public access to your DynamoDB resources. BPA performs checks during the creation or modification of a resource-based policy and helps improve your security posture with DynamoDB.

BPA uses [automated reasoning](#) to analyze the access granted by your resource-based policy and alerts you if such permissions are found at the time of administering a resource-based policy. The analysis verifies access across all resource-based policy statements, actions, and the set of condition keys used in your policies.

 **Important**

BPA helps protect your resources by preventing public access from being granted through the resource-based policies that are directly attached to your DynamoDB resources, such as tables, indexes, and streams. In addition to using BPA, carefully inspect the following policies to confirm that they do not grant public access:

- Identity-based policies attached to associated AWS principals (for example, IAM roles)
- Resource-based policies attached to associated AWS resources (for example, AWS Key Management Service (KMS) keys)

You must ensure that the [principal](#) doesn't include a * entry or that one of the specified condition keys restrict access from principals to the resource. If the resource-based policy grants public access

to your table, indexes, or stream across AWS accounts, DynamoDB will block you from creating or modifying the policy until the specification within the policy is corrected and deemed non-public.

You can make a policy non-public by specifying one or more principals inside the Principal block. The following resource-based policy example blocks public access by specifying two principals.

```
{  
    "Effect": "Allow",  
    "Principal": {  
        "AWS": [  
            "123456789012",  
            "111122223333"  
        ]  
    },  
    "Action": "dynamodb:*",  
    "Resource": "*"  
}
```

Policies that restrict access by specifying certain condition keys are also not considered public. Along with evaluation of the principal specified in the resource-based policy, the following [trusted condition keys](#) are used to complete the evaluation of a resource-based policy for non-public access:

- aws:PrincipalAccount
- aws:PrincipalArn
- aws:PrincipalOrgID
- aws:PrincipalOrgPaths
- aws:SourceAccount
- aws:SourceArn
- aws:SourceVpc
- aws:SourceVpce
- aws:UserId
- aws:PrincipalServiceName
- aws:PrincipalServiceNamesList
- aws:PrincipalIsAWSService
- aws:Ec2InstanceState
- aws:Ec2InstanceState

- aws:SourceOrgID
- aws:SourceOrgPaths

Additionally, for a resource-based policy to be non-public, the values for Amazon Resource Name (ARN) and string keys must not contain wildcards or variables. If your resource-based policy uses the aws:PrincipalIsAWSService key, you must make sure that you've set the key value to true.

The following policy limits access to the user John in the specified account. The condition makes the Principal constrained and not be considered as public.

```
{  
    "Effect": "Allow",  
    "Principal": {  
        "AWS": "*"  
    },  
    "Action": "dynamodb:*",  
    "Resource": "*",  
    "Condition": {  
        "StringEquals": {  
            "aws:PrincipalArn": "arn:aws:iam::123456789012:user/John"  
        }  
    }  
}
```

The following example of a non-public resource-based policy constrains sourceVPC using the StringEquals operator.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "*"  
            },  
            "Action": "dynamodb:*",  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",  
            "Condition": {  
                "StringEquals": {  
                    "aws:SourceVpc": [  
                        "vpc-91237329"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        ]
    }
}
]
}
```

IAM actions supported by resource-based policies

IAM actions and cross-account support through resource-based policies is restricted to a certain set of DynamoDB APIs. You can't attach resource-based policies to resource types, such as backups and imports. Also, APIs that operate on these resource types are excluded from the supported IAM actions in resource-based policies. Because table administrators configure internal table settings within the same account, APIs, such as [UpdateTimeToLive](#) and [DisableKinesisStreamingDestination](#), don't support cross-account access through resource-based policies.

The DynamoDB data plane and control plane APIs that support cross-account access also support table name overloading, which lets you specify the table ARN instead of the table name. You can specify table ARN in the `TableName` parameter of these APIs. However, not all of these APIs support cross-account access.

The following table lists the API-level support for resource-based policies and cross-account access.

API action	Resource-based policy support	Cross-account support
Data Plane - Tables/indexes		
DeleteItem	Yes	Yes
GetItem	Yes	Yes
PutItem	Yes	Yes
Query	Yes	Yes
Scan	Yes	Yes
UpdateItem	Yes	Yes
TransactGetItems	Yes	Yes

API action	Resource-based policy support	Cross-account support
TransactWriteItems	Yes	Yes
BatchGetItem	Yes	Yes
BatchWriteItem	Yes	Yes
PartiQL		
BatchExecuteStatement	Yes	No
ExecuteStatement	Yes	No
ExecuteTransaction	Yes	No
Control Plane - Tables		
CreateTable	No	No
DeleteTable	Yes	Yes
DescribeTable	Yes	Yes
UpdateTable	Yes	Yes
Version 2019.11.21 (Current) global tables		
DescribeTableReplicaAutoScaling	Yes	No
UpdateTableReplicaAutoScaling	Yes	No
Version 2017.11.29 (Legacy) global table		
CreateGlobalTable	No	No
DescribeGlobalTable	No	No
DescribeGlobalTableSettings	No	No

API action	Resource-based policy support	Cross-account support
ListGlobalTables	No	No
UpdateGlobalTable	No	No
UpdateGlobalTableSettings	No	No
Tags		
ListTagsOfResource	Yes	Yes
TagResource	Yes	Yes
UntagResource	Yes	Yes
Backup/Restore		
CreateBackup	Yes	No
DescribeBackup	No	No
DeleteBackup	No	No
RestoreTableFromBackup	No	No
Continuous Backup/Restore (PITR)		
DescribeContinuousBackups	Yes	No
RestoreTableToPointInTime	Yes	No
UpdateContinuousBackups	Yes	No
Contributor Insights		
DescribeContributorInsights	Yes	No
ListContributorInsights	No	No
UpdateContributorInsights	Yes	No

API action	Resource-based policy support	Cross-account support
Export		
DescribeExport	No	No
ExportTableToPointInTime	Yes	No
ListExports	No	No
Import		
DescribeImport	No	No
ImportTable	No	No
ListImports	No	No
Kinesis		
DescribeKinesisStreamingDestination	Yes	No
DisableKinesisStreamingDestination	Yes	No
EnableKinesisStreamingDestination	Yes	No
UpdateKinesisStreamingDestination	Yes	No
Resource policies		
GetResourcePolicy	Yes	No
PutResourcePolicy	Yes	No
DeleteResourcePolicy	Yes	No
Time-to-Live		

API action	Resource-based policy support	Cross-account support
DescribeTimeToLive	Yes	No
UpdateTimeToLive	Yes	No
Others		
DescribeLimits	No	No
DescribeEndpoints	No	No
ListBackups	No	No
ListTables	No	No

The following table lists the API-level support of DynamoDB Streams APIs for resource-based policies and cross-account access.

API action	Resource-based policy support	Cross-account support
DescribeStream	Yes	Yes
GetRecords	Yes	Yes
GetShardIterator	Yes	Yes
ListStreams	No	No

Authorization with IAM identity-based policies and DynamoDB resource-based policies

Identity-based policies are attached to an identity, such as IAM users, groups of users, and roles. These are IAM policy documents that control what actions an identity can perform, on which resources, and under what conditions. Identity-based policies can be [managed](#) or [inline](#) policies.

Resource-based policies are IAM policy documents that you attach to a resource, such as a DynamoDB table. These policies grant the specified principal permission to perform specific actions on that resource and defines under what conditions this applies. For example, the resource-based policy for a DynamoDB table also includes the index associated with the table. Resource-based policies are inline policies. There are no managed resource-based policies.

For more information about these policies, see [Identity-based policies and resource-based policies](#) in the *IAM User Guide*.

If the IAM principal is from the same account as the resource owner, a resource-based policy is sufficient to specify access permissions to the resource. You can still choose to have an IAM identity-based policy along with a resource-based policy. For cross-account access, you must explicitly allow access in both the identity and resource policies as specified in [Cross-account access with resource-based policies](#). When you use both types of policies, a policy is evaluated as described in [Determining whether a request is allowed or denied within an account](#).

Resource-based policy examples

When you specify an ARN in the Resource field of a resource-based policy, the policy takes effect only if the specified ARN matches the ARN of the DynamoDB resource to which it is attached.

 **Note**

Remember to replace the *italicized* text with your resource-specific information.

Resource-based policy for a table

The following resource-based policy attached to a DynamoDB table named *MusicCollection*, gives the IAM users *John* and *Jane* permission to perform [GetItem](#) and [BatchGetItem](#) actions on the *MusicCollection* resource.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "1111",  
            "Effect": "Allow",  
            "Action": "DynamoDB:PutItem",  
            "Resource": "arn:aws:dynamodb:Region:Account:table/MusicCollection/Primary Key Value"  
        }  
    ]  
}
```

```
    "Principal": {
        "AWS": [
            "arn:aws:iam::111122223333:user/John",
            "arn:aws:iam::111122223333:user/Jane"
        ]
    },
    "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem"
    ],
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    ]
}
]
}
```

Resource-based policy for a stream

The following resource-based policy attached to a DynamoDB stream named 2024-02-12T18:57:26.492 gives the IAM users *John* and *Jane* permission to perform [GetRecords](#), [GetShardIterator](#), and [DescribeStream](#) API actions on the 2024-02-12T18:57:26.492 resource.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "1111",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:user/John",
                    "arn:aws:iam::111122223333:user/Jane"
                ]
            },
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator"
            ],
        }
    ]
}
```

```
"Resource": [
    "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/
    stream/2024-02-12T18:57:26.492"
]
}
]
```

Resource-based policy for access to perform all actions on specified resources

To allow a user to perform all actions on a table and all associated indexes with a table, you can use a wildcard (*) to represent the actions and the resources associated with the table. Using a wild card character for the resources, will allow the user access to the DynamoDB table and all its associated indexes, including the ones that haven't yet been created. For example, the following policy will give the user **John** permission to perform any actions on the **MusicCollection** table and all of its indexes, including any indexes that will be created in the future.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "1111",
            "Effect": "Allow",
            "Principal": "arn:aws:iam::111122223333:user/John",
            "Action": "dynamodb:*",
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/index/*"
            ]
        }
    ]
}
```

Resource-based policy for cross-account access

You can specify permissions for a cross-account IAM identity to access DynamoDB resources. For example, you might need a user from a trusted account to get access to read the contents of your table, with the condition that they access only specific items and specific attributes in those items.

The following policy allows access to user *John* from a trusted AWS account ID **111111111111** to access data from a table in account **123456789012** by using the [GetItem](#) API. The policy ensures that the user can access only items with a primary key *Jane* and that the user can only retrieve the attributes *Artist* and *SongTitle*, but no other attributes.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "CrossAccountTablePolicy",  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "arn:aws:iam::111111111111:user/John"  
            },  
            "Action": "dynamodb:GetItem",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:LeadingKeys": "Jane",  
                    "dynamodb:Attributes": [  
                        "Artist",  
                        "SongTitle"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

In addition to the preceding resource-based policy, the identity-based policy attached to the user *John* also needs to allow the GetItem API action for the cross-account access to work. The following is an example of an identity-based policy that you must attach to the user *John*.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "CrossAccountIdentityBasedPolicy",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:BatchWriteItem",  
                "dynamodb:ListTables",  
                "dynamodb:DescribeTable",  
                "dynamodb:UpdateTable",  
                "dynamodb:CreateTable",  
                "dynamodb:DeleteTable"  
            ]  
        }  
    ]  
}
```

```
        "dynamodb:GetItem"
    ],
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    ],
    "Condition": {
        "ForAllValues:StringEquals": {
            "dynamodb:LeadingKeys": "Jane",
            "dynamodb:Attributes": [
                "Artist",
                "SongTitle"
            ]
        }
    }
}
]
```

The user John can make a GetItem request by specifying the table ARN in the table-name parameter for accessing the table *MusicCollection* in the account **123456789012**.

```
aws dynamodb get-item \
--table-name arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
--key '{"Artist": {"S": "Jane"}, "SongTitle": "Monsoon"}' \
--return-consumed-capacity TOTAL
```

Resource-based policy with IP address conditions

You can apply a condition to restrict source IP addresses, virtual private clouds (VPCs), and VPC endpoint (VPCE). You can specify permissions based on the source addresses of the originating request. For example, you might want to allow a user to access DynamoDB resources only if they are being accessed from a specific IP source, such as a corporate VPN endpoint. Specify these IP addresses in the Condition statement.

The following example allows the user *John* access to any DynamoDB resource when the source IPs are 54.240.143.0/24 and 2001:DB8:1234:5678::/64.

```
{
    "Id": "PolicyId2",
    "Version": "2012-10-17",
    "Statement": [
```

```
{  
    "Sid": "AllowIPmix",  
    "Effect": "Allow",  
    "Principal": "arn:aws:iam::111111111111:user/John",  
    "Action": "dynamodb:*",  
    "Resource": "*",  
    "Condition": {  
        "IpAddress": {  
            "aws:SourceIp": [  
                "54.240.143.0/24",  
                "2001:DB8:1234:5678::/64"  
            ]  
        }  
    }  
}  
]
```

You can also deny all access to DynamoDB resources except when the source is a specific VPC endpoint, for example *vpce-1a2b3c4d*.

```
{  
    "Id": "PolicyId",  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AccessToSpecificVPCEOnly",  
            "Principal": "*",  
            "Action": "dynamodb:*",  
            "Effect": "Deny",  
            "Resource": "*",  
            "Condition": {  
                "StringNotEquals": {  
                    "aws:sourceVpce": "vpce-1a2b3c4d"  
                }  
            }  
        }  
    ]  
}
```

Resource-based policy using an IAM role

You can also specify an IAM service role in the resource-based policy. IAM entities that assume this role are bounded by the permissible actions specified for the role and to the specific set of resources within the resource-based policy.

The following example allows an IAM entity to perform all DynamoDB actions on the *MusicCollection* and *MusicCollection* DynamoDB resources.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "1111",  
            "Effect": "Allow",  
            "Principal": { "AWS": "arn:aws:iam::111122223333:role/John" },  
            "Action": "dynamodb:*",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection",  
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/*"  
            ]  
        }  
    ]  
}
```

Resource-based policy considerations

When you define resource-based policies for your DynamoDB resources, the following considerations apply:

General considerations

- The maximum size supported for a resource-based policy document is 20 KB. DynamoDB counts whitespaces when calculating the size of a policy against this limit.
- Subsequent updates to a policy for a given resource are blocked for 15 seconds after a successful update of the policy for the same resource.
- Currently, you can only attach a resource-based policy to existing streams. You can't attach a policy to a stream while creating it.

Global table considerations

- Resource-based policies aren't supported for [Global table version 2017.11.29 \(Legacy\)](#) replicas.
- Within a resource-based policy, if the action for a DynamoDB service-linked role (SLR) to replicate data for a global table is denied, adding or deleting a replica will fail with an error.
- The [AWS::DynamoDB::GlobalTable](#) resource doesn't support creating a replica and adding a resource-based policy to that replica in the same stack update in Regions other than the Region where you deploy the stack update.

Cross-account considerations

- Cross-account access using resource-based policies doesn't support encrypted tables with AWS managed keys because you can't grant cross-account access to the AWS managed KMS policy.

AWS CloudFormation considerations

- Resource-based policies don't support [drift detection](#). If you update a resource-based policy outside of the AWS CloudFormation stack template, you'll need to update the CloudFormation stack with the changes.
- Resource-based policies don't support out of band changes. If you add, update, or delete a policy outside of the CloudFormation template, the change won't be overwritten if there're no changes to the policy within the template.

For example, say that your template contains a resource-based policy which you later update outside of the template. If you don't make any changes to the policy in the template, the updated policy in DynamoDB won't be synced with the policy in the template.

Conversely, say that your template doesn't contain a resource-based policy, but you add a policy outside of the template. This policy won't be removed from DynamoDB as long as you don't add it to the template. When you add a policy to the template and update the stack, the existing policy in DynamoDB will be updated to match the one defined in the template.

Resource-based policy best practices

This topic describes the best practices for defining access permissions for your DynamoDB resources and the actions allowed on these resources.

Simplify access control to DynamoDB resources

If the AWS Identity and Access Management principals that need access to a DynamoDB resource are part of the same AWS account as the resource owner, an IAM identity-based policy is not required for each principal. A resource-based policy that is attached to the given resources will suffice. This type of configuration simplifies access control.

Protect your DynamoDB resources with resource-based policies

For all DynamoDB tables and streams, create resource-based policies to enforce access control for these resources. Resource-based policies enable you to centralize permissions at the resource level, simplify access control to DynamoDB tables, indexes, and streams, and reduce administration overhead. If no resource-based policy is specified for a table or a stream, access to the table or stream will be implicitly denied, unless identity-based policies associated with the IAM principals allow access.

Apply least-privilege permissions

When you set permissions with resource-based policies for DynamoDB resources, grant only the permissions required to perform an action. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as least-privilege permissions. You might start with broad permissions while you explore the permissions that are required for your workload or use case. As your use case matures, you can work to reduce the permissions that you grant to work toward least privilege.

Analyze cross-account access activity for generating least-privilege policies

IAM Access Analyzer reports cross-account access to external entities specified in resource-based policies, and provides visibility to help you refine permissions and conform to least privilege. For more information about policy generation, see [IAM Access Analyzer policy generation](#).

Use IAM Access Analyzer to generate least-privilege policies

To grant only the permissions required to perform a task, you can generate policies based on your access activity that is logged in AWS CloudTrail. IAM Access Analyzer analyzes the services and actions that your policies use.

Data protection in DynamoDB

Amazon DynamoDB provides a highly durable storage infrastructure designed for mission-critical and primary data storage. Data is redundantly stored on multiple devices across multiple facilities in an Amazon DynamoDB Region.

DynamoDB protects user data stored at rest and also data in transit between on-premises clients and DynamoDB, and between DynamoDB and other AWS resources within the same AWS Region.

Topics

- [DynamoDB encryption at rest](#)
- [Data protection in DynamoDB Accelerator](#)
- [Internetwork traffic privacy](#)

DynamoDB encryption at rest

All user data stored in Amazon DynamoDB is fully encrypted at rest. DynamoDB encryption at rest provides enhanced security by encrypting all your data at rest using encryption keys stored in [AWS Key Management Service \(AWS KMS\)](#). This functionality helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements.

DynamoDB encryption at rest provides an additional layer of data protection by always securing your data in an encrypted table—including its primary key, local and global secondary indexes, streams, global tables, backups, and DynamoDB Accelerator (DAX) clusters whenever the data is stored in durable media. Organizational policies, industry or government regulations, and compliance requirements often require the use of encryption at rest to increase the data security of your applications.

Encryption at rest integrates with AWS KMS for managing the encryption keys that are used to encrypt your tables. For more information on key types and states, see [AWS Key Management Service concepts](#) in the [AWS Key Management Service Developer Guide](#).

When creating a new table, you can choose one of the following AWS KMS key types to encrypt your table. You can switch between these key types at any time.

- **AWS owned key** – Default encryption type. The key is owned by DynamoDB (no additional charge).

- **AWS managed key** – The key is stored in your account and is managed by AWS KMS (AWS KMS charges apply).
- **Customer managed key** – The key is stored in your account and is created, owned, and managed by you. You have full control over the KMS key (AWS KMS charges apply).

For more information on key types, see [Customer keys and AWS keys](#).

 **Note**

- When creating a new DAX cluster with encryption at rest enabled, an AWS managed key will be used to encrypt data at rest in the cluster.
- If your table has a sort key, some of the sort keys that mark range boundaries are stored in plaintext in the table metadata.

When you access an encrypted table, DynamoDB decrypts the table data transparently. You don't have to change any code or applications to use or manage encrypted tables. DynamoDB continues to deliver the same single-digit millisecond latency that you have come to expect, and all DynamoDB queries work seamlessly on your encrypted data.

You can specify an encryption key when you create a new table or switch the encryption keys on an existing table by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the Amazon DynamoDB API. To learn how, see [Managing encrypted tables in DynamoDB](#).

Encryption at rest using the AWS owned key is offered at no additional charge. However, AWS KMS charges apply for an AWS managed key and for a customer managed key. For more information about pricing, see [AWS KMS pricing](#).

DynamoDB encryption at rest is available in all AWS Regions, including the AWS China (Beijing) and AWS China (Ningxia) Regions and the AWS GovCloud (US) Regions. For more information, see [Encryption at rest: How it works](#) and [DynamoDB encryption at rest usage notes](#).

Encryption at rest: How it works

Amazon DynamoDB encryption at rest encrypts your data using 256-bit Advanced Encryption Standard (AES-256), which helps secure your data from unauthorized access to the underlying storage.

Encryption at rest integrates with AWS Key Management Service (AWS KMS) for managing the encryption keys that are used to encrypt your tables.

Note

In May 2022, AWS KMS changed the rotation schedule for AWS managed keys from every three years (approximately 1,095 days) to every year (approximately 365 days).

New AWS managed keys are automatically rotated one year after they are created, and approximately every year thereafter.

Existing AWS managed keys are automatically rotated one year after their most recent rotation, and every year thereafter.

AWS owned keys

AWS owned keys are not stored in your AWS account. They are part of a collection of KMS keys that AWS owns and manages for use in multiple AWS accounts. AWS services can use AWS owned keys to protect your data. AWS owned keys used by DynamoDB are rotated every year (approximately 365 days).

You cannot view, manage, or use AWS owned keys, or audit their use. However, you do not need to do any work or change any programs to protect the keys that encrypt your data.

You are not charged a monthly fee or a usage fee for use of AWS owned keys, and they do not count against AWS KMS quotas for your account.

AWS managed keys

AWS managed keys are KMS keys in your account that are created, managed, and used on your behalf by an AWS service that is integrated with AWS KMS. You can view the AWS managed keys in your account, view their key policies, and audit their use in AWS CloudTrail logs. However, you cannot manage these KMS keys or change their permissions.

Encryption at rest automatically integrates with AWS KMS for managing the AWS managed keys for DynamoDB (`aws/dynamodb`) that are used to encrypt your tables. If an AWS managed key doesn't exist when you create your encrypted DynamoDB table, AWS KMS automatically creates a new key for you. This key is used with encrypted tables that are created in the future. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

For more information about managing permissions of the AWS managed key, see [Authorizing use of the AWS managed key](#) in the *AWS Key Management Service Developer Guide*.

Customer managed keys

Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys, including establishing and maintaining their key policies, IAM policies, and grants; enabling and disabling them; rotating their cryptographic material; adding tags; creating aliases that refer to them; and scheduling them for deletion. For more information about managing permissions of a customer managed key, see [Customer managed key policy](#).

When you specify a customer managed key as the table-level encryption key, the DynamoDB table, local and global secondary indexes, and streams are encrypted with the same customer managed key. On-demand backups are encrypted with the table-level encryption key that is specified at the time the backup is created. Updating the table-level encryption key does not change the encryption key that is associated with existing on-demand backups.

Setting the state of the customer managed key to disabled or scheduling it for deletion prevents all users and the DynamoDB service from being able to encrypt or decrypt data and to perform read and write operations on the table. DynamoDB must have access to your encryption key to ensure that you can continue to access your table and to prevent data loss.

If you disable your customer managed key or schedule it for deletion, your table status becomes **Inaccessible**. To ensure that you can continue working with the table, you must provide DynamoDB access to the specified encryption key within seven days. As soon as the service detects that your encryption key is inaccessible, DynamoDB sends you an email notification to alert you.

Note

- If your customer managed key remains inaccessible to the DynamoDB service for longer than seven days, the table is archived and can no longer be accessed. DynamoDB creates an on-demand backup of your table, and you are billed for it. You can use this on-demand backup to restore your data to a new table. To initiate the restore, the last customer managed key on the table must be enabled, and DynamoDB must have access to it.
- If your customer managed key that was used to encrypt a global table replica is inaccessible DynamoDB will remove this replica from the replication group. The replica

will not be deleted and replication will stop from and to this region, 20 hours after detecting the customer managed key as inaccessible.

For more information, see [enabling keys](#) and [deleting keys](#).

Notes on using AWS managed keys

Amazon DynamoDB can't read your table data unless it has access to the KMS key stored in your AWS KMS account. DynamoDB uses envelope encryption and key hierarchy to encrypt data. Your AWS KMS encryption key is used to encrypt the root key of this key hierarchy. For more information, see [Envelope encryption](#) in the *AWS Key Management Service Developer Guide*.

You can use AWS CloudTrail and Amazon CloudWatch Logs to track the requests that DynamoDB sends to AWS KMS on your behalf. For more information, see [Monitoring DynamoDB interaction with AWS KMS](#) in the *AWS Key Management Service Developer Guide*.

DynamoDB doesn't call AWS KMS for every DynamoDB operation. The key is refreshed once every 5 minutes per caller with active traffic.

Ensure that you have configured the SDK to reuse connections. Otherwise, you will experience latencies from DynamoDB having to reestablish new AWS KMS cache entries for each DynamoDB operation. In addition, you might potentially have to face higher AWS KMS and CloudTrail costs. For example, to do this using the Node.js SDK, you can create a new HTTPS agent with `keepAlive` turned on. For more information, see [Configuring keepAlive in Node.js](#) in the *AWS SDK for JavaScript Developer Guide*.

DynamoDB encryption at rest usage notes

Consider the following when you are using encryption at rest in Amazon DynamoDB.

All table data is encrypted

Server-side encryption at rest is enabled on all DynamoDB table data and cannot be disabled. You cannot encrypt only a subset of items in a table.

Encryption at rest only encrypts data while it is static (at rest) on a persistent storage media. If data security is a concern for data in transit or data in use, you might need to take additional measures:

- Data in transit: All your data in DynamoDB is encrypted in transit. By default, communications to and from DynamoDB use the HTTPS protocol, which protects network traffic by using Secure Sockets Layer (SSL)/Transport Layer Security (TLS) encryption.
- Data in use: Protect your data before sending it to DynamoDB using client-side encryption. For more information, see [Client-side and server-side encryption](#) in the *Amazon DynamoDB Encryption Client Developer Guide*.

You can use streams with encrypted tables. DynamoDB streams are always encrypted with a table-level encryption key. For more information, see [Change data capture for DynamoDB Streams](#).

DynamoDB backups are encrypted, and the table that is restored from a backup also has encryption enabled. You can use the AWS owned key, AWS managed key, or customer managed key to encrypt your backup data. For more information, see [Using On-Demand backup and restore for DynamoDB](#).

Local secondary indexes and global secondary indexes are encrypted using the same key as the base table.

Encryption types

Note

Customer managed keys are not supported in Global Table Version 2017. If you want to use a customer managed key in a DynamoDB Global Table, you need to upgrade the table to Global Table Version 2019 and then enable it.

On the AWS Management Console, the encryption type is KMS when you use the AWS managed key or customer managed key to encrypt your data. The encryption type is DEFAULT when you use the AWS owned key. In the Amazon DynamoDB API, the encryption type is KMS when you use the AWS managed key or customer managed key. In the absence of encryption type, your data is encrypted using the AWS owned key. You can switch between the AWS owned key, AWS managed key, and customer managed key at any given time. You can use the console, the AWS Command Line Interface (AWS CLI), or the Amazon DynamoDB API to switch the encryption keys.

Note the following limitations when using customer managed keys:

- You cannot use a customer managed key with DynamoDB Accelerator (DAX) clusters. For more information, see [DAX encryption at rest](#).

- You can use a customer managed key to encrypt tables that use transactions. However, to ensure durability for propagation of transactions, a copy of the transaction request is temporarily stored by the service and encrypted using an AWS owned key. Committed data in your tables and secondary indexes is always encrypted at rest using your customer managed key.
- You can use a customer managed key to encrypt tables that use Contributor Insights. However, data that is transmitted to Amazon CloudWatch is encrypted with an AWS owned key.
- When you transition to a new customer managed key, be sure to keep the original key enabled until the process is complete. AWS will still need the original key to decrypt the data before encrypting it with the new key. The process will be complete when the table's SSEDescription Status is ENABLED and the KMSMasterKeyArn of the new customer managed key is displayed. At this point the original key can be disabled or scheduled for deletion.
- Once the new customer managed key is displayed, the table and any new on-demand backups are encrypted with the new key.
- Any existing on-demand backups remain encrypted with the customer managed key that was used when those backups were created. You will need that same key to restore those backups. You can identify the key for the period when each backup was created by using the DescribeBackup API to view that backup's SSEDescription.
- If you disable your customer managed key or schedule it for deletion, any data in DynamoDB Streams is still subject to a 24-hour lifetime. Any unretrieved activity data is eligible for trimming when it is older than 24 hours.
- If you disable your customer managed key or schedule it for deletion, Time to Live (TTL) deletes continue for 30 minutes. These TTL deletes continue to be emitted to DynamoDB Streams and are subject to the standard trimming/retention interval.

For more information, see [enabling keys](#) and [deleting keys](#).

Using KMS keys and data keys

The DynamoDB encryption at rest feature uses an AWS KMS key and a hierarchy of data keys to protect your table data. DynamoDB uses the same key hierarchy to protect DynamoDB streams, global tables, and backups when they are written to durable media.

We recommend that you plan your encryption strategy before implementing your table in DynamoDB. If you store sensitive or confidential data in DynamoDB, consider including client-side encryption in your plan. This way you can encrypt data as close as possible to its origin, and ensure

its protection throughout its lifecycle. For more information see the [DynamoDB encryption client](#) documentation.

AWS KMS key

Encryption at rest protects your DynamoDB tables under an AWS KMS key. By default, DynamoDB uses an [AWS owned key](#), a multi-tenant encryption key that is created and managed in a DynamoDB service account. But you can encrypt your DynamoDB tables under a [customer managed key](#) for DynamoDB (aws/dynamodb) in your AWS account. You can select a different KMS key for each table. The KMS key you select for a table is also used to encrypt its local and global secondary indexes, streams, and backups.

You select the KMS key for a table when you create or update the table. You can change the KMS key for a table at any time, either in the DynamoDB console or by using the [UpdateTable](#) operation. The process of switching keys is seamless and does not require downtime or degrade service.

Important

DynamoDB supports only [symmetric KMS keys](#). You cannot use an [asymmetric KMS key](#) to encrypt your DynamoDB tables.

Use a customer managed key to get the following features:

- You create and manage the KMS key, including setting the [key policies](#), [IAM policies](#) and [grants](#) to control access to the KMS key. You can [enable and disable](#) the KMS key, enable and disable [automatic key rotation](#), and [delete the KMS key](#) when it is no longer in use.
- You can use a customer managed key with [imported key material](#) or a customer managed key in a [custom key store](#) that you own and manage.
- You can audit the encryption and decryption of your DynamoDB table by examining the DynamoDB API calls to AWS KMS in [AWS CloudTrail logs](#).

Use the AWS managed key if you need any of the following features:

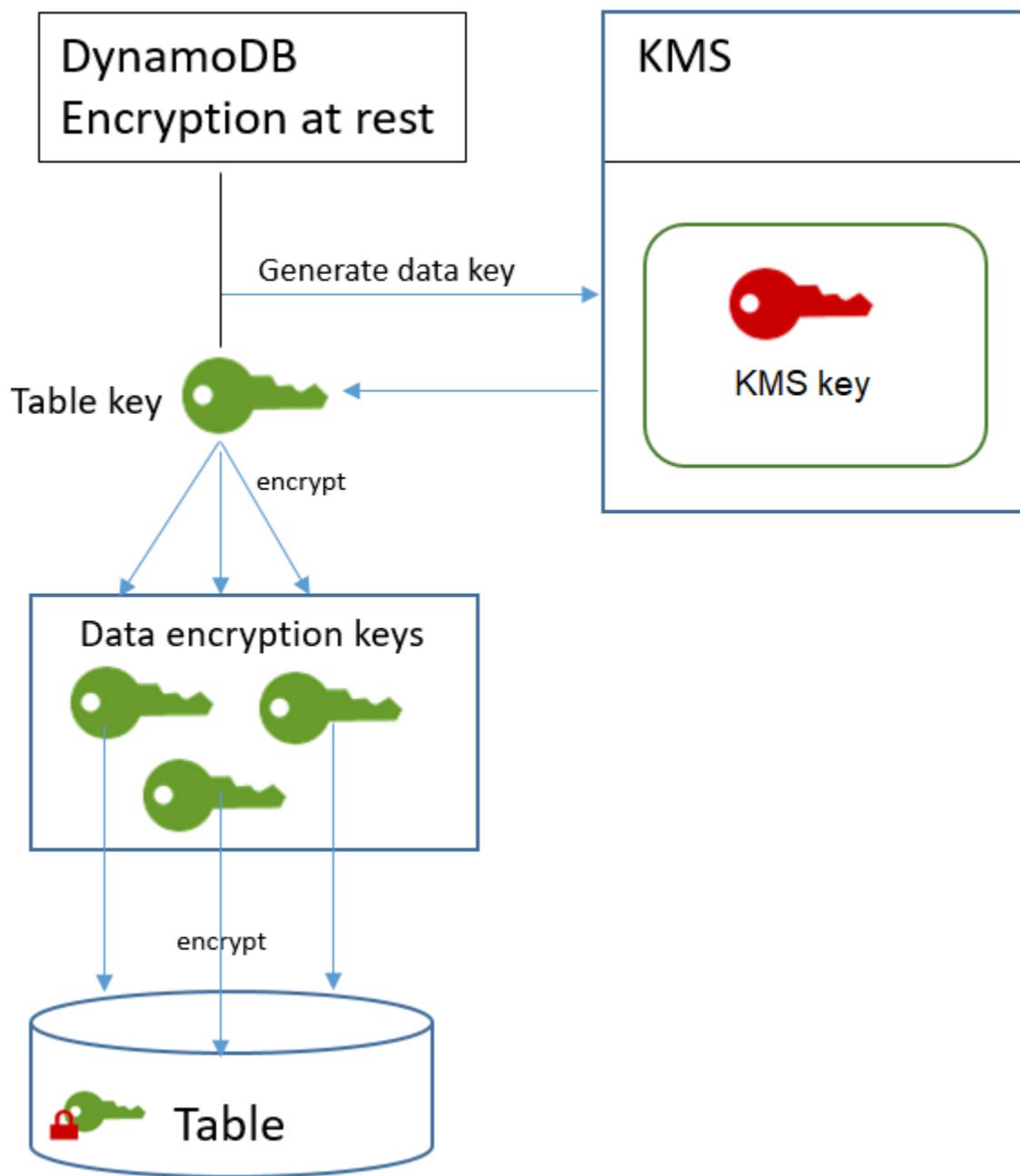
- You can [view the KMS key](#) and [view its key policy](#). (You cannot change the key policy.)
- You can audit the encryption and decryption of your DynamoDB table by examining the DynamoDB API calls to AWS KMS in [AWS CloudTrail logs](#).

However, the AWS owned key is free of charge and its use does not count against [AWS KMS resource or request quotas](#). Customer managed keys and AWS managed keys [incur a charge](#) for each API call and AWS KMS quotas apply to these KMS keys.

Table keys

DynamoDB uses the KMS key for the table to [generate](#) and encrypt a unique [data key](#) for the table, known as the *table key*. The table key persists for the lifetime of the encrypted table.

The table key is used as a key encryption key. DynamoDB uses this table key to protect data encryption keys that are used to encrypt the table data. DynamoDB generates a unique data encryption key for each underlying structure in a table, but multiple table items might be protected by the same data encryption key.



When you first access an encrypted table, DynamoDB sends a request to AWS KMS to use the KMS key to decrypt the table key. Then, it uses the plaintext table key to decrypt the data encryption keys, and uses the plaintext data encryption keys to decrypt table data.

DynamoDB stores and uses the table key and data encryption keys outside of AWS KMS. It protects all keys with [Advanced Encryption Standard \(AES\)](#) encryption and 256-bit encryption

keys. Then, it stores the encrypted keys with the encrypted data so they are available to decrypt the table data on demand.

If you change the KMS key for your table, DynamoDB generates a new table key. Then, it uses the new table key to re-encrypt the data encryption keys.

Table key caching

To avoid calling AWS KMS for every DynamoDB operation, DynamoDB caches the plaintext table keys for each caller in memory. If DynamoDB gets a request for the cached table key after five minutes of inactivity, it sends a new request to AWS KMS to decrypt the table key. This call will capture any changes made to the access policies of the KMS key in AWS KMS or AWS Identity and Access Management (IAM) since the last request to decrypt the table key.

Authorizing use of your KMS key

If you use a [customer managed key](#) or the [AWS managed key](#) in your account to protect your DynamoDB table, the policies on that KMS key must give DynamoDB permission to use it on your behalf. The authorization context on the AWS managed key for DynamoDB includes its key policy and grants that delegate the permissions to use it.

You have full control over the policies and grants on a customer managed key. Because the AWS managed key is in your account, you can view its policies and grants. But, because it is managed by AWS, you cannot change the policies.

DynamoDB does not need additional authorization to use the default [AWS owned key](#) to protect the DynamoDB tables in your AWS account.

Topics

- [Key policy for an AWS managed key](#)
- [Key policy for a customer managed key](#)
- [Using grants to authorize DynamoDB](#)

Key policy for an AWS managed key

When DynamoDB uses the [AWS managed key](#) for DynamoDB (aws/dynamodb) in cryptographic operations, it does so on behalf of the user who is accessing the [DynamoDB resource](#). The key policy on the AWS managed key gives all users in the account permission to use the AWS managed key for specified operations. But permission is granted only when DynamoDB makes the request on

the user's behalf. The [ViaService condition](#) in the key policy does not allow any user to use the AWS managed key unless the request originates with the DynamoDB service.

This key policy, like the policies of all AWS managed keys, is established by AWS. You cannot change it, but you can view it at any time. For details, see [Viewing a key policy](#).

The policy statements in the key policy have the following effect:

- Allow users in the account to use the AWS managed key for DynamoDB in cryptographic operations when the request comes from DynamoDB on their behalf. The policy also allows users to [create grants](#) for the KMS key.
- Allows authorized IAM identities in the account to view the properties of the AWS managed key for DynamoDB and to [revoke the grant](#) that allows DynamoDB to use the KMS key. DynamoDB uses [grants](#) for ongoing maintenance operations.
- Allows DynamoDB to perform read-only operations to find the AWS managed key for DynamoDB in your account.

```
{  
    "Version" : "2012-10-17",  
    "Id" : "auto-dynamodb-1",  
    "Statement" : [ {  
        "Sid" : "Allow access through Amazon DynamoDB for all principals in the account  
        that are authorized to use Amazon DynamoDB",  
        "Effect" : "Allow",  
        "Principal" : {  
            "AWS" : "*"  
        },  
        "Action" : [ "kms:Encrypt", "kms:Decrypt", "kms:ReEncrypt*",  
        "kms:GenerateDataKey*", "kms>CreateGrant", "kms:DescribeKey" ],  
        "Resource" : "*",  
        "Condition" : {  
            "StringEquals" : {  
                "kms:CallerAccount" : "111122223333",  
                "kms:ViaService" : "dynamodb.us-west-2.amazonaws.com"  
            }  
        }  
    }, {  
        "Sid" : "Allow direct access to key metadata to the account",  
        "Effect" : "Allow",  
        "Principal" : {  
            "AWS" : "arn:aws:iam::111122223333:root"  
        }  
    }  
}
```

```
},
  "Action" : [ "kms:Describe*", "kms:Get*", "kms>List*" ], "kms:RevokeGrant" ],
  "Resource" : "*"
}, {
  "Sid" : "Allow DynamoDB Service with service principal name dynamodb.amazonaws.com to describe the key directly",
  "Effect" : "Allow",
  "Principal" : {
    "Service" : "dynamodb.amazonaws.com"
  },
  "Action" : [ "kms:Describe*", "kms:Get*", "kms>List*" ],
  "Resource" : "*"
} ]
}
```

Key policy for a customer managed key

When you select a [customer managed key](#) to protect a DynamoDB table, DynamoDB gets permission to use the KMS key on behalf of the principal who makes the selection. That principal, a user or role, must have the permissions on the KMS key that DynamoDB requires. You can provide these permissions in a [key policy](#), an [IAM policy](#), or a [grant](#).

At a minimum, DynamoDB requires the following permissions on a customer managed key:

- [kms:Encrypt](#)
- [kms:Decrypt](#)
- [kms:ReEncrypt*](#) (for kms:ReEncryptFrom and kms:ReEncryptTo)
- [kms:GenerateDataKey*](#) (for [kms:GenerateDataKey](#) and [kms:GenerateDataKeyWithoutPlaintext](#))
- [kms:DescribeKey](#)
- [kms>CreateGrant](#)

For example, the following example key policy provides only the required permissions. The policy has the following effects:

- Allows DynamoDB to use the KMS key in cryptographic operations and create grants, but only when it is acting on behalf of principals in the account who have permission to use DynamoDB. If the principals specified in the policy statement don't have permission to use DynamoDB, the call fails, even when it comes from the DynamoDB service.

- The [kms:ViaService](#) condition key allows the permissions only when the request comes from DynamoDB on behalf of the principals listed in the policy statement. These principals can't call these operations directly. Note that the kms:ViaService value, dynamodb.*.amazonaws.com, has an asterisk (*) in the Region position. DynamoDB requires the permission to be independent of any particular AWS Region so it can make cross-Region calls to support [DynamoDB global tables](#).
- Gives the KMS key administrators (users who can assume the db-team role) read-only access to the KMS key and permission to revoke grants, including the [grants that DynamoDB requires](#) to protect the table.

Before using an example key policy, replace the example principals with actual principals from your AWS account.

```
{  
  "Id": "key-policy-dynamodb",  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "Allow access through Amazon DynamoDB for all principals in the account  
      that are authorized to use Amazon DynamoDB",  
      "Effect": "Allow",  
      "Principal": {"AWS": "arn:aws:iam::111122223333:user/db-lead"},  
      "Action": [  
        "kms:Encrypt",  
        "kms:Decrypt",  
        "kms:ReEncrypt*",  
        "kms:GenerateDataKey*",  
        "kms:DescribeKey",  
        "kms>CreateGrant"  
      ],  
      "Resource": "*",  
      "Condition": {  
        "StringLike": {  
          "kms:ViaService" : "dynamodb.*.amazonaws.com"  
        }  
      }  
    },  
    {  
      "Sid": "Allow administrators to view the KMS key and revoke grants",  
      "Effect": "Allow",  
      "Principal": {
```

```
        "AWS": "arn:aws:iam::111122223333:role/db-team"
    },
    "Action": [
        "kms:Describe*",
        "kms:Get*",
        "kms>List*",
        "kms:RevokeGrant"
    ],
    "Resource": "*"
}
]
```

Using grants to authorize DynamoDB

In addition to key policies, DynamoDB uses grants to set permissions on a customer managed key or the AWS managed key for DynamoDB (aws/dynamodb). To view the grants on a KMS key in your account, use the [ListGrants](#) operation. DynamoDB does not need grants, or any additional permissions, to use the [AWS owned key](#) to protect your table.

DynamoDB uses the grant permissions when it performs background system maintenance and continuous data protection tasks. It also uses grants to generate [table keys](#).

Each grant is specific to a table. If the account includes multiple tables encrypted under the same KMS key, there is a grant of each type for each table. The grant is constrained by the [DynamoDB encryption context](#), which includes the table name and the AWS account ID, and it includes permission to the [retire the grant](#) if it is no longer needed.

To create the grants, DynamoDB must have permission to call `CreateGrant` on behalf of the user who created the encrypted table. For AWS managed keys, DynamoDB gets `kms:CreateGrant` permission from the [key policy](#), which allows account users to call [CreateGrant](#) on the KMS key only when DynamoDB makes the request on an authorized user's behalf.

The key policy can also allow the account to [revoke the grant](#) on the KMS key. However, if you revoke the grant on an active encrypted table, DynamoDB will not be able to protect and maintain the table.

DynamoDB encryption context

An [encryption context](#) is a set of key-value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds

the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

DynamoDB uses the same encryption context in all AWS KMS cryptographic operations. If you use a [customer managed key](#) or an [AWS managed key](#) to protect your DynamoDB table, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as [AWS CloudTrail](#) and [Amazon CloudWatch Logs](#).

The encryption context can also be used as a condition for authorization in policies and grants. DynamoDB uses the encryption context to constrain the [grants](#) that allow access to the customer managed key or AWS managed key in your account and region.

In its requests to AWS KMS, DynamoDB uses an encryption context with two key–value pairs.

```
"encryptionContextSubset": {  
    "aws:dynamodb:tableName": "Books"  
    "aws:dynamodb:subscriberId": "111122223333"  
}
```

- **Table** – The first key–value pair identifies the table that DynamoDB is encrypting. The key is `aws:dynamodb:tableName`. The value is the name of the table.

```
"aws:dynamodb:tableName": "<table-name>"
```

For example:

```
"aws:dynamodb:tableName": "Books"
```

- **Account** – The second key–value pair identifies the AWS account. The key is `aws:dynamodb:subscriberId`. The value is the account ID.

```
"aws:dynamodb:subscriberId": "<account-id>"
```

For example:

```
"aws:dynamodb:subscriberId": "111122223333"
```

Monitoring DynamoDB interaction with AWS KMS

If you use a [customer managed key](#) or an [AWS managed key](#) to protect your DynamoDB tables, you can use AWS CloudTrail logs to track the requests that DynamoDB sends to AWS KMS on your behalf.

The `GenerateDataKey`, `Decrypt`, and `CreateGrant` requests are discussed in this section. In addition, DynamoDB uses a [DescribeKey](#) operation to determine whether the KMS key you selected exists in the account and region. It also uses a [RetireGrant](#) operation to remove a grant when you delete a table.

GenerateDataKey

When you enable encryption at rest on a table, DynamoDB creates a unique table key. It sends a [GenerateDataKey](#) request to AWS KMS that specifies the KMS key for the table.

The event that records the `GenerateDataKey` operation is similar to the following example event. The user is the DynamoDB service account. The parameters include the Amazon Resource Name (ARN) of the KMS key, a key specifier that requires a 256-bit key, and the [encryption context](#) that identifies the table and the AWS account.

```
{  
    "eventVersion": "1.05",  
    "userIdentity": {  
        "type": "AWSService",  
        "invokedBy": "dynamodb.amazonaws.com"  
    },  
    "eventTime": "2018-02-14T00:15:17Z",  
    "eventSource": "kms.amazonaws.com",  
    "eventName": "GenerateDataKey",  
    "awsRegion": "us-west-2",  
    "sourceIPAddress": "dynamodb.amazonaws.com",  
    "userAgent": "dynamodb.amazonaws.com",  
    "requestParameters": {  
        "encryptionContext": {  
            "aws:dynamodb:tableName": "Services",  
            "aws:dynamodb:subscriberId": "111122223333"  
        },  
        "keySpec": "AES_256",  
        "keyId": "1234abcd-12ab-34cd-56ef-1234567890ab"  
    },  
    "responseElements": null,  
}
```

```
"requestID": "229386c1-111c-11e8-9e21-c11ed5a52190",
"eventID": "e3c436e9-ebca-494e-9457-8123a1f5e979",
"readOnly": true,
"resources": [
    {
        "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        "accountId": "111122223333",
        "type": "AWS::KMS::Key"
    }
],
"eventType": "AwsApiCall",
"recipientAccountId": "111122223333",
"sharedEventID": "bf915fa6-6ceb-4659-8912-e36b69846aad"
}
```

Decrypt

When you access an encrypted DynamoDB table, DynamoDB needs to decrypt the table key so that it can decrypt the keys below it in the hierarchy. It then decrypts the data in the table. To decrypt the table key, DynamoDB sends a [Decrypt](#) request to AWS KMS that specifies the KMS key for the table.

The event that records the Decrypt operation is similar to the following example event. The user is the principal in your AWS account who is accessing the table. The parameters include the encrypted table key (as a ciphertext blob) and the [encryption context](#) that identifies the table and the AWS account. AWS KMS derives the ID of the KMS key from the ciphertext.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-02-14T16:42:15Z"
            },
            "sessionIssuer": {
                "type": "Role",

```

```
        "principalId": "AROAIQDT3HGFQZX4RY6RU",
        "arn": "arn:aws:iam::111122223333:role/Admin",
        "accountId": "111122223333",
        "userName": "Admin"
    },
},
"invokeBy": "dynamodb.amazonaws.com"
},
"eventTime": "2018-02-14T16:42:39Z",
"eventSource": "kms.amazonaws.com",
"eventName": "Decrypt",
"awsRegion": "us-west-2",
"sourceIPAddress": "dynamodb.amazonaws.com",
"userAgent": "dynamodb.amazonaws.com",
"requestParameters":
{
    "encryptionContext":
    {
        "aws:dynamodb:tableName": "Books",
        "aws:dynamodb:subscriberId": "111122223333"
    }
},
"responseElements": null,
"requestID": "11cab293-11a6-11e8-8386-13160d3e5db5",
"eventID": "b7d16574-e887-4b5b-a064-bf92f8ec9ad3",
"readOnly": true,
"resources": [
    {
        "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        "accountId": "111122223333",
        "type": "AWS::KMS::Key"
    }
],
"eventType": "AwsApiCall",
"recipientAccountId": "111122223333"
}
```

CreateGrant

When you use a [customer managed key](#) or an [AWS managed key](#) to protect your DynamoDB table, DynamoDB uses [grants](#) to allow the service to perform continuous data protection and maintenance and durability tasks. These grants are not required on [AWS owned key](#).

The grants that DynamoDB creates are specific to a table. The principal in the [CreateGrant](#) request is the user who created the table.

The event that records the CreateGrant operation is similar to the following example event. The parameters include the Amazon Resource Name (ARN) of the KMS key for the table, the grantee principal and retiring principal (the DynamoDB service), and the operations that the grant covers. It also includes a constraint that requires all encryption operation use the specified [encryption context](#).

```
        "encryptionContextSubset": {
            "aws:dynamodb:tableName": "Books",
            "aws:dynamodb:subscriberId": "111122223333"
        },
        "granteePrincipal": "dynamodb.us-west-2.amazonaws.com",
        "operations": [
            "DescribeKey",
            "GenerateDataKey",
            "Decrypt",
            "Encrypt",
            "ReEncryptFrom",
            "ReEncryptTo",
            "RetireGrant"
        ],
        "responseElements": {
            "grantId": "5c5cd4a3d68e65e77795f5ccc2516dff057308172b0cd107c85b5215c6e48bde"
        },
        "requestID": "2192b82a-111c-11e8-a528-f398979205d8",
        "eventID": "a03d65c3-9fee-4111-9816-8bf96b73df01",
        "readOnly": false,
        "resources": [
            {
                "ARN": "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
                "accountId": "111122223333",
                "type": "AWS::KMS::Key"
            }
        ],
        "eventType": "AwsApiCall",
        "recipientAccountId": "111122223333"
    }
}
```

Managing encrypted tables in DynamoDB

You can use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to specify the encryption key on new tables and update the encryption keys on existing tables in Amazon DynamoDB.

Topics

- [Specifying the encryption key for a new table](#)
- [Updating an encryption key](#)

Specifying the encryption key for a new table

Follow these steps to specify the encryption key on a new table using the Amazon DynamoDB console or the AWS CLI.

Creating an encrypted table (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**. For the **Table name**, enter **Music**. For the primary key, enter **Artist**, and for the sort key, enter **SongTitle**, both as strings.
4. In **Settings**, make sure that **Customize settings** is selected.

 **Note**

If **Use default settings** is selected, tables are encrypted at rest with the AWS owned key at no additional cost.

5. Under **Encryption at rest**, choose an encryption type - AWS owned key, AWS managed key, or customer managed key.
 - **Owned by Amazon DynamoDB.** AWS owned key, specifically owned and managed by DynamoDB. You are not charged an additional fee for using this key.
 - **AWS managed key.** Key alias: aws/dynamodb. The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.
 - **Stored in your account, and owned and managed by you.** Customer managed key. The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.

Note

If you select to own and manage your own key, make sure the KMS Key Policy is appropriately set. For more information including examples, see [Key policy for a customer managed key](#).

6. Choose **Create table** to create the encrypted table. To confirm the encryption type, select the table details on the **Overview** tab and review the **Additional details** section.

Creating an encrypted table (AWS CLI)

Use the AWS CLI to create a table with the default AWS owned key, the AWS managed key, or a customer managed key for Amazon DynamoDB.

To create an encrypted table with the default AWS owned key

- Create the encrypted Music table as follows.

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
  AttributeName=Artist,AttributeType=S \
  AttributeName=SongTitle,AttributeType=S \
--key-schema \
  AttributeName=Artist,KeyType=HASH \
  AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
  ReadCapacityUnits=10,WriteCapacityUnits=5
```

Note

This table is now encrypted using the default AWS owned key in the DynamoDB service account.

To create an encrypted table with the AWS managed key for DynamoDB

- Create the encrypted Music table as follows.

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
  AttributeName=Artist,AttributeType=S \
  AttributeName=SongTitle,AttributeType=S \
--key-schema \
  AttributeName=Artist,KeyType=HASH \
  AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
  ReadCapacityUnits=10,WriteCapacityUnits=5 \
--sse-specification Enabled=true,SSEType=KMS
```

The SSEDescription status of the table description is set to ENABLED and the SSEType is KMS.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

To create an encrypted table with a customer managed key for DynamoDB

- Create the encrypted Music table as follows.

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
  AttributeName=Artist,AttributeType=S \
  AttributeName=SongTitle,AttributeType=S \
--key-schema \
  AttributeName=Artist,KeyType=HASH \
  AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
  ReadCapacityUnits=10,WriteCapacityUnits=5 \
--sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-
a123-ab1234a1b234
```

The SSEDescription status of the table description is set to ENABLED and the SSEType is KMS.

```
"SSEDescription": {  
    "SSEType": "KMS",  
    "Status": "ENABLED",  
    "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-ab1234a1b234",  
}
```

Updating an encryption key

You can also use the DynamoDB console or the AWS CLI to update the encryption keys of an existing table between an AWS owned key, AWS managed key, and customer managed key at any time.

Updating an encryption key (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table that you want to update.
4. Select the **Actions** dropdown, and then select the **Update settings** option.
5. Go to the **Additional settings** tab.
6. Under **Encryption**, choose **Manage encryption**.
7. Choose an encryption type:
 - **Owned by Amazon DynamoDB.** The AWS KMS key is owned and managed by DynamoDB. You are not charged an additional fee for using this key.
 - **AWS managed key** Key alias: aws/dynamodb. The key is stored in your account and is managed by AWS Key Management Service. (AWS KMS). AWS KMS charges apply.
 - **Stored in your account, and owned and managed by you.** The key is stored in your account and is managed by AWS Key Management Service. (AWS KMS). AWS KMS charges apply.

Note

If you select to own and manage your own key, make sure the KMS Key Policy is appropriately set. For more information see [Key policy for a customer managed key](#).

Then choose **Save** to update the encrypted table. To confirm the encryption type, check the table details under the **Overview** tab.

Updating an encryption key (AWS CLI)

The following examples show how to update an encrypted table using the AWS CLI.

To update an encrypted table with the default AWS owned key

- Update the encrypted Music table, as in the following example.

```
aws dynamodb update-table \
--table-name Music \
--sse-specification Enabled=false
```

Note

This table is now encrypted using the default AWS owned key in the DynamoDB service account.

To update an encrypted table with the AWS managed key for DynamoDB

- Update the encrypted Music table, as in the following example.

```
aws dynamodb update-table \
--table-name Music \
--sse-specification Enabled=true
```

The SSEDescription status of the table description is set to ENABLED and the SSEType is KMS.

```
"SSEDescription": {  
    "SSEType": "KMS",  
    "Status": "ENABLED",  
    "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-  
a123-ab1234a1b234",  
}
```

To update an encrypted table with a customer managed key for DynamoDB

- Update the encrypted Music table, as in the following example.

```
aws dynamodb update-table \  
--table-name Music \  
--sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-  
a123-ab1234a1b234
```

The SSEDescription status of the table description is set to ENABLED and the SSEType is KMS.

```
"SSEDescription": {  
    "SSEType": "KMS",  
    "Status": "ENABLED",  
    "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-  
a123-ab1234a1b234",  
}
```

Data protection in DynamoDB Accelerator

Amazon DynamoDB Accelerator (DAX) encryption at rest provides an additional layer of data protection by helping secure your data from unauthorized access to the underlying storage. Organizational policies, industry or government regulations, and compliance requirements might require the use of encryption at rest to protect your data. You can use encryption to increase the data security of your applications that are deployed in the cloud.

For more information about data protection in DAX, see [DAX encryption at rest](#).

Internet network traffic privacy

Connections are protected both between Amazon DynamoDB and on-premises applications and between DynamoDB and other AWS resources within the same AWS Region.

Required policy for endpoints

Amazon DynamoDB provides a [DescribeEndpoints](#) API that enables you to enumerate regional endpoint information. For requests from a VPC endpoint, both the IAM and Virtual Private Cloud (VPC) endpoint policies must authorize the `DescribeEndpoints` API call for the requesting Identity and Access Management (IAM) principal(s) using the IAM `dynamodb:DescribeEndpoints` action. Otherwise, access to the `DescribeEndpoints` API will be denied. The IAM and VPC endpoint policy authorization steps for `DescribeEndpoints` API calls aren't applicable when you access public endpoints of DynamoDB.

The following is an example of an endpoints policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": "(Include IAM Principals)",  
            "Action": "dynamodb:DescribeEndpoints",  
            "Resource": "*"  
        }  
    ]  
}
```

Traffic between service and on-premises clients and applications

You have two connectivity options between your private network and AWS:

- An AWS Site-to-Site VPN connection. For more information, see [What is AWS Site-to-Site VPN?](#) in the [AWS Site-to-Site VPN User Guide](#).
- An AWS Direct Connect connection. For more information, see [What is AWS Direct Connect?](#) in the [AWS Direct Connect User Guide](#).

Access to DynamoDB via the network is through AWS published APIs. Clients must support Transport Layer Security (TLS) 1.2. We recommend TLS 1.3. Clients must also support cipher suites

with Perfect Forward Secrecy (PFS), such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, you must sign requests using an access key ID and a secret access key that are associated with an IAM principal, or you can use the [AWS Security Token Service \(STS\)](#) to generate temporary security credentials to sign requests.

Traffic between AWS resources in the same Region

An Amazon Virtual Private Cloud (Amazon VPC) endpoint for DynamoDB is a logical entity within a VPC that allows connectivity only to DynamoDB. The Amazon VPC routes requests to DynamoDB and routes responses back to the VPC. For more information, see [VPC endpoints](#) in the *Amazon VPC User Guide*. For example policies that you can use to control access from VPC endpoints, see [Using IAM policies to control access to DynamoDB](#).

 **Note**

Amazon VPC endpoints are not accessible via AWS Site-to-Site VPN or AWS Direct Connect.

AWS Identity and Access Management (IAM)

AWS Identity and Access Management is an AWS service that helps an administrator securely control access to AWS resources. Administrators control who can be authenticated (signed in) and authorized (have permissions) to use Amazon DynamoDB and DynamoDB Accelerator resources. You can use IAM to manage access permissions and implement security policies for both Amazon DynamoDB and DynamoDB Accelerator. IAM is an AWS service that you can use with no additional charge.

Topics

- [Identity and Access Management for Amazon DynamoDB](#)
- [Using IAM policy conditions for fine-grained access control](#)
- [Identity and access management in DynamoDB Accelerator](#)

Identity and Access Management for Amazon DynamoDB

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use DynamoDB resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience](#)
- [Authenticating with identities](#)
- [Managing access using policies](#)
- [How Amazon DynamoDB works with IAM](#)
- [Identity-based policy examples for Amazon DynamoDB](#)
- [Troubleshooting Amazon DynamoDB identity and access](#)
- [IAM policy to prevent the purchase of DynamoDB reserved capacity](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in DynamoDB.

Service user – If you use the DynamoDB service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more DynamoDB features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in DynamoDB, see [Troubleshooting Amazon DynamoDB identity and access](#).

Service administrator – If you're in charge of DynamoDB resources at your company, you probably have full access to DynamoDB. It's your job to determine which DynamoDB features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with DynamoDB, see [How Amazon DynamoDB works with IAM](#).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to DynamoDB. To view example DynamoDB identity-based policies that you can use in IAM, see [Identity-based policy examples for Amazon DynamoDB](#).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signing AWS API requests](#) in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the *AWS IAM Identity Center User Guide* and [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see [What is IAM Identity Center?](#) in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An [*IAM user*](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the *IAM User Guide*.

An [*IAM group*](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

IAM roles

An [*IAM role*](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in

the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Creating a role for a third-party Identity Provider](#) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see [Permission sets](#) in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
 - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).
 - **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For

more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How Amazon DynamoDB works with IAM

Before you use IAM to manage access to DynamoDB, learn what IAM features are available to use with DynamoDB.

IAM features you can use with Amazon DynamoDB

IAM feature	DynamoDB support
Identity-based policies	Yes

IAM feature	DynamoDB support
Resource-based policies	Yes
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how DynamoDB and other AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

Identity-based policies for DynamoDB

Supports identity-based policies	Yes
----------------------------------	-----

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see [IAM JSON policy elements reference](#) in the *IAM User Guide*.

Identity-based policy examples for DynamoDB

To view examples of DynamoDB identity-based policies, see [Identity-based policy examples for Amazon DynamoDB](#).

Resource-based policies within DynamoDB

Supports resource-based policies	Yes
----------------------------------	-----

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

Policy actions for DynamoDB

Supports policy actions	Yes
-------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Action element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API

operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of DynamoDB actions, see [Actions defined by Amazon DynamoDB in the Service Authorization Reference](#).

Policy actions in DynamoDB use the following prefix before the action:

aws

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [  
    "aws:action1",  
    "aws:action2"  
]
```

To view examples of DynamoDB identity-based policies, see [Identity-based policy examples for Amazon DynamoDB](#).

Policy resources for DynamoDB

Supports policy resources	Yes
---------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of DynamoDB resource types and their ARNs, see [Resources defined by Amazon DynamoDB](#) in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Actions defined by Amazon DynamoDB](#).

To view examples of DynamoDB identity-based policies, see [Identity-based policy examples for Amazon DynamoDB](#).

Policy condition keys for DynamoDB

Supports service-specific policy condition keys	Yes
---	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Condition element (or Condition *block*) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

To see a list of DynamoDB condition keys, see [Condition keys for Amazon DynamoDB](#) in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Actions defined by Amazon DynamoDB](#).

To view examples of DynamoDB identity-based policies, see [Identity-based policy examples for Amazon DynamoDB](#).

Access control lists (ACLs) in DynamoDB

Supports ACLs	No
---------------	----

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with DynamoDB

Supports ABAC (tags in policies)	Partial
----------------------------------	---------

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the [condition element](#) of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see [What is ABAC?](#) in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see [Use attribute-based access control \(ABAC\)](#) in the *IAM User Guide*.

Using Temporary credentials with DynamoDB

Supports temporary credentials	Yes
--------------------------------	-----

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see [AWS services that work with IAM](#) in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see [Switching to a role \(console\)](#) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see [Temporary security credentials in IAM](#).

Cross-service principal permissions for DynamoDB

Supports forward access sessions (FAS)	Yes
--	-----

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).

Service roles for DynamoDB

Supports service roles	Yes
------------------------	-----

A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

Warning

Changing the permissions for a service role might break DynamoDB functionality. Edit service roles only when DynamoDB provides guidance to do so.

Service-linked roles for DynamoDB

Supports service-linked roles	No
-------------------------------	----

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see [AWS services that work with IAM](#). Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon DynamoDB

By default, users and roles don't have permission to create or modify DynamoDB resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see [Creating IAM policies](#) in the *IAM User Guide*.

For details about actions and resource types defined by DynamoDB, including the format of the ARNs for each of the resource types, see [Actions, resources, and condition keys for Amazon DynamoDB](#) in the *Service Authorization Reference*.

Topics

- [Policy best practices](#)
- [Using the DynamoDB console](#)

- [Allow users to view their own permissions](#)
- [Using identity-based policies with Amazon DynamoDB](#)

Policy best practices

Identity-based policies determine whether someone can create, access, or delete DynamoDB resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see [AWS managed policies](#) or [AWS managed policies for job functions](#) in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see [Policies and permissions in IAM](#) in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see [IAM Access Analyzer policy validation](#) in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see [Configuring MFA-protected API access](#) in the *IAM User Guide*.

For more information about best practices in IAM, see [Security best practices in IAM](#) in the *IAM User Guide*.

Using the DynamoDB console

To access the Amazon DynamoDB console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the DynamoDB resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the DynamoDB console, also attach the DynamoDB `ConsoleAccess` or `ReadOnly` AWS managed policy to the entities. For more information, see [Adding permissions to a user](#) in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam>ListGroupsForUser",
                "iam>ListAttachedUserPolicies",
                "iam>ListUserPolicies",
                "iam GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "execute-api:Invoke"
            ]
        }
    ]
}
```

```
        "iam:GetGroupPolicy",
        "iam:GetPolicyVersion",
        "iam:GetPolicy",
        "iam>ListAttachedGroupPolicies",
        "iam>ListGroupPolicies",
        "iam>ListPolicyVersions",
        "iam>ListPolicies",
        "iam>ListUsers"
    ],
    "Resource": "*"
}
]
}
```

Using identity-based policies with Amazon DynamoDB

This topic covers using identity-based AWS Identity and Access Management (IAM) policies with Amazon DynamoDB and provides examples. The examples show how an account administrator can attach permissions policies to IAM identities (users, groups, and roles) and thereby grant permissions to perform operations on Amazon DynamoDB resources.

The sections in this topic cover the following:

- [IAM permissions required to use the Amazon DynamoDB console](#)
- [AWS managed \(predefined\) IAM policies for Amazon DynamoDB](#)
- [Customer managed policy examples](#)

The following is an example of a permissions policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DescribeQueryScanBooksTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:account-id:table/Books"
```

```
    }
]
}
```

The preceding policy has one statement that grants permissions for three DynamoDB actions (`dynamodb:DescribeTable`, `dynamodb:Query`, and `dynamodb:Scan`) on a table in the `us-west-2` AWS Region, which is owned by the AWS account specified by `account-id`. The *Amazon Resource Name (ARN)* in the `Resource` value specifies the table that the permissions apply to.

IAM permissions required to use the Amazon DynamoDB console

To work with the DynamoDB console, a user must have a minimum set of permissions that allow the user to work with their AWS account's DynamoDB resources. In addition to these DynamoDB permissions, the console requires permissions:

- Amazon CloudWatch permissions to display metrics and graphs.
- AWS Data Pipeline permissions to export and import DynamoDB data.
- AWS Identity and Access Management permissions to access roles necessary for exports and imports.
- Amazon Simple Notification Service permissions to notify you whenever a CloudWatch alarm is triggered.
- AWS Lambda permissions to process DynamoDB Streams records.

If you create an IAM policy that is more restrictive than the minimum required permissions, the console won't function as intended for users with that IAM policy. To ensure that those users can still use the DynamoDB console, also attach the `AmazonDynamoDBReadOnlyAccess` AWS managed policy to the user, as described in [AWS managed \(predefined\) IAM policies for Amazon DynamoDB](#).

You don't need to allow minimum console permissions for users who are making calls only to the AWS CLI or the Amazon DynamoDB API.

Note

If you refer to a VPC endpoint, you will also need to authorize the `DescribeEndpoints` API call for the requesting IAM principal(s) with the IAM action (`dynamodb:DescribeEndpoints`). For more information see [Required policy for endpoints](#).

AWS managed (predefined) IAM policies for Amazon DynamoDB

AWS addresses some common use cases by providing standalone IAM policies that are created and administered by AWS. These AWS managed policies grant necessary permissions for common use cases so that you can avoid having to investigate which permissions are needed. For more information, see [AWS Managed Policies](#) in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are specific to DynamoDB and are grouped by use-case scenario:

- **AmazonDynamoDBReadOnlyAccess** – Grants read-only access to DynamoDB resources through the AWS Management Console.
- **AmazonDynamoDBFullAccess** – Grants full access to DynamoDB resources through the AWS Management Console.

You can review these AWS managed permissions policies by signing in to the IAM console and searching for specific policies there.

Important

The best practice is to create custom IAM policies that grant [least-privilege](#) to the users, roles, or groups that require them.

Customer managed policy examples

In this section, you can find policy examples that grant permissions for various DynamoDB actions. These policies work when you use AWS SDKs or the AWS CLI. When you use the console, you need to grant additional permissions that are specific to the console. For more information, see [IAM permissions required to use the Amazon DynamoDB console](#).

Note

All of the following policy examples use one of the AWS Regions and contain fictitious account IDs and table names.

Examples:

- [IAM policy to grant permissions to all DynamoDB actions on a table](#)
- [IAM policy to grant read-only permissions on items in a DynamoDB table](#)
- [IAM policy to grant access to a specific DynamoDB table and its indexes](#)
- [IAM policy to read, write, update, and delete access on a DynamoDB table](#)
- [IAM policy to separate DynamoDB environments in the same AWS account](#)
- [IAM policy to prevent the purchase of DynamoDB reserved capacity](#)
- [IAM policy to grant read access for a DynamoDB stream only \(not for the table\)](#)
- [IAM policy to allow an AWS Lambda function to access DynamoDB stream records](#)
- [IAM policy for read and write access to a DynamoDB Accelerator \(DAX\) cluster](#)

The *IAM User Guide*, includes [three additional DynamoDB examples](#):

- [Amazon DynamoDB: Allows Access to a Specific Table](#)
- [Amazon DynamoDB: Allows Access to Specific Columns](#)
- [Amazon DynamoDB: Allows Row-Level Access to DynamoDB Based on an Amazon Cognito ID](#)

IAM policy to grant permissions to all DynamoDB actions on a table

The following policy grants permissions for *all* DynamoDB actions on a table called Books. The resource ARN specified in the Resource identifies a table in a specific AWS Region. If you replace the table name Books in the Resource ARN with a wildcard character (*), *all* DynamoDB actions are allowed on *all* tables in the account. Carefully consider the possible security implications before using a wildcard character on this or any IAM policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllAPIActionsOnBooks",  
            "Effect": "Allow",  
            "Action": "dynamodb:*",  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

Note

This is an example of using a wildcard character (*) to allow *all* actions, including administration, data operations, monitoring, and purchase of DynamoDB reserved capacity. Instead, it is a best practice to explicitly specify each action to be granted and only what that user, role, or group needs.

IAM policy to grant read-only permissions on items in a DynamoDB table

The following permissions policy grants permissions for the GetItem, BatchGetItem, Scan, Query, and ConditionCheckItem DynamoDB actions only, and as a result, sets read-only access on the Books table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ReadOnlyAPIActionsOnBooks",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Scan",  
                "dynamodb:Query",  
                "dynamodb:ConditionCheckItem"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

IAM policy to grant access to a specific DynamoDB table and its indexes

The following policy grants permissions for data modification actions on a DynamoDB table called Books and all of that table's indexes. For more information about how indexes work, see [Improving data access with secondary indexes](#).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {
```

```
{  
    "Sid": "AccessTableAllIndexesOnBooks",  
    "Effect": "Allow",  
    "Action": [  
        "dynamodb:PutItem",  
        "dynamodb:UpdateItem",  
        "dynamodb:DeleteItem",  
        "dynamodb:BatchWriteItem",  
        "dynamodb:GetItem",  
        "dynamodb:BatchGetItem",  
        "dynamodb:Scan",  
        "dynamodb:Query",  
        "dynamodb:ConditionCheckItem"  
    ],  
    "Resource": [  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books",  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"  
    ]  
}  
}
```

IAM policy to read, write, update, and delete access on a DynamoDB table

Use this policy if you need to allow your application to create, read, update, and delete data in Amazon DynamoDB tables, indexes, and streams. Substitute the AWS Region name, your account ID, and the table name or wildcard character (*) where appropriate.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DynamoDBIndexAndStreamAccess",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetShardIterator",  
                "dynamodb:Scan",  
                "dynamodb:Query",  
                "dynamodb:DescribeStream",  
                "dynamodb:GetRecords",  
                "dynamodb>ListStreams"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:  
                    region:  
                    table/  
                    <table_name>  
            ]  
        }  
    ]  
}
```

```
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*",
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books/stream/*"
    ],
},
{
    "Sid": "DynamoDBTableAccess",
    "Effect": "Allow",
    "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem",
        "dynamodb:ConditionCheckItem",
        "dynamodb:PutItem",
        "dynamodb:DescribeTable",
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:Scan",
        "dynamodb:Query",
        "dynamodb:UpdateItem"
    ],
    "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
},
{
    "Sid": "DynamoDBDescribeLimitsAccess",
    "Effect": "Allow",
    "Action": "dynamodb:DescribeLimits",
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
    ]
}
]
```

To expand this policy to cover all DynamoDB tables in all AWS Regions for this account, use a wildcard (*) for the Region and table name. For example:

```
"Resource": [
    "arn:aws:dynamodb:*:123456789012:table/*",
    "arn:aws:dynamodb:*:123456789012:table/*/*index/*"
]
```

IAM policy to separate DynamoDB environments in the same AWS account

Suppose that you have separate environments where each environment maintains its own version of a table named `ProductCatalog`. If you create two `ProductCatalog` tables in the same AWS account, work in one environment might affect the other environment because of the way that permissions are set up. For example, quotas on the number of concurrent control plane operations (such as `CreateTable`) are set at the AWS account level.

As a result, each action in one environment reduces the number of operations available in the other environment. There is also a risk that the code in one environment might accidentally access tables in the other environment.

Note

If you want to separate production and test workloads to help control an event's potential "blast radius," the best practice is to create separate AWS accounts for test and production workloads. For more information, see [AWS Account Management and Separation](#).

Suppose further that you have two developers, Amit and Alice, who are testing the `ProductCatalog` table. Instead of each developer requiring a separate AWS account, your developers can share the same test AWS account. In this test account, you can create a copy of the same table for each developer to work on, such as `Alice_ProductCatalog` and `Amit_ProductCatalog`. In this case, you can create users Alice and Amit in the AWS account that you created for the test environment. You then can grant permissions to these users to perform DynamoDB actions on the tables that they own.

To grant these IAM user permissions, you can do either of the following:

- Create a separate policy for each user, and then attach each policy to its user separately. For example, you can attach the following policy to user Alice to allow her access to DynamoDB actions on the `Alice_ProductCatalog` table:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllAPIActionsOnAliceTable",  
            "Effect": "Allow",  
            "Action": [
```

```
"dynamodb>DeleteItem",
"dynamodb>DescribeContributorInsights",
"dynamodb>RestoreTableToPointInTime",
"dynamodb>ListTagsOfResource",
"dynamodb>CreateTableReplica",
"dynamodb>UpdateContributorInsights",
"dynamodb>CreateBackup",
"dynamodb>DeleteTable",
"dynamodb>UpdateTableReplicaAutoScaling",
"dynamodb>UpdateContinuousBackups",
"dynamodb>TagResource",
"dynamodb>DescribeTable",
"dynamodb>GetItem",
"dynamodb>DescribeContinuousBackups",
"dynamodb>BatchGetItem",
"dynamodb>UpdateTimeToLive",
"dynamodb>BatchWriteItem",
"dynamodb>ConditionCheckItem",
"dynamodb>UntagResource",
"dynamodb>PutItem",
"dynamodb>Scan",
"dynamodb>Query",
"dynamodb>UpdateItem",
"dynamodb>DeleteTableReplica",
"dynamodb>DescribeTimeToLive",
"dynamodb>RestoreTableFromBackup",
"dynamodb>UpdateTable",
"dynamodb>DescribeTableReplicaAutoScaling",
"dynamodb>GetShardIterator",
"dynamodb>DescribeStream",
"dynamodb>GetRecords",
"dynamodb>DescribeLimits",
"dynamodb>ListStreams"
],
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
Alice_ProductCatalog/*"
}
]
}
```

Then, you can create a similar policy with a different resource (the Amit_ProductCatalog table) for user Amit.

- Instead of attaching policies to individual users, you can use IAM policy variables to write a single policy and attach it to a group. You need to create a group and, for this example, add both users Alice and Amit to the group. The following example grants permissions to perform all DynamoDB actions on the \${aws:username}_ProductCatalog table. The policy variable \${aws:username} is replaced by the requester's user name when the policy is evaluated. For example, if Alice sends a request to add an item, the action is allowed only if Alice is adding items to the Alice_ProductCatalog table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ActionsOnUserSpecificTable",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem",  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Scan",  
                "dynamodb:Query",  
                "dynamodb:ConditionCheckItem"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/  
${aws:username}_ProductCatalog"  
        },  
        {  
            "Sid": "AdditionalPrivileges",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb>ListTables",  
                "dynamodb:DescribeTable",  
                "dynamodb:DescribeContributorInsights"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/*"  
        }  
    ]  
}
```

Note

When using IAM policy variables, you must explicitly specify the 2012-10-17 version of the IAM policy language in the policy. The default version of the IAM policy language (2008-10-17) does not support policy variables.

Instead of identifying a specific table as a resource as you normally would, you could use a wildcard character (*) to grant permissions on all tables where the table name is prefixed with the user that is making the request, as shown in the following example.

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_*"
```

IAM policy to prevent the purchase of DynamoDB reserved capacity

With Amazon DynamoDB reserved capacity, you pay a one-time, upfront fee and commit to paying for a minimum usage level at significant savings over a period of time. You can use the AWS Management Console to view and purchase reserved capacity. However, you might not want all of the users in your organization to be able to purchase reserved capacity. For more information about reserved capacity, see [Amazon DynamoDB pricing](#).

DynamoDB provides the following API operations for controlling access to reserved capacity management:

- dynamodb:DescribeReservedCapacity – Returns the reserved capacity purchases that are currently in effect.
- dynamodb:DescribeReservedCapacityOfferings – Returns details about the reserved capacity plans that are currently offered by AWS.
- dynamodb:PurchaseReservedCapacityOfferings – Performs an actual purchase of reserved capacity.

The AWS Management Console uses these API actions to display reserved capacity information and make purchases. You cannot call these operations from an application program because they can be accessed only from the console. However, you can allow or deny access to these operations in an IAM permissions policy.

The following policy allows users to view reserved capacity purchases and offerings by using the AWS Management Console — but new purchases are denied.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowReservedCapacityDescriptions",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeReservedCapacity",  
                "dynamodb:DescribeReservedCapacityOfferings"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:/*"  
        },  
        {  
            "Sid": "DenyReservedCapacityPurchases",  
            "Effect": "Deny",  
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:/*"  
        }  
    ]  
}
```

Note that this policy uses the wildcard character (*) to allow describe permissions for all, and to deny the purchase of DynamoDB reserved capacity for all.

IAM policy to grant read access for a DynamoDB stream only (not for the table)

When you enable DynamoDB Streams on a table, information is captured about every modification to items in the table. For more information, see [Change data capture for DynamoDB Streams](#).

In some cases, you might want to prevent an application from reading data from a DynamoDB table, but still allow access to that table's streams. For example, you can configure AWS Lambda to poll a stream and invoke a Lambda function when item updates are detected, and then perform additional processing.

The following actions are available for controlling access to DynamoDB streams:

- dynamodb:DescribeStream
- dynamodb:GetRecords
- dynamodb:GetShardIterator
- dynamodb>ListStreams

The following example policy grants users permissions to access the streams of a table named GameScores. The wildcard character (*) in the ARN matches any stream associated with that table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AccessGameScoresStreamOnly",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeStream",  
                "dynamodb:GetRecords",  
                "dynamodb:GetShardIterator",  
                "dynamodb>ListStreams"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/  
stream/*"  
        }  
    ]  
}
```

Note that this policy grants access to the GameScores table's streams, but not to the table itself.

IAM policy to allow an AWS Lambda function to access DynamoDB stream records

If you want certain actions to be performed based on events in a DynamoDB stream, you can write an AWS Lambda function that is triggered by these events. A Lambda function such as this needs permissions to read data from a DynamoDB stream. For more information about using Lambda with DynamoDB Streams, see [DynamoDB Streams and AWS Lambda triggers](#).

To grant permissions to Lambda, use the permissions policy that is associated with the Lambda function's IAM role (also known as an execution role). Specify this policy when you create the Lambda function.

For example, you can associate the following permissions policy with an execution role to grant Lambda permissions to perform the DynamoDB Streams actions listed.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "APIAccessForDynamoDBStreams",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeStream",  
                "dynamodb:GetRecords",  
                "dynamodb:GetShardIterator",  
                "dynamodb>ListStreams"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/  
stream/*"  
        }  
    ]  
}
```

```
        "Effect": "Allow",
        "Action": [
            "dynamodb:GetRecords",
            "dynamodb:GetShardIterator",
            "dynamodb:DescribeStream",
            "dynamodb>ListStreams"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/
stream/*"
    }
]
```

For more information, see [AWS Lambda permissions](#) in the *AWS Lambda Developer Guide*.

IAM policy for read and write access to a DynamoDB Accelerator (DAX) cluster

The following policy allows read, write, update, and delete access to a DynamoDB Accelerator (DAX) cluster, but not to the associated DynamoDB table. To use this policy, substitute the AWS Region name, your account ID, and the name of your DAX cluster.

Note

This policy gives access to DAX cluster, but not to the associated DynamoDB table. Make sure that your DAX cluster has the correct policy to perform these same operations on the DynamoDB table on your behalf.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AmazonDynamoDBDAXDataOperations",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:PutItem",
                "dax:ConditionCheckItem",
                "dax:BatchGetItem",
                "dax:BatchWriteItem",
                "dax>DeleteItem",
                "dax:Query",
                "dax:Scan"
            ]
        }
    ]
}
```

```
        "dax:UpdateItem",
        "dax:Scan"
    ],
    "Resource": "arn:aws:dax:eu-west-1:123456789012:cache/MyDAXCluster"
}
]
```

To expand this policy to cover DAX access for all AWS Regions for an account, use a wildcard character (*) for the Region name.

```
"Resource": "arn:aws:dax:*:123456789012:cache/MyDAXCluster"
```

Troubleshooting Amazon DynamoDB identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with DynamoDB and IAM.

Topics

- [I am not authorized to perform an action in DynamoDB](#)
- [I am not authorized to perform iam:PassRole](#)
- [I want to allow people outside of my AWS account to access my DynamoDB resources](#)

I am not authorized to perform an action in DynamoDB

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the mateojackson user tries to use the console to view details about a fictional *my-example-widget* resource but does not have the fictional aws:*GetWidget* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
aws:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *my-example-widget* resource using the aws : *GetWidget* action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to DynamoDB.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in DynamoDB. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:  
    iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my DynamoDB resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether DynamoDB supports these features, see [How Amazon DynamoDB works with IAM](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.

- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

IAM policy to prevent the purchase of DynamoDB reserved capacity

With Amazon DynamoDB reserved capacity, you pay a one-time, upfront fee and commit to paying for a minimum usage level at significant savings over a period of time. You can use the AWS Management Console to view and purchase reserved capacity. However, you might not want all of the users in your organization to be able to purchase reserved capacity. For more information about reserved capacity, see [Amazon DynamoDB pricing](#).

DynamoDB provides the following API operations for controlling access to reserved capacity management:

- `dynamodb:DescribeReservedCapacity` – Returns the reserved capacity purchases that are currently in effect.
- `dynamodb:DescribeReservedCapacityOfferings` – Returns details about the reserved capacity plans that are currently offered by AWS.
- `dynamodb:PurchaseReservedCapacityOfferings` – Performs an actual purchase of reserved capacity.

The AWS Management Console uses these API actions to display reserved capacity information and make purchases. You cannot call these operations from an application program because they can be accessed only from the console. However, you can allow or deny access to these operations in an IAM permissions policy.

The following policy allows users to view reserved capacity purchases and offerings by using the AWS Management Console — but new purchases are denied.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowReservedCapacityDescriptions",  
            "Effect": "Allow",  
            "Action": "dynamodb:DescribeReservedCapacityOfferings",  
            "Resource": "*"  
        },  
        {  
            "Sid": "DenyPurchaseReservedCapacityOfferings",  
            "Effect": "Deny",  
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",  
            "Resource": "*"  
        }  
    ]  
}
```

```
    "Action": [
        "dynamodb:DescribeReservedCapacity",
        "dynamodb:DescribeReservedCapacityOfferings"
    ],
    "Resource": "arn:aws:dynamodb:us-west-2:123456789012:/*"
},
{
    "Sid": "DenyReservedCapacityPurchases",
    "Effect": "Deny",
    "Action": "dynamodb:PurchaseReservedCapacityOfferings",
    "Resource": "arn:aws:dynamodb:us-west-2:123456789012:/*"
}
]
```

Note that this policy uses the wildcard character (*) to allow describe permissions for all, and to deny the purchase of DynamoDB reserved capacity for all.

Using IAM policy conditions for fine-grained access control

When you grant permissions in DynamoDB, you can specify conditions that determine how a permissions policy takes effect.

Overview

In DynamoDB, you have the option to specify conditions when granting permissions using an IAM policy (see [Identity and Access Management for Amazon DynamoDB](#)). For example, you can:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based upon the identity of that user.

In DynamoDB, you can specify conditions in an IAM policy using condition keys, as illustrated in the use case in the following section.

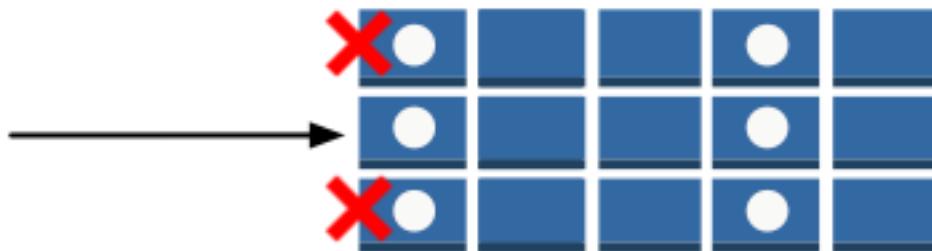
Note

Some AWS services also support tag-based conditions; however, DynamoDB does not.

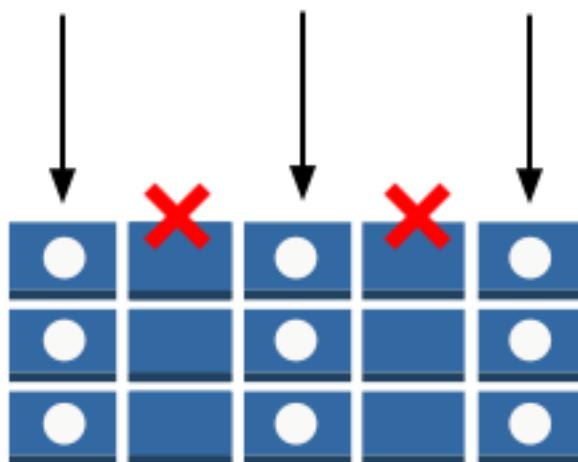
Permissions use case

In addition to controlling access to DynamoDB API actions, you can also control access to individual data items and attributes. For example, you can do the following:

- Grant permissions on a table, but restrict access to specific items in that table based on certain primary key values. An example might be a social networking app for games, where all users' saved game data is stored in a single table, but no users can access data items that they do not own, as shown in the following illustration:



- Hide information so that only a subset of attributes is visible to the user. An example might be an app that displays flight data for nearby airports, based on the user's location. Airline names, arrival and departure times, and flight numbers are all displayed. However, attributes such as pilot names or the number of passengers are hidden, as shown in the following illustration:



To implement this kind of fine-grained access control, you write an IAM permissions policy that specifies conditions for accessing security credentials and the associated permissions. You then apply the policy to users, groups, or roles that you create using the IAM console. Your IAM policy can restrict access to individual items in a table, access to the attributes in those items, or both at the same time.

You can optionally use web identity federation to control access by users who are authenticated by Login with Amazon, Facebook, or Google. For more information, see [Using web identity federation](#).

You use the IAM Condition element to implement a fine-grained access control policy. By adding a Condition element to a permissions policy, you can allow or deny access to items and attributes in DynamoDB tables and indexes, based upon your particular business requirements.

As an example, consider a mobile gaming app that lets players select from and play a variety of different games. The app uses a DynamoDB table named GameScores to keep track of high scores and other user data. Each item in the table is uniquely identified by a user ID and the name of the game that the user played. The GameScores table has a primary key consisting of a partition key (UserId) and sort key (GameTitle). Users only have access to game data associated with their user ID. A user who wants to play a game must belong to an IAM role named GameRole, which has a security policy attached to it.

To manage user permissions in this app, you could write a permissions policy such as the following:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowAccessToOnlyItemsMatchingUserID",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"  
            ],  
            "Condition": {}  
        }  
    ]  
}
```

```
"Condition":{  
    "ForAllValues:StringEquals":{  
        "dynamodb:LeadingKeys": [  
            "${www.amazon.com:user_id}"  
        ],  
        "dynamodb:Attributes": [  
            "UserId",  
            "GameTitle",  
            "Wins",  
            "Losses",  
            "TopScore",  
            "TopScoreDateTime"  
        ]  
    },  
    "StringEqualsIfExists":{  
        "dynamodb:Select":"SPECIFIC_ATTRIBUTES"  
    }  
}  
}  
]  
}
```

In addition to granting permissions for specific DynamoDB actions (Action element) on the GameScores table (Resource element), the Condition element uses the following condition keys specific to DynamoDB that limit the permissions as follows:

- dynamodb:LeadingKeys – This condition key allows users to access only the items where the partition key value matches their user ID. This ID, \${www.amazon.com:user_id}, is a substitution variable. For more information about substitution variables, see [Using web identity federation](#).
- dynamodb:Attributes – This condition key limits access to the specified attributes so that only the actions listed in the permissions policy can return values for these attributes. In addition, the StringEqualsIfExists clause ensures that the app must always provide a list of specific attributes to act upon and that the app can't request all attributes.

When an IAM policy is evaluated, the result is always either true (access is allowed) or false (access is denied). If any part of the Condition element is false, the entire policy evaluates to false and access is denied.

⚠ Important

If you use `dynamodb:Attributes`, you must specify the names of all of the primary key and index key attributes for the table and any secondary indexes that are listed in the policy. Otherwise, DynamoDB can't use these key attributes to perform the requested action.

IAM policy documents can contain only the following Unicode characters: horizontal tab (U+0009), linefeed (U+000A), carriage return (U+000D), and characters in the range U+0020 to U+00FF.

Specifying conditions: Using condition keys

AWS provides a set of predefined condition keys (AWS-wide condition keys) for all AWS services that support IAM for access control. For example, you can use the `aws:SourceIp` condition key to check the requester's IP address before allowing an action to be performed. For more information and a list of the AWS-wide keys, see [Available Keys for Conditions](#) in the IAM User Guide.

The following table shows the DynamoDB service-specific condition keys that apply to DynamoDB.

DynamoDB Condition Key	Description
<code>dynamodb:LeadingKeys</code>	Represents the first key attribute of a table—in other words, the partition key. The key name <code>LeadingKeys</code> is plural, even if the key is used with single-item actions. In addition, you must use the <code>ForAllValues</code> modifier when using <code>LeadingKeys</code> in a condition.
<code>dynamodb:Select</code>	Represents the <code>Select</code> parameter of a <code>Query</code> or <code>Scan</code> request. <code>Select</code> can be any of the following values: <ul style="list-style-type: none">• <code>ALL_ATTRIBUTES</code>• <code>ALL_PROJECTED_ATTRIBUTES</code>• <code>SPECIFIC_ATTRIBUTES</code>• <code>COUNT</code>
<code>dynamodb:Attributes</code>	Represents a list of the attribute names in a request, or the attributes that are returned from a request. <code>Attributes</code> values are named the

DynamoDB Condition Key	Description
	<p>same way and have the same meaning as the parameters for certain DynamoDB API actions, as shown following:</p> <ul style="list-style-type: none">• AttributesToGet Used by: <code>BatchGetItem</code>, <code>.GetItem</code>, <code>Query</code>, <code>Scan</code>• AttributeUpdates Used by: <code>UpdateItem</code>• Expected Used by: <code>DeleteItem</code>, <code>PutItem</code>, <code>UpdateItem</code>• Item Used by: <code>PutItem</code>• ScanFilter Used by: <code>Scan</code>
<code>dynamodb:ReturnValues</code>	Represents the <code>ReturnValues</code> parameter of a request. <code>ReturnValues</code> can be any of the following values: <ul style="list-style-type: none">• <code>ALL_OLD</code>• <code>UPDATED_OLD</code>• <code>ALL_NEW</code>• <code>UPDATED_NEW</code>• <code>NONE</code>
<code>dynamodb:ReturnConsumedCapacity</code>	Represents the <code>ReturnConsumedCapacity</code> parameter of a request. <code>ReturnConsumedCapacity</code> can be one of the following values: <ul style="list-style-type: none">• <code>TOTAL</code>• <code>NONE</code>

Limiting user access

Many IAM permissions policies allow users to access only those items in a table where the partition key value matches the user identifier. For example, the game app preceding limits access in this way so that users can only access game data that is associated with their user ID. The IAM substitution variables \${www.amazon.com:user_id}, \${graph.facebook.com:id}, and \${accounts.google.com:sub} contain user identifiers for Login with Amazon, Facebook, and Google. To learn how an application logs in to one of these identity providers and obtains these identifiers, see [Using web identity federation](#).

Note

Each of the examples in the following section sets the Effect clause to Allow and specifies only the actions, resources, and parameters that are allowed. Access is permitted only to what is explicitly listed in the IAM policy.

In some cases, it is possible to rewrite these policies so that they are deny-based (that is, setting the Effect clause to Deny and inverting all of the logic in the policy). However, we recommend that you avoid using deny-based policies with DynamoDB because they are difficult to write correctly, compared to allow-based policies. In addition, future changes to the DynamoDB API (or changes to existing API inputs) can render a deny-based policy ineffective.

Example policies: Using conditions for fine-grained access control

This section shows several policies for implementing fine-grained access control on DynamoDB tables and indexes.

Note

All examples use the us-west-2 Region and contain fictitious account IDs.

1: Grant permissions that limit access to items with a specific partition key value

The following permissions policy grants permissions that allow a set of DynamoDB actions on the GamesScore table. It uses the dynamodb:LeadingKeys condition key to limit user actions only on the items whose UserID partition key value matches the Login with Amazon unique user ID for this app.

⚠ Important

The list of actions does not include permissions for Scan because Scan returns all items regardless of the leading keys.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "FullAccessToUserItems",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:LeadingKeys": [  
                        "${www.amazon.com:user_id}"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

ⓘ Note

When using policy variables, you must explicitly specify version 2012-10-17 in the policy. The default version of the access policy language, 2008-10-17, does not support policy variables.

To implement read-only access, you can remove any actions that can modify the data. In the following policy, only those actions that provide read-only access are included in the condition.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ReadOnlyAccessToUserItems",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:LeadingKeys": [  
                        "${www.amazon.com:user_id}"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

Important

If you use `dynamodb:Attributes`, you must specify the names of all of the primary key and index key attributes, for the table and any secondary indexes that are listed in the policy. Otherwise, DynamoDB can't use these key attributes to perform the requested action.

2: Grant permissions that limit access to specific attributes in a table

The following permissions policy allows access to only two specific attributes in a table by adding the `dynamodb:Attributes` condition key. These attributes can be read, written, or evaluated in a conditional write or scan filter.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "LimitAccessToSpecificAttributes",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:UpdateItem",  
                "dynamodb:GetItem",  
                "dynamodb:Query",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Scan"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:Attributes": [  
                        "UserId",  
                        "TopScore"  
                    ]  
                },  
                "StringEqualsIfExists": {  
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES",  
                    "dynamodb:ReturnValues": [  
                        "NONE",  
                        "UPDATED_OLD",  
                        "UPDATED_NEW"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

Note

The policy takes an *allow list* approach, which allows access to a named set of attributes. You can write an equivalent policy that denies access to other attributes instead. We don't recommend this *deny list* approach. Users can determine the names of these denied attributes by following the *principle of least privilege*, as explained in Wikipedia at <http://>

en.wikipedia.org/wiki/Principle_of_least_privilege, and use an *allow list* approach to enumerate all of the allowed values, rather than specifying the denied attributes.

This policy doesn't permit PutItem, DeleteItem, or BatchWriteItem. These actions always replace the entire previous item, which would allow users to delete the previous values for attributes that they are not allowed to access.

The `StringEqualsIfExists` clause in the permissions policy ensures the following:

- If the user specifies the `Select` parameter, then its value must be `SPECIFIC_ATTRIBUTES`. This requirement prevents the API action from returning any attributes that aren't allowed, such as from an index projection.
- If the user specifies the `ReturnValues` parameter, then its value must be `NONE`, `UPDATED_OLD`, or `UPDATED_NEW`. This is required because the `UpdateItem` action also performs implicit read operations to check whether an item exists before replacing it, and so that previous attribute values can be returned if requested. Restricting `ReturnValues` in this way ensures that users can only read or write the allowed attributes.
- The `StringEqualsIfExists` clause assures that only one of these parameters — `Select` or `ReturnValues` — can be used per request, in the context of the allowed actions.

The following are some variations on this policy:

- To allow only read actions, you can remove `UpdateItem` from the list of allowed actions. Because none of the remaining actions accept `ReturnValues`, you can remove `ReturnValues` from the condition. You can also change `StringEqualsIfExists` to `StringEquals` because the `Select` parameter always has a value (`ALL_ATTRIBUTES`, unless otherwise specified).
- To allow only write actions, you can remove everything except `UpdateItem` from the list of allowed actions. Because `UpdateItem` does not use the `Select` parameter, you can remove `Select` from the condition. You must also change `StringEqualsIfExists` to `StringEquals` because the `ReturnValues` parameter always has a value (`NONE` unless otherwise specified).
- To allow all attributes whose name matches a pattern, use `StringLike` instead of `StringEquals`, and use a multi-character pattern match wildcard character (*).

3: Grant permissions to prevent updates on certain attributes

The following permissions policy limits user access to updating only the specific attributes identified by the dynamodb:Attributes condition key. The StringNotLike condition prevents an application from updating the attributes specified using the dynamodb:Attributes condition key.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "PreventUpdatesOnCertainAttributes",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:UpdateItem"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",  
            "Condition": {  
                "ForAllValues:StringNotLike": {  
                    "dynamodb:Attributes": [  
                        "FreeGamesAvailable",  
                        "BossLevelUnlocked"  
                    ]  
                },  
                "StringEquals": {  
                    "dynamodb:ReturnValues": [  
                        "NONE",  
                        "UPDATED_OLD",  
                        "UPDATED_NEW"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

Note the following:

- The UpdateItem action, like other write actions, requires read access to the items so that it can return values before and after the update. In the policy, you limit the action to accessing only the attributes that are allowed to be updated by specifying the dynamodb:ReturnValues

condition key. The condition key restricts `ReturnValues` in the request to specify only `NONE`, `UPDATED_OLD`, or `UPDATED_NEW` and doesn't include `ALL_OLD` or `ALL_NEW`.

- The `PutItem` and `DeleteItem` actions replace an entire item, and thus allows applications to modify any attributes. So when limiting an application to updating only specific attributes, you should not grant permission for these APIs.

4: Grant permissions to query only projected attributes in an index

The following permissions policy allows queries on a secondary index (`TopScoreDateTimeIndex`) by using the `dynamodb:Attributes` condition key. The policy also limits queries to requesting only specific attributes that have been projected into the index.

To require the application to specify a list of attributes in the query, the policy also specifies the `dynamodb:Select` condition key to require that the `Select` parameter of the DynamoDB Query action is `SPECIFIC_ATTRIBUTES`. The list of attributes is limited to a specific list that is provided using the `dynamodb:Attributes` condition key.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "QueryOnlyProjectedIndexAttributes",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:Query"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/  
TopScoreDateTimeIndex"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:Attributes": [  
                        "TopScoreDateTime",  
                        "GameTitle",  
                        "Wins",  
                        "Losses",  
                        "Attempts"  
                    ]  
                },  
                "StringEquals": {  
                    "ConditionKey": "TopScoreDateTime"  
                }  
            }  
        }  
    ]  
}
```

```
        "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
    }
}
]
}
```

The following permissions policy is similar, but the query must request all of the attributes that have been projected into the index.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "QueryAllIndexAttributes",
            "Effect": "Allow",
            "Action": [
                "dynamodb:Query"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/TopScoreDateTimeIndex"
            ],
            "Condition": {
                "StringEquals": {
                    "dynamodb:Select": "ALL_PROJECTED_ATTRIBUTES"
                }
            }
        }
    ]
}
```

5: Grant permissions to limit access to certain attributes and partition key values

The following permissions policy allows specific DynamoDB actions (specified in the Action element) on a table and a table index (specified in the Resource element). The policy uses the dynamodb:LeadingKeys condition key to restrict permissions to only the items whose partition key value matches the user's Facebook ID.

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
{  
    "Sid": "LimitAccessToCertainAttributesAndKeyValues",  
    "Effect": "Allow",  
    "Action": [  
        "dynamodb:UpdateItem",  
        "dynamodb:GetItem",  
        "dynamodb:Query",  
        "dynamodb:BatchGetItem"  
    ],  
    "Resource": [  
        "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",  
        "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/  
TopScoreDateTimeIndex"  
    ],  
    "Condition": {  
        "ForAllValues:StringEquals": {  
            "dynamodb:LeadingKeys": [  
                "${graph.facebook.com:id}"  
            ],  
            "dynamodb:Attributes": [  
                "attribute-A",  
                "attribute-B"  
            ]  
        },  
        "StringEqualsIfExists": {  
            "dynamodb:Select": "SPECIFIC_ATTRIBUTES",  
            "dynamodb:ReturnValues": [  
                "NONE",  
                "UPDATED_OLD",  
                "UPDATED_NEW"  
            ]  
        }  
    }  
}
```

Note the following:

- Write actions allowed by the policy (UpdateItem) can only modify attribute-A or attribute-B.
- Because the policy allows UpdateItem, an application can insert new items, and the hidden attributes will be null in the new items. If these attributes are projected into

TopScoreDateTimeIndex, the policy has the added benefit of preventing queries that cause fetches from the table.

- Applications cannot read any attributes other than those listed in dynamodb:Attributes. With this policy in place, an application must set the Select parameter to SPECIFIC_ATTRIBUTES in read requests, and only attributes in the allow list can be requested. For write requests, the application cannot set ReturnValue to ALL_OLD or ALL_NEW and it cannot perform conditional write operations based on any other attributes.

Related topics

- [Identity and Access Management for Amazon DynamoDB](#)
- [DynamoDB API permissions: Actions, resources, and conditions reference](#)

Using web identity federation

If you are writing an application targeted at large numbers of users, you can optionally use *web identity federation* for authentication and authorization. Web identity federation removes the need for creating individual users. Instead, users can sign in to an identity provider and then obtain temporary security credentials from AWS Security Token Service (AWS STS). The app can then use these credentials to access AWS services.

Web identity federation supports the following identity providers:

- Login with Amazon
- Facebook
- Google

Additional resources for web identity federation

The following resources can help you learn more about web identity federation:

- The post [Web Identity Federation using the AWS SDK for .NET](#) on the AWS Developer blog walks through how to use web identity federation with Facebook. It includes code snippets in C# that show how to assume an IAM role with web identity and how to use temporary security credentials to access an AWS resource.

- The [AWS Mobile SDK for iOS](#) and the [AWS Mobile SDK for Android](#) contain sample apps. They include code that shows how to invoke the identity providers, and then how to use the information from these providers to get and use temporary security credentials.
- The article [Web Identity Federation with Mobile Applications](#) discusses web identity federation and shows an example of how to use web identity federation to access an AWS resource.

Example policy for web identity federation

To show how you can use web identity federation with DynamoDB, revisit the *GameScores* table that was introduced in [Using IAM policy conditions for fine-grained access control](#). Here is the primary key for *GameScores*.

Table Name	Primary Key Type	Partition Key Name and Type	Sort Key Name and Type
GameScores (<u>UserId</u> , <u>GameTitle</u> , ...)	Composite	Attribute Name: UserId Type: String	Attribute Name: GameTitle Type: String

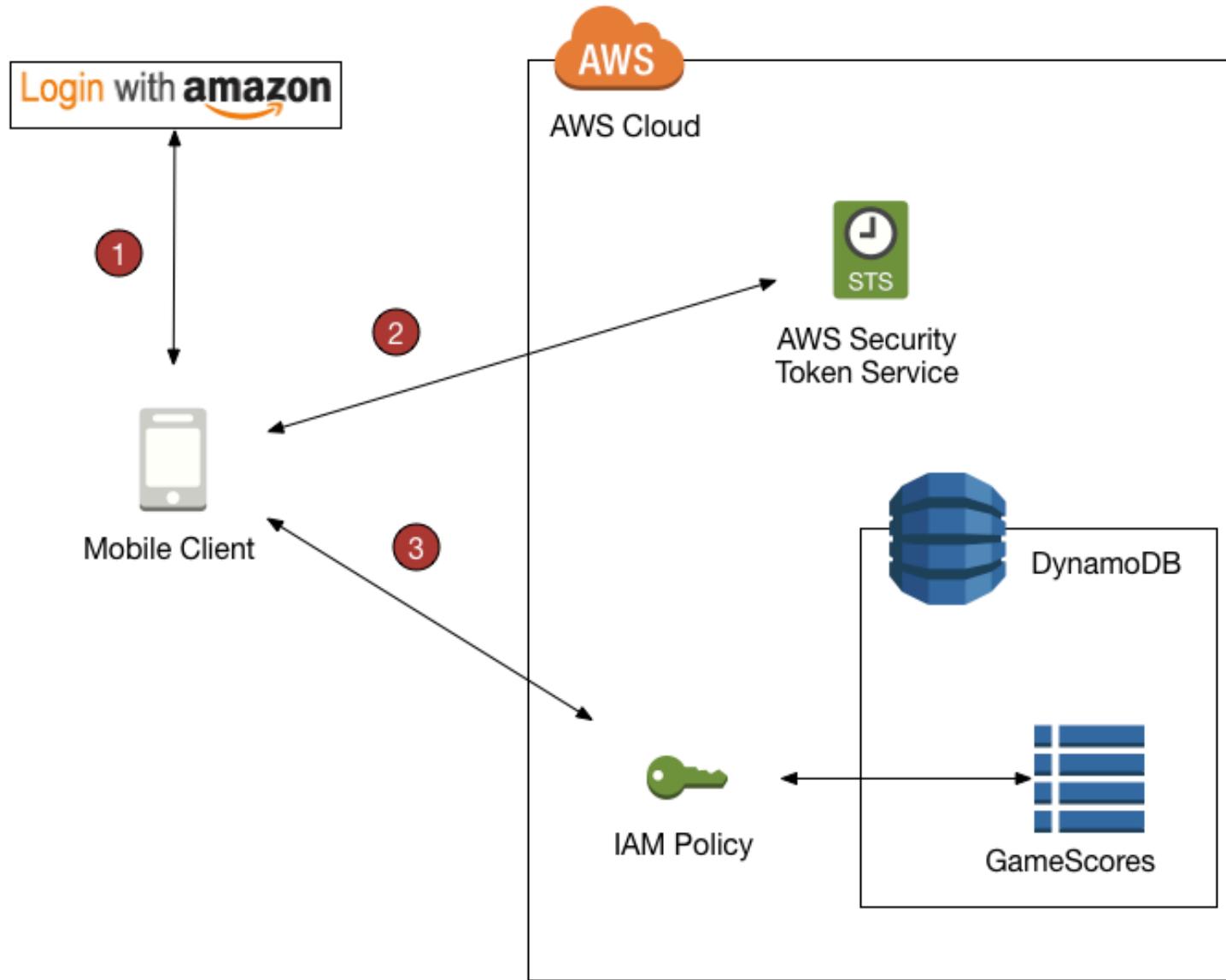
Now suppose that a mobile gaming app uses this table, and that app needs to support thousands, or even millions, of users. At this scale, it becomes very difficult to manage individual app users, and to guarantee that each user can only access their own data in the *GameScores* table. Fortunately, many users already have accounts with a third-party identity provider, such as Facebook, Google, or Login with Amazon. So it makes sense to use one of these providers for authentication tasks.

To do this using web identity federation, the app developer must register the app with an identity provider (such as Login with Amazon) and obtain a unique app ID. Next, the developer needs to create an IAM role. (For this example, this role is named *GameRole*.) The role must have an IAM policy document attached to it, specifying the conditions under which the app can access *GameScores* table.

When a user wants to play a game, they sign in to their Login with Amazon account from within the gaming app. The app then calls AWS Security Token Service (AWS STS), providing the Login with Amazon app ID and requesting membership in *GameRole*. AWS STS returns temporary AWS

credentials to the app and allows it to access the *GameScores* table, subject to the *GameRole* policy document.

The following diagram shows how these pieces fit together.



Web identity federation overview

1. The app calls a third-party identity provider to authenticate the user and the app. The identity provider returns a web identity token to the app.
2. The app calls AWS STS and passes the web identity token as input. AWS STS authorizes the app and gives it temporary AWS access credentials. The app is allowed to assume an IAM role (*GameRole*) and access AWS resources in accordance with the role's security policy.

3. The app calls DynamoDB to access the *GameScores* table. Because it has assumed the *GameRole*, the app is subject to the security policy associated with that role. The policy document prevents the app from accessing data that does not belong to the user.

Once again, here is the security policy for *GameRole* that was shown in [Using IAM policy conditions for fine-grained access control](#):

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowAccessToOnlyItemsMatchingUserID",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:LeadingKeys": [  
                        "${www.amazon.com:user_id}"  
                    ],  
                    "dynamodb:Attributes": [  
                        "UserId",  
                        "GameTitle",  
                        "Wins",  
                        "Losses",  
                        "TopScore",  
                        "TopScoreDateTime"  
                    ]  
                },  
                "StringEqualsIfExists": {  
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"  
                }  
            }  
        }  
    ]  
}
```

```
        }
    }
]
}
```

The Condition clause determines which items in *GameScores* are visible to the app. It does this by comparing the Login with Amazon ID to the UserId partition key values in *GameScores*. Only the items belonging to the current user can be processed using one of DynamoDB actions that are listed in this policy. Other items in the table cannot be accessed. Furthermore, only the specific attributes listed in the policy can be accessed.

Preparing to use web identity federation

If you are an application developer and want to use web identity federation for your app, follow these steps:

- 1. Sign up as a developer with a third-party identity provider.** The following external links provide information about signing up with supported identity providers:
 - [Login with Amazon Developer Center](#)
 - [Registration on the Facebook site](#)
 - [Using OAuth 2.0 to Access Google APIs](#) on the Google site
- 2. Register your app with the identity provider.** When you do this, the provider gives you an ID that's unique to your app. If you want your app to work with multiple identity providers, you need to obtain an app ID from each provider.
- 3. Create one or more IAM roles.** You need one role for each identity provider for each app. For example, you might create a role that can be assumed by an app where the user signed in using Login with Amazon, a second role for the same app where the user has signed in using Facebook, and a third role for the app where users sign in using Google.

As part of the role creation process, you need to attach an IAM policy to the role. Your policy document should define the DynamoDB resources required by your app, and the permissions for accessing those resources.

For more information, see [About Web Identity Federation](#) in *IAM User Guide*.

Note

As an alternative to AWS Security Token Service, you can use Amazon Cognito. Amazon Cognito is the preferred service for managing temporary credentials for mobile apps. For more information, see [Getting credentials](#) in the *Amazon Cognito Developer Guide*.

Generating an IAM policy using the DynamoDB console

The DynamoDB console can help you create an IAM policy for use with web identity federation. To do this, you choose a DynamoDB table and specify the identity provider, actions, and attributes to be included in the policy. The DynamoDB console then generates a policy that you can attach to an IAM role.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**.
3. In the list of tables, choose the table for which you want to create the IAM policy.
4. Select the **Actions** button, and choose **Create Access Control Policy**.
5. Choose the identity provider, actions, and attributes for the policy.

When the settings are as you want them, choose **Generate Policy**. The generated policy appears.

6. Choose **See Documentation**, and follow the steps required to attach the generated policy to an IAM role.

Writing your app to use web identity federation

To use web identity federation, your app must assume the IAM role that you created. From that point on, the app honors the access policy that you attached to the role.

At runtime, if your app uses web identity federation, it must follow these steps:

1. **Authenticate with a third-party identity provider.** Your app must call the identity provider using an interface that they provide. The exact way in which you authenticate the user depends on the provider and on what platform your app is running. Typically, if the user is not already signed in, the identity provider takes care of displaying a sign-in page for that provider.

After the identity provider authenticates the user, the provider returns a web identity token to your app. The format of this token depends on the provider, but is typically a very long string of characters.

2. Obtain temporary AWS security credentials. To do this, your app sends a `AssumeRoleWithWebIdentity` request to AWS Security Token Service (AWS STS). This request contains the following:

- The web identity token from the previous step
- The app ID from the identity provider
- The Amazon Resource Name (ARN) of the IAM role that you created for this identity provider for this app

AWS STS returns a set of AWS security credentials that expire after a certain amount of time (3,600 seconds, by default).

The following is a sample request and response from a `AssumeRoleWithWebIdentity` action in AWS STS. The web identity token was obtained from the Login with Amazon identity provider.

```
GET / HTTP/1.1
Host: sts.amazonaws.com
Content-Type: application/json; charset=utf-8
URL: https://sts.amazonaws.com/?ProviderId=www.amazon.com
&DurationSeconds=900&Action=AssumeRoleWithWebIdentity
&Version=2011-06-15&RoleSessionName=web-identity-federation
&RoleArn=arn:aws:iam::123456789012:role/GameRole
&WebIdentityToken=Atza|IQEBLjAsAhQluyKqyBiYZ8-kclvGTYM81e... (remaining characters
omitted)
```

```
<AssumeRoleWithWebIdentityResponse
  xmlns="https://sts.amazonaws.com/doc/2011-06-15/">
  <AssumeRoleWithWebIdentityResult>
    <SubjectFromWebIdentityToken>amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE</
    SubjectFromWebIdentityToken>
    <Credentials>
      <SessionToken>AQoDYXdzEMf//////////wEa8AP6nNDwcSLnf+cHupC... (remaining
      characters omitted)</SessionToken>
      <SecretAccessKey>8Jhi60+EWUUbbUShTEsjTxqQtM8UKvsM6XAjdA==</SecretAccessKey>
      <Expiration>2013-10-01T22:14:35Z</Expiration>
      <AccessKeyId>06198791C436IEXAMPLE</AccessKeyId>
```

```
</Credentials>
<AssumedRoleUser>
  <Arn>arn:aws:sts::123456789012:assumed-role/GameRole/web-identity-federation</Arn>
    <AssumedRoleId>AR0AJU4SA2VW5SZRF2YMG:web-identity-federation</AssumedRoleId>
  </AssumedRoleUser>
</AssumeRoleWithWebIdentityResult>
<ResponseMetadata>
  <RequestId>c265ac8e-2ae4-11e3-8775-6969323a932d</RequestId>
</ResponseMetadata>
</AssumeRoleWithWebIdentityResponse>
```

3. Access AWS resources. The response from AWS STS contains information that your app requires in order to access DynamoDB resources:

- The `AccessKeyId`, `SecretAccessKey`, and `SessionToken` fields contain security credentials that are valid for this user and this app only.
- The `Expiration` field signifies the time limit for these credentials, after which they are no longer valid.
- The `AssumedRoleId` field contains the name of a session-specific IAM role that has been assumed by the app. The app honors the access controls in the IAM policy document for the duration of this session.
- The `SubjectFromWebIdentityToken` field contains the unique ID that appears in an IAM policy variable for this particular identity provider. The following are the IAM policy variables for supported providers, and some example values for them:

Policy Variable	Example Value
<code> \${www.amazon.com:user_id}</code>	amzn1.account.AGJZDKHJKAUUS W6C44CHPEXAMPLE
<code> \${graph.facebook.com:id}</code>	123456789
<code> \${accounts.google.com:sub}</code>	123456789012345678901

For example IAM policies where these policy variables are used, see [Example policies: Using conditions for fine-grained access control](#).

For more information about how AWS STS generates temporary access credentials, see [Requesting Temporary Security Credentials](#) in *IAM User Guide*.

DynamoDB API permissions: Actions, resources, and conditions reference

When you are setting up [Identity and Access Management for Amazon DynamoDB](#) and writing a permissions policy that you can attach to an IAM identity (identity-based policies), you can use the list of [Actions, resources, and condition keys for Amazon DynamoDB](#) in the *IAM User Guide* as a reference. The page lists each DynamoDB API operation, the corresponding actions for which you can grant permissions to perform the action, and the AWS resource for which you can grant the permissions. You specify the actions in the policy's Action field, and you specify the resource value in the policy's Resource field.

You can use AWS-wide condition keys in your DynamoDB policies to express conditions. For a complete list of AWS-wide keys, see the [IAM JSON policy elements reference](#) in the *IAM User Guide*.

In addition to the AWS-wide condition keys, DynamoDB has its own specific keys that you can use in conditions. For more information, see [Using IAM policy conditions for fine-grained access control](#).

Related topics

- [Identity and Access Management for Amazon DynamoDB](#)
- [Using IAM policy conditions for fine-grained access control](#)

Identity and access management in DynamoDB Accelerator

DynamoDB Accelerator (DAX) is designed to work together with DynamoDB, to seamlessly add a caching layer to your applications. However, DAX and DynamoDB have separate access control mechanisms. Both services use AWS Identity and Access Management (IAM) to implement their respective security policies, but the security models for DAX and DynamoDB are different.

For more information about Identity and Access Management in DAX, see [DAX access control](#).

Compliance validation by industry for DynamoDB

To learn whether an AWS service is within the scope of specific compliance programs, see [AWS services in Scope by Compliance Program](#) and choose the compliance program that you are interested in. For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- [Architecting for HIPAA Security and Compliance on Amazon Web Services](#) – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

 **Note**

Not all AWS services are HIPAA eligible. For more information, see the [HIPAA Eligible Services Reference](#).

- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [AWS Customer Compliance Guides](#) – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see [Security Hub controls reference](#).
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience and disaster recovery in Amazon DynamoDB

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

If you need to replicate your data or applications over greater geographic distances, use AWS Local Regions. An AWS Local Region is a single data center designed to complement an existing AWS Region. Like all AWS Regions, AWS Local Regions are completely isolated from other AWS Regions.

For more information about AWS Regions and Availability Zones, see [AWS global infrastructure](#).

In addition to the AWS global infrastructure, Amazon DynamoDB offers several features to help support your data resiliency and backup needs.

On-demand backup and restore

DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival. For more information, see [On-Demand backup and restore for DynamoDB](#).

Point-in-time recovery

Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point in time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For more information, see [Point-in-time recovery for DynamoDB](#).

Global tables that sync across AWS regions

DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid-state disks (SSDs) and is automatically replicated across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables in sync across AWS Regions.

Infrastructure security in Amazon DynamoDB

As a managed service, Amazon DynamoDB is protected by the AWS global network security procedures that are described in [Infrastructure protection](#) located in the AWS Well-Architected Framework.

You use AWS published API calls to access DynamoDB through the network. Clients can use TLS (Transport Layer Security) version 1.2 or 1.3. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, requests must be signed using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

You can also use a virtual private cloud (VPC) endpoint for DynamoDB to enable Amazon EC2 instances in your VPC to use their private IP addresses to access DynamoDB with no exposure to the public internet. For more information, see [Using Amazon VPC endpoints to access DynamoDB](#).

Using Amazon VPC endpoints to access DynamoDB

For security reasons, many AWS customers run their applications within an Amazon Virtual Private Cloud environment (Amazon VPC). With Amazon VPC, you can launch Amazon EC2 instances into a virtual private cloud, which is logically isolated from other networks—including the public internet. With an Amazon VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security settings.

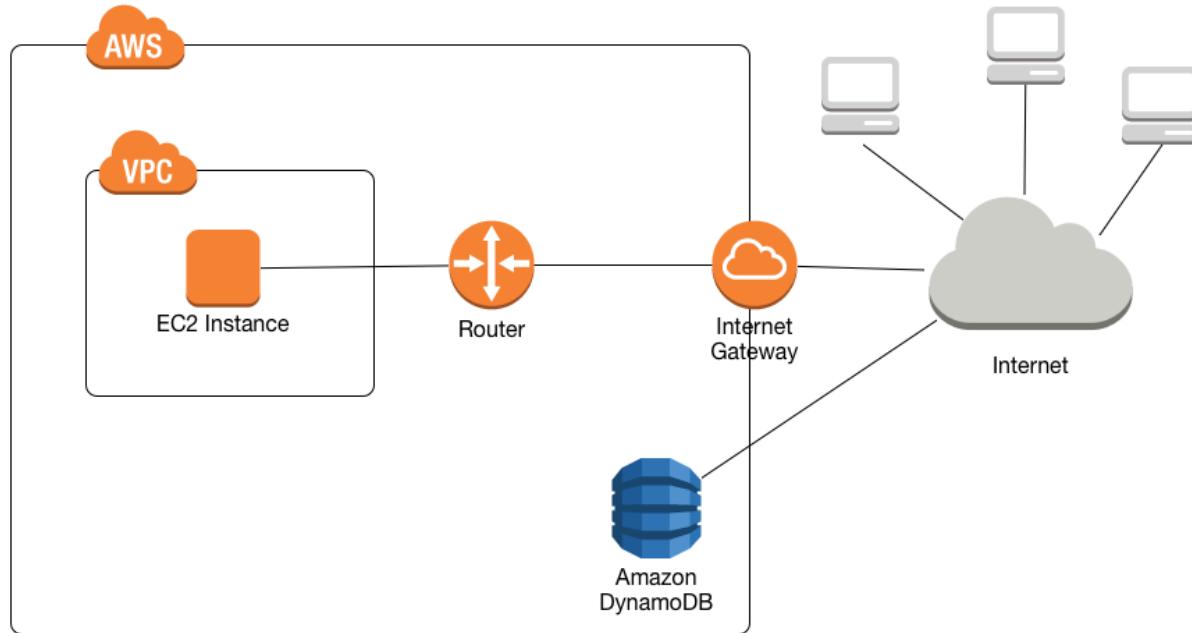
Note

If you created your AWS account after December 4, 2013, then you already have a default VPC in each AWS Region. A default VPC is ready for you to use—you can immediately start using it without having to perform any additional configuration steps.

For more information, see [Default VPC and Default Subnets](#) in the *Amazon VPC User Guide*.

To access the public internet, your VPC must have an internet gateway—a virtual router that connects your VPC to the internet. This allows applications running on Amazon EC2 in your VPC to access internet resources, such as Amazon DynamoDB.

By default, communications to and from DynamoDB use the HTTPS protocol, which protects network traffic by using SSL/TLS encryption. The following diagram shows an Amazon EC2 instance in a VPC accessing DynamoDB, by having DynamoDB use an internet gateway rather than VPC endpoints.



Many customers have legitimate privacy and security concerns about sending and receiving data across the public internet. You can address these concerns by using a virtual private network (VPN) to route all DynamoDB network traffic through your own corporate network infrastructure. However, this approach can introduce bandwidth and availability challenges.

VPC endpoints for DynamoDB can alleviate these challenges. A *VPC endpoint* for DynamoDB enables Amazon EC2 instances in your VPC to use their private IP addresses to access DynamoDB with no exposure to the public internet. Your EC2 instances do not require public IP addresses, and you don't need an internet gateway, a NAT device, or a virtual private gateway in your VPC. You use endpoint policies to control access to DynamoDB. Traffic between your VPC and the AWS service does not leave the Amazon network.

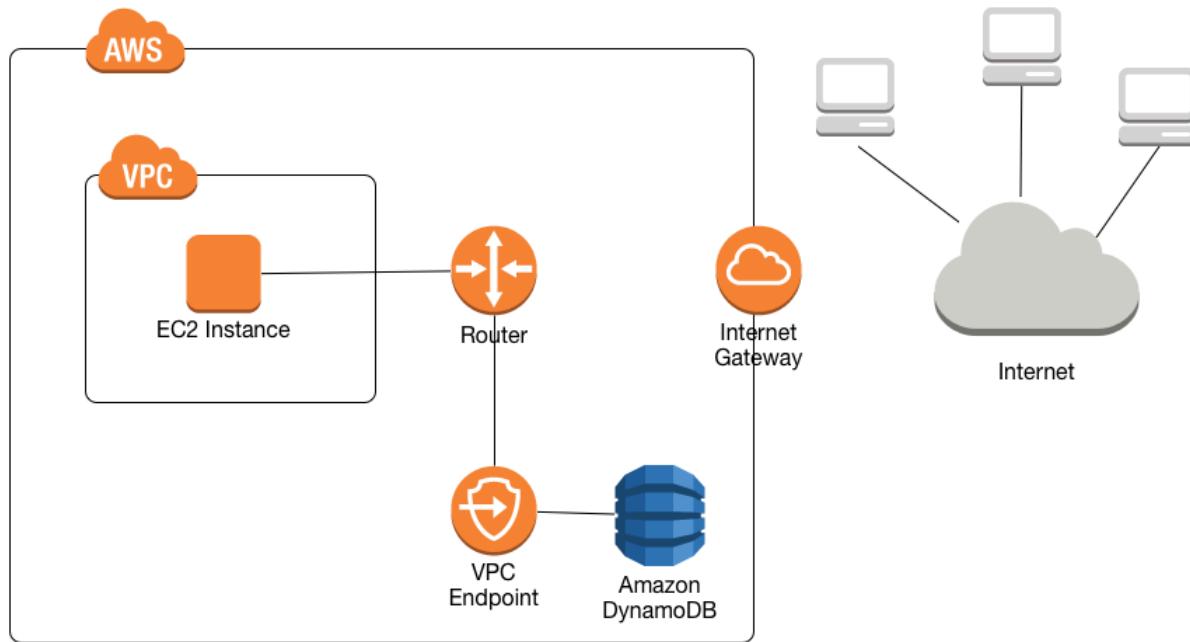
 **Note**

Even when you use public IP addresses, all VPC communication between instances and services hosted in AWS is kept private within the AWS network. Packets that originate from

the AWS network with a destination on the AWS network stay on the AWS global network, except traffic to or from AWS China Regions.

When you create a VPC endpoint for DynamoDB, any requests to a DynamoDB endpoint within the Region (for example, `dynamodb.us-west-2.amazonaws.com`) are routed to a private DynamoDB endpoint within the Amazon network. You don't need to modify your applications running on EC2 instances in your VPC. The endpoint name remains the same, but the route to DynamoDB stays entirely within the Amazon network, and does not access the public internet.

The following diagram shows how an EC2 instance in a VPC can use a VPC endpoint to access DynamoDB.



For more information, see [the section called “Tutorial: Using a VPC endpoint for DynamoDB”](#).

Sharing Amazon VPC endpoints and DynamoDB

In order to enable access to the DynamoDB service through a VPC subnet's gateway endpoint, you must have owner account permissions for that VPC subnet.

Once the VPC subnet's gateway endpoint has been granted access to DynamoDB, any AWS account with access to that subnet can use DynamoDB. This means all account users within the VPC

subnet can use any DynamoDB tables which they have access to. This includes DynamoDB tables associated with a different account than the VPC subnet. The VPC subnet owner can still restrict any particular user within the subnet from using the DynamoDB service through the gateway endpoint, at their discretion.

Tutorial: Using a VPC endpoint for DynamoDB

This section walks you through setting up and using a VPC endpoint for DynamoDB.

Topics

- [Step 1: Launch an Amazon EC2 instance](#)
- [Step 2: Configure your Amazon EC2 instance](#)
- [Step 3: Create a VPC endpoint for DynamoDB](#)
- [Step 4: \(Optional\) Clean up](#)

Step 1: Launch an Amazon EC2 instance

In this step, you launch an Amazon EC2 instance in your default Amazon VPC. You can then create and use a VPC endpoint for DynamoDB.

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Choose **Launch Instance** and do the following:

Step 1: Choose an Amazon Machine Image (AMI)

- At the top of the list of AMIs, go to **Amazon Linux AMI** and choose **Select**.

Step 2: Choose an Instance Type

- At the top of the list of instance types, choose **t2.micro**.
- Choose **Next: Configure Instance Details**.

Step 3: Configure Instance Details

- Go to **Network** and choose your default VPC.

Choose **Next: Add Storage**.

Step 4: Add Storage

- Skip this step by choosing **Next: Tag Instance**.

Step 5: Tag Instance

- Skip this step by choosing **Next: Configure Security Group**.

Step 6: Configure Security Group

- Choose **Select an existing security group**.
- In the list of security groups, choose **default**. This is the default security group for your VPC.
- Choose **Next: Review and Launch**.

Step 7: Review Instance Launch

- Choose **Launch**.

3. In the **Select an existing key pair or create a new key pair** window, do one of the following:
 - If you do not have an Amazon EC2 key pair, choose **Create a new key pair** and follow the instructions. You will be asked to download a private key file (*.pem* file); you will need this file later when you log in to your Amazon EC2 instance.
 - If you already have an existing Amazon EC2 key pair, go to **Select a key pair** and choose your key pair from the list. You must already have the private key file (*.pem* file) available in order to log in to your Amazon EC2 instance.
4. When you have configured your key pair, choose **Launch Instances**.
5. Return to the Amazon EC2 console home page and choose the instance that you launched. In the lower pane, on the **Description** tab, find the **Public DNS** for your instance. For example: `ec2-00-00-00-00.us-east-1.compute.amazonaws.com`.

Make a note of this public DNS name, because you will need it in the next step in this tutorial ([Step 2: Configure your Amazon EC2 instance](#)).

Note

It will take a few minutes for your Amazon EC2 instance to become available. Before you go on to the next step, ensure that the **Instance State** is running and that all of its **Status Checks** have passed.

Step 2: Configure your Amazon EC2 instance

When your Amazon EC2 instance is available, you will be able to log into it and prepare it for first use.

Note

The following steps assume that you are connecting to your Amazon EC2 instance from a computer running Linux. For other ways to connect, see [Connect to Your Linux Instance](#) in the Amazon EC2 User Guide for Linux Instances.

1. You will need to authorize inbound SSH traffic to your Amazon EC2 instance. To do this, you will create a new EC2 security group, and then assign the security group to your EC2 instance.
 - a. In the navigation pane, choose **Security Groups**.
 - b. Choose **Create Security Group**. In the **Create Security Group** window, do the following:
 - **Security group name**—type a name for your security group. For example: my-ssh-access
 - **Description**—type a short description for the security group.
 - **VPC**—choose your default VPC.
 - In the **Security group rules** section, choose **Add Rule** and do the following:
 - **Type**—choose SSH.
 - **Source**—choose My IP.
- When the settings are as you want them, choose **Create**.
- c. In the navigation pane, choose **Instances**.

- d. Choose the Amazon EC2 instance that you launched in [Step 1: Launch an Amazon EC2 instance](#).
 - e. Choose **Actions --> Networking --> Change Security Groups**.
 - f. In the **Change Security Groups**, select the security group that you created earlier in this procedure (for example: my-ssh-access). The existing default security group should also be selected. When the settings are as you want them, choose **Assign Security Groups**.
2. Use the ssh command to log in to your Amazon EC2 instance, as in the following example.

```
ssh -i my-keypair.pem ec2-user@public-dns-name
```

You will need to specify your private key file (*.pem* file) and the public DNS name of your instance. (See [Step 1: Launch an Amazon EC2 instance](#)).

The login ID is ec2-user. No password is required.

3. Configure your AWS credentials, as shown following. Enter your AWS access key ID, secret key, and default Region name when prompted.

```
aws configure
```

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]:
```

You are now ready to create a VPC endpoint for DynamoDB.

Step 3: Create a VPC endpoint for DynamoDB

In this step, you will create a VPC endpoint for DynamoDB and test it to make sure that it works.

1. Before you begin, verify that you can communicate with DynamoDB using its public endpoint.

```
aws dynamodb list-tables
```

The output will show a list of DynamoDB tables that you currently own. (If you don't have any tables, the list will be empty.).

- Verify that DynamoDB is an available service for creating VPC endpoints in the current AWS Region. (The command is shown in bold text, followed by example output.)

```
aws ec2 describe-vpc-endpoint-services

{
    "ServiceNames": [
        "com.amazonaws.us-east-1.s3",
        "com.amazonaws.us-east-1.dynamodb"
    ]
}
```

In the example output, DynamoDB is one of the services available, so you can proceed with creating a VPC endpoint for it.

- Determine your VPC identifier.

```
aws ec2 describe-vpcs

{
    "Vpcs": [
        {
            "VpcId": "vpc-0bbc736e",
            "InstanceTenancy": "default",
            "State": "available",
            "DhcpOptionsId": "dopt-8454b7e1",
            "CidrBlock": "172.31.0.0/16",
            "IsDefault": true
        }
    ]
}
```

In the example output, the VPC ID is vpc-0bbc736e.

- Create the VPC endpoint. For the `--vpc-id` parameter, specify the VPC ID from the previous step. Use the `--route-table-ids` parameter to associate the endpoint with your route tables.

```
aws ec2 create-vpc-endpoint --vpc-id vpc-0bbc736e --service-name com.amazonaws.us-east-1.dynamodb --route-table-ids rtb-11aa22bb
```

```
"VpcEndpoint": {  
    "PolicyDocument": "{\"Version\":\"2008-10-17\", \"Statement\":[{\"Effect\": \"Allow\", \"Principal\":\"*\", \"Action\":\"*\", \"Resource\":\"*\"}]}",  
    "VpcId": "vpc-0bbc736e",  
    "State": "available",  
    "ServiceName": "com.amazonaws.us-east-1.dynamodb",  
    "RouteTableIds": [  
        "rtb-11aa22bb"  
    ],  
    "VpcEndpointId": "vpce-9b15e2f2",  
    "CreationTimestamp": "2017-07-26T22:00:14Z"  
}  
}
```

5. Verify that you can access DynamoDB through the VPC endpoint.

```
aws dynamodb list-tables
```

If you want, you can try some other AWS CLI commands for DynamoDB. For more information, see the [AWS CLI Command Reference](#).

Step 4: (Optional) Clean up

If you want to delete the resources you have created in this tutorial, follow these procedures:

To remove your VPC endpoint for DynamoDB

1. Log in to your Amazon EC2 instance.
2. Determine the VPC endpoint ID.

```
aws ec2 describe-vpc-endpoints
```

```
{  
    "VpcEndpoint": {  
        "PolicyDocument": "{\"Version\":\"2008-10-17\", \"Statement\":[{\"Effect\": \"Allow\", \"Principal\":\"*\", \"Action\":\"*\", \"Resource\":\"*\"}]}",  
        "VpcId": "vpc-0bbc736e",  
        "State": "available",  
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",  
        "RouteTableIds": [],  
        "VpcEndpointId": "vpce-9b15e2f2",  
        "CreationTimestamp": "2017-07-26T22:00:14Z"
```

```
    }  
}
```

In the example output, the VPC endpoint ID is vpce-9b15e2f2.

3. Delete the VPC endpoint.

```
aws ec2 delete-vpc-endpoints --vpc-endpoint-ids vpce-9b15e2f2  
  
{  
    "Unsuccessful": []  
}
```

The empty array [] indicates success (there were no unsuccessful requests).

To terminate your Amazon EC2 instance

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. In the navigation pane, choose **Instances**.
3. Choose your Amazon EC2 instance.
4. Choose **Actions, Instance State, Terminate**.
5. In the confirmation window, choose **Yes, Terminate**.

AWS PrivateLink for DynamoDB

With AWS PrivateLink for DynamoDB, you can provision *interface Amazon VPC endpoints* (interface endpoints) in your virtual private cloud (Amazon VPC). These endpoints are directly accessible from applications that are on premises over VPN and AWS Direct Connect, or in a different AWS Region over [Amazon VPC peering](#). Using AWS PrivateLink and interface endpoints, you can simplify private network connectivity from your applications to DynamoDB.

Applications in your VPC do not need public IP addresses to communicate with DynamoDB interface VPC endpoints for DynamoDB operations. Interface endpoints are represented by one or more elastic network interfaces (ENIs) that are assigned private IP addresses from subnets in your Amazon VPC. Requests to DynamoDB over interface endpoints stay on the Amazon network. You can also access interface endpoints in your Amazon VPC from on-premises applications through AWS Direct Connect or AWS Virtual Private Network (AWS VPN). For more information about how

to connect your Amazon VPC with your on-premises network, see the [AWS Direct Connect User Guide](#) and the [AWS Site-to-Site VPN User Guide](#).

For general information about interface endpoints, see [Interface Amazon VPC endpoints \(AWS PrivateLink\)](#) in the *AWS PrivateLink Guide*.

Topics

- [Types of Amazon VPC endpoints for Amazon DynamoDB](#)
- [Considerations when using AWS PrivateLink for Amazon DynamoDB](#)
- [Creating an Amazon VPC endpoint](#)
- [Accessing Amazon DynamoDB interface endpoints](#)
- [Accessing DynamoDB tables and control API operations from DynamoDB interface endpoints](#)
- [Updating an on-premises DNS configuration](#)
- [Creating an Amazon VPC endpoint policy for DynamoDB](#)

Types of Amazon VPC endpoints for Amazon DynamoDB

You can use two types of Amazon VPC endpoints to access Amazon DynamoDB: *gateway endpoints* and *interface endpoints* (by using AWS PrivateLink). A *gateway endpoint* is a gateway that you specify in your route table to access DynamoDB from your Amazon VPC over the AWS network. *Interface endpoints* extend the functionality of gateway endpoints by using private IP addresses to route requests to DynamoDB from within your Amazon VPC, on premises, or from an Amazon VPC in another AWS Region by using Amazon VPC peering or AWS Transit Gateway. For more information, see [What is Amazon VPC peering?](#) and [Transit Gateway vs Amazon VPC peering](#).

Interface endpoints are compatible with gateway endpoints. If you have an existing gateway endpoint in the Amazon VPC, you can use both types of endpoints in the same Amazon VPC.

Gateway endpoints for DynamoDB	Interface endpoints for DynamoDB
In both cases, your network traffic remains on the AWS network.	
Use Amazon DynamoDB public IP addresses	Use private IP addresses from your Amazon VPC to access Amazon DynamoDB
Do not allow access from on premises	Allow access from on premises

Gateway endpoints for DynamoDB	Interface endpoints for DynamoDB
Do not allow access from another AWS Region	Allow access from an Amazon VPC endpoint in another AWS Region by using Amazon VPC peering or AWS Transit Gateway
Not billed	Billed

For more information about gateway endpoints, see [Gateway Amazon VPC endpoints](#) in the *AWS PrivateLink Guide*.

Considerations when using AWS PrivateLink for Amazon DynamoDB

Amazon VPC considerations apply to AWS PrivateLink for Amazon DynamoDB. For more information, see [Interface endpoint considerations](#) and [AWS PrivateLink quotas](#) in the *AWS PrivateLink Guide*. In addition, the following restrictions apply.

AWS PrivateLink for Amazon DynamoDB does not support the following:

- [Federal Information Processing Standard \(FIPS\) endpoints](#)
- Transport Layer Security (TLS) 1.1
- Private and Hybrid Domain Name System (DNS) services

AWS PrivateLink is currently not supported for Amazon DynamoDB Streams endpoints.

You can submit up to 50,000 requests per second for each AWS PrivateLink endpoint that you enable.

 **Note**

Network connectivity timeouts to AWS PrivateLink endpoints are not within the scope of DynamoDB error responses and need to be appropriately handled by your applications connecting to the PrivateLink endpoints.

Creating an Amazon VPC endpoint

To create an Amazon VPC interface endpoint, see [Create an endpoint](#) in the *AWS PrivateLink Guide*.

Accessing Amazon DynamoDB interface endpoints

When you create an interface endpoint, DynamoDB generates two types of endpoint-specific, DynamoDB DNS names: *Regional* and *zonal*.

- A *Regional* DNS name includes a unique Amazon VPC endpoint ID, a service identifier, the AWS Region, and `vpce.amazonaws.com` in its name. For example, for Amazon VPC endpoint ID `vpce-1a2b3c4d`, the DNS name generated might be similar to `vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com`.
- A *Zonal* DNS name includes the Availability Zone—for example, `vpce-1a2b3c4d-5e6f-us-east-1a.dynamodb.us-east-1.vpce.amazonaws.com`. You might use this option if your architecture isolates Availability Zones. For example, you could use it for fault containment or to reduce Regional data transfer costs.

Endpoint-specific DynamoDB DNS names can be resolved from the DynamoDB public DNS domain.

Accessing DynamoDB tables and control API operations from DynamoDB interface endpoints

You can use the AWS CLI or AWS SDKs to access DynamoDB tables and control API operations through DynamoDB interface endpoints.

AWS CLI examples

To access DynamoDB tables or DynamoDB control API operations through DynamoDB interface endpoints in AWS CLI commands, use the `--region` and `--endpoint-url` parameters.

Example: Create a VPC endpoint

```
aws ec2 create-vpc-endpoint \
--region us-east-1 \
--service-name dynamodb-service-name \
--vpc-id client-vpc-id \
--subnet-ids client-subnet-id \
--vpc-endpoint-type Interface \
--security-group-ids client-sg-id
```

Example: Modify a VPC endpoint

```
aws ec2 modify-vpc-endpoint \
--region us-east-1 \
--vpc-endpoint-id client-vpc-endpoint-id \
--policy-document policy-document \ #example optional parameter
--add-security-group-ids security-group-ids \ #example optional parameter
# any additional parameters needed, see PrivateLink documentation for more details
```

Example: List tables using an endpoint URL

In the following example, replace the Region `us-east-1` and the DNS name of the VPC endpoint ID `vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
aws dynamodb --region us-east-1 --endpoint https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com list-tables
```

AWS SDK examples

To access DynamoDB tables or DynamoDB control API operations through DynamoDB interface endpoints when using the AWS SDKs, update your SDKs to the latest version. Then, configure your clients to use an endpoint URL for accessing a table or DynamoDB control API operation through DynamoDB interface endpoints.

SDK for Python (Boto3)

Example: Use an endpoint URL to access a DynamoDB table

In the following example, replace the Region `us-east-1` and VPC endpoint ID `https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
ddb_client = session.client(
    service_name='dynamodb',
    region_name='us-east-1',
    endpoint_url='https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com'
)
```

SDK for Java 1.x

Example: Use an endpoint URL to access a DynamoDB table

In the following example, replace the Region us-east-1 and VPC endpoint ID https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com with your own information.

```
//client build with endpoint config
final AmazonDynamoDB dynamodb =
    AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration(
            "https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com",
            Regions.DEFAULT_REGION.getName()
        )
    ).build();
```

SDK for Java 2.x

Example: Use an endpoint URL to access an S3 bucket

In the following example, replace the Region us-east-1 and VPC endpoint ID https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com with your own information.

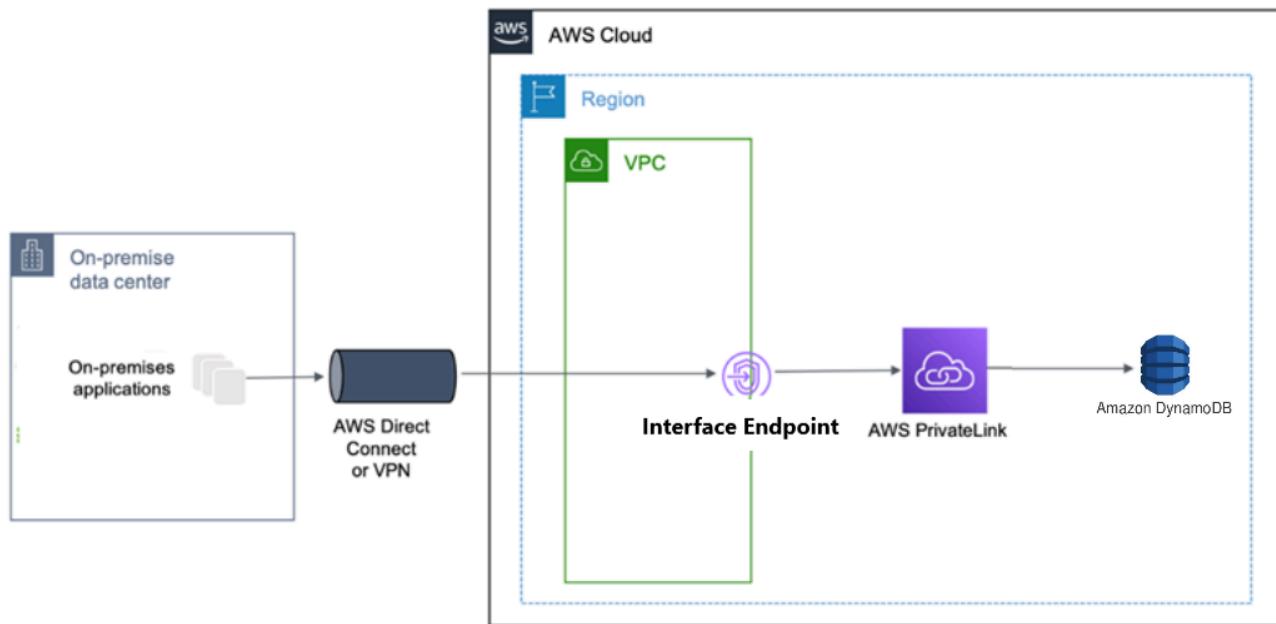
```
Region region = Region.US_EAST_1;
dynamoDbClient = DynamoDbClient.builder().region(region)
    .endpointOverride(URI.create("https://vpce-1a2b3c4d-5e6f.dynamodb.us-
east-1.vpce.amazonaws.com"))
    .build()
```

Updating an on-premises DNS configuration

When using endpoint-specific DNS names to access the interface endpoints for DynamoDB, you don't have to update your on-premises DNS resolver. You can resolve the endpoint-specific DNS name with the private IP address of the interface endpoint from the public DynamoDB DNS domain.

Using interface endpoints to access DynamoDB without a gateway endpoint or an internet gateway in the Amazon VPC

Interface endpoints in your Amazon VPC can route both in-Amazon VPC applications and on-premises applications to DynamoDB over the Amazon network, as illustrated in the following diagram.

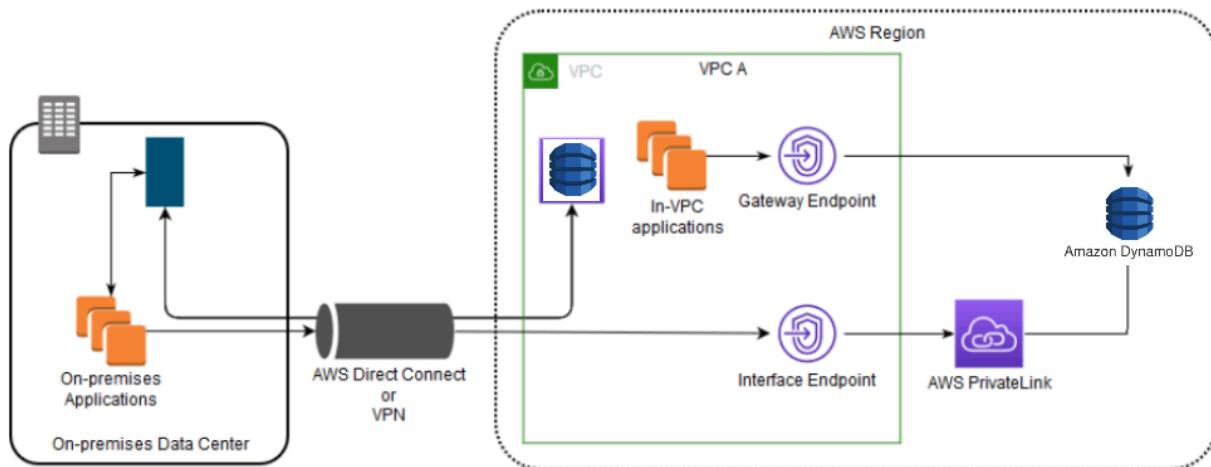


The diagram illustrates the following:

- Your on-premises network uses AWS Direct Connect or AWS VPN to connect to Amazon VPC A.
- Your applications on-premises and in Amazon VPC A use endpoint-specific DNS names to access DynamoDB through the DynamoDB interface endpoint.
- On-premises applications send data to the interface endpoint in the Amazon VPC through AWS Direct Connect (or AWS VPN). AWS PrivateLink moves the data from the interface endpoint to DynamoDB over the AWS network.
- In-Amazon VPC applications also send traffic to the interface endpoint. AWS PrivateLink moves the data from the interface endpoint to DynamoDB over the AWS network.

Using gateway endpoints and interface endpoints together in the same Amazon VPC to access DynamoDB

You can create interface endpoints and retain the existing gateway endpoint in the same Amazon VPC, as the following diagram shows. By taking this approach, you allow in-Amazon VPC applications to continue accessing DynamoDB through the gateway endpoint, which is not billed. Then, only your on-premises applications would use interface endpoints to access DynamoDB. To access DynamoDB this way, you must update your on-premises applications to use endpoint-specific DNS names for DynamoDB.



The diagram illustrates the following:

- On-premises applications use endpoint-specific DNS names to send data to the interface endpoint within the Amazon VPC through AWS Direct Connect (or AWS VPN). AWS PrivateLink moves the data from the interface endpoint to DynamoDB over the AWS network.
- Using default Regional DynamoDB names, in-Amazon VPC applications send data to the gateway endpoint that connects to DynamoDB over the AWS network.

For more information about gateway endpoints, see [Gateway Amazon VPC endpoints](#) in the *Amazon VPC User Guide*.

Creating an Amazon VPC endpoint policy for DynamoDB

You can attach an endpoint policy to your Amazon VPC endpoint that controls access to DynamoDB. The policy specifies the following information:

- The AWS Identity and Access Management (IAM) principal that can perform actions
- The actions that can be performed
- The resources on which actions can be performed

Topics

- [Example: Restricting access to a specific table from an Amazon VPC endpoint](#)

Example: Restricting access to a specific table from an Amazon VPC endpoint

You can create an endpoint policy that restricts access to only specific DynamoDB tables. This type of policy is useful if you have other AWS services in your Amazon VPC that use tables. The following table policy restricts access to only the *DOC-EXAMPLE-TABLE*. To use this endpoint policy, replace *DOC-EXAMPLE-TABLE* with the name of your table.

```
{  
    "Version": "2012-10-17",  
    "Id": "Policy1216114807515",  
    "Statement": [  
        { "Sid": "Access-to-specific-table-only",  
          "Principal": "*",  
          "Action": [  
              "dynamodb:GetItem",  
              "dynamodb:PutItem"  
          ],  
          "Effect": "Allow",  
          "Resource": ["arn:aws:dynamodb:::DOC-EXAMPLE-TABLE",  
                      "arn:aws:dynamodb:::DOC-EXAMPLE-TABLE/*"]  
        }  
    ]  
}
```

Configuration and vulnerability analysis in Amazon DynamoDB

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- [Compliance validation for Amazon DynamoDB](#)
- [Shared responsibility model](#)
- [Amazon Web Services: Overview of security processes](#)(whitepaper)

The following security best practices also address configuration and vulnerability analysis in Amazon DynamoDB:

- [Monitor DynamoDB compliance with AWS Config Rules](#)
- [Monitor DynamoDB configuration with AWS Config](#)

Security best practices for Amazon DynamoDB

Amazon DynamoDB provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Topics

- [DynamoDB preventative security best practices](#)
- [DynamoDB detective security best practices](#)

DynamoDB preventative security best practices

The following best practices can help you anticipate and prevent security incidents in Amazon DynamoDB.

Encryption at rest

DynamoDB encrypts at rest all user data stored in tables, indexes, streams, and backups using encryption keys stored in [AWS Key Management Service \(AWS KMS\)](#). This provides an additional layer of data protection by securing your data from unauthorized access to the underlying storage .

You can specify whether DynamoDB should use an AWS owned key (default encryption type), an AWS managed key, or a customer managed key to encrypt user data. For more information, see [Amazon DynamoDB Encryption at Rest](#).

Use IAM roles to authenticate access to DynamoDB

For users, applications, and other AWS services to access DynamoDB, they must include valid AWS credentials in their AWS API requests. You should not store AWS credentials directly in the application or EC2 instance. These are long-term credentials that are not automatically rotated, and therefore could have significant business impact if they are compromised. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources.

For more information, see [Identity and Access Management for Amazon DynamoDB](#).

Use IAM policies for DynamoDB base authorization

When granting permissions, you decide who is getting them, which DynamoDB APIs they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on DynamoDB resources.

You can do this by using the following:

- [AWS Managed \(predefined\) policies](#)
- [Customer managed policies](#)

Use IAM policy conditions for fine-grained access control

When you grant permissions in DynamoDB, you can specify conditions that determine how a permissions policy takes effect. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

You can specify conditions when granting permissions using an IAM policy. For example, you can do the following:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based upon the identity of that user.

For more information, see [Using IAM Policy Conditions for Fine-Grained Access Control](#).

Use a VPC endpoint and policies to access DynamoDB

If you only require access to DynamoDB from within a virtual private cloud (VPC), you should use a VPC endpoint to limit access from only the required VPC. Doing this prevents that traffic from traversing the open internet and being subject to that environment.

Using a VPC endpoint for DynamoDB allows you to control and limit access using the following:

- VPC endpoint policies – These policies are applied on the DynamoDB VPC endpoint. They allow you to control and limit API access to the DynamoDB table.
- IAM policies – By using the `aws:sourceVpce` condition on policies attached to users, groups, or roles, you can enforce that all access to the DynamoDB table is via the specified VPC endpoint.

For more information, see [Endpoints for Amazon DynamoDB](#).

Consider client-side encryption

We recommend that you plan your encryption strategy before implementing your table in DynamoDB. If you store sensitive or confidential data in DynamoDB, consider including client-side encryption in your plan. This way you can encrypt data as close as possible to its origin, and ensure its protection throughout its lifecycle. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn't available to any third party.

The [AWS Database Encryption SDK for DynamoDB](#) is a software library that helps you protect your table data before you send it to DynamoDB. It encrypts, signs, verifies, and decrypts your DynamoDB table items. You control which attributes are encrypted and signed.

DynamoDB detective security best practices

The following best practices for Amazon DynamoDB can help you detect potential security weaknesses and incidents.

Use AWS CloudTrail to monitor AWS managed KMS key usage

If you are using an [AWS managed key](#) for encryption at rest, usage of this key is logged into AWS CloudTrail. CloudTrail provides visibility into user activity by recording actions taken on your account. CloudTrail records important information about each action, including who made the request, the services used, the actions performed, parameters for the actions, and the response elements returned by the AWS service. This information helps you track changes made to your AWS resources and troubleshoot operational issues. CloudTrail makes it easier to ensure compliance with internal policies and regulatory standards.

You can use CloudTrail to audit key usage. CloudTrail creates log files that contain a history of AWS API calls and related events for your account. These log files include all AWS KMS API requests made using the AWS Management Console, AWS SDKs, and command line tools, in addition to those made through integrated AWS services. You can use these log files to get information about when the KMS key was used, the operation that was requested, the identity of the requester, the IP address that the request came from, and so on. For more information, see [Logging AWS KMS API Calls with AWS CloudTrail](#) and the [AWS CloudTrail User Guide](#).

Monitor DynamoDB operations using CloudTrail

CloudTrail can monitor both control plane events and data plane events. Control plane operations let you create and manage DynamoDB tables. They also let you work with indexes,

streams, and other objects that are dependent on tables. Data plane operations let you perform create, read, update, and delete (also called *CRUD*) actions on data in a table. Some data plane operations also let you read data from a secondary index. To enable logging of data plane events in CloudTrail, you'll need to enable logging of data plane API activity in CloudTrail. See [Logging data events for trails](#) for more information.

When activity occurs in DynamoDB, that activity is recorded in a CloudTrail event along with other AWS service events in the event history. For more information, see [Logging DynamoDB Operations by Using AWS CloudTrail](#). You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#) in the [AWS CloudTrail User Guide](#).

For an ongoing record of events in your AWS account, including events for DynamoDB, create a [trail](#). A trail enables CloudTrail to deliver log files to an Amazon Simple Storage Service (Amazon S3) bucket. By default, when you create a trail on the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

Use DynamoDB Streams to monitor data plane operations

DynamoDB is integrated with AWS Lambda so that you can create triggers—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream Amazon Resource Name (ARN) with a Lambda function that you write. Immediately after an item in the table is modified, a new record appears in the table's stream. AWS Lambda polls the stream and invokes your Lambda function synchronously when it detects new stream records. The Lambda function can perform any actions that you specify, such as sending a notification or initiating a workflow.

For an example, see [Tutorial: Using AWS Lambda with Amazon DynamoDB Streams](#). This example receives a DynamoDB event input, processes the messages that it contains, and writes some of the incoming event data to Amazon CloudWatch Logs.

Monitor DynamoDB configuration with AWS Config

Using [AWS Config](#), you can continuously monitor and record configuration changes of your AWS resources. You can also use AWS Config to inventory your AWS resources. When a change from a previous state is detected, an Amazon Simple Notification Service (Amazon SNS) notification

can be delivered for you to review and take action. Follow the guidance in [Setting Up AWS Config with the Console](#), ensuring that DynamoDB resource types are included.

You can configure AWS Config to stream configuration changes and notifications to an Amazon SNS topic. For example, when a resource is updated, you can get a notification sent to your email, so that you can view the changes. You can also be notified when AWS Config evaluates your custom or managed rules against your resources.

For an example, see [Notifications that AWS Config Sends to an Amazon SNS topic](#) in the *AWS Config Developer Guide*.

Monitor DynamoDB compliance with AWS Config rules

AWS Config continuously tracks the configuration changes that occur among your resources. It checks whether these changes violate any of the conditions in your rules. If a resource violates a rule, AWS Config flags the resource and the rule as noncompliant.

By using AWS Config to evaluate your resource configurations, you can assess how well your resource configurations comply with internal practices, industry guidelines, and regulations. AWS Config provides [AWS managed rules](#), which are predefined, customizable rules that AWS Config uses to evaluate whether your AWS resources comply with common best practices.

Tag your DynamoDB resources for identification and automation

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they enable you to categorize resources by purpose, owner, environment, or other criteria. The following are some examples:

- Security – Used to determine requirements such as encryption.
- Confidentiality – An identifier for the specific data-confidentiality level a resource supports.
- Environment – Used to distinguish between development, test, and production infrastructure.

For more information, see [AWS Tagging Strategies](#) and [Tagging for DynamoDB](#).

Monitor your usage of Amazon DynamoDB as it relates to security best practices by using AWS Security Hub.

Security Hub uses security controls to evaluate resource configurations and security standards to help you comply with various compliance frameworks.

For more information about using Security Hub to evaluate DynamoDB resources, see [Amazon DynamoDB controls](#) in the *AWS Security Hub User Guide*.

Monitoring and logging

Monitoring is an important part of maintaining the reliability, availability, and performance of DynamoDB and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your DynamoDB resources and responding to potential incidents:

Topics

- [Logging and monitoring in DynamoDB](#)
- [Logging and monitoring in DynamoDB Accelerator](#)
- [Analyzing data access using CloudWatch contributor insights for DynamoDB](#)
- [Using AWS User Notifications User Notifications with Amazon DynamoDB](#)

Logging and monitoring in DynamoDB

Monitoring is an important part of maintaining the reliability, availability, and performance of DynamoDB and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs. Before you start monitoring DynamoDB, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal DynamoDB performance in your environment, by measuring performance at various times and under different load conditions. As you monitor DynamoDB, you should consider storing historical monitoring data. This stored data will give you a baseline from which to compare current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline you should, at a minimum, monitor the following items:

- The number of read or write capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used.
- Requests that exceeded a table's provisioned write or read capacity during the specified time period, so you can determine which requests exceed the provisioned throughput quotas of a table.
- System errors, so you can determine if any requests resulted in an error.

Topics

- [Monitoring tools](#)
- [Monitoring with Amazon CloudWatch](#)
- [Logging DynamoDB operations by using AWS CloudTrail](#)

Monitoring tools

AWS provides tools that you can use to monitor DynamoDB. You can configure some of these tools to do the monitoring for you; some require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Automated monitoring tools

You can use the following automated monitoring tools to watch DynamoDB and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see [Monitoring with Amazon CloudWatch](#).
- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see [Monitoring Log Files](#) in the *Amazon CloudWatch User Guide*.

- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see [What is Amazon CloudWatch Events](#) in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see [Working with CloudTrail Log Files](#) in the *AWS CloudTrail User Guide*.

Manual monitoring tools

Another important part of monitoring DynamoDB involves manually monitoring those items that the CloudWatch alarms don't cover. The DynamoDB, CloudWatch, Trusted Advisor, and other AWS console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on Amazon DynamoDB.

- DynamoDB dashboard shows:
 - Recent alerts
 - Total capacity
 - Service health
- CloudWatch home page shows:
 - Current alarms and status
 - Graphs of alarms and resources
 - Service health status

In addition, you can use CloudWatch to do the following:

- Create [customized dashboards](#) to monitor the services you care about
- Graph metric data to troubleshoot issues and discover trends
- Search and browse all of your AWS resource metrics
- Create and edit alarms to be notified of problems

Monitoring with Amazon CloudWatch

You can monitor Amazon DynamoDB using CloudWatch, which collects and processes raw data from DynamoDB into readable, near real-time metrics. These statistics are retained for a period of time, so that you can access historical information for a better perspective on how your web

application or service is performing. By default, DynamoDB metric data is sent to CloudWatch automatically. For more information, see [What is Amazon CloudWatch?](#) and [Metrics retention](#) in the *Amazon CloudWatch User Guide*.

Topics

- [DynamoDB Metrics and dimensions](#)
- [How do I use DynamoDB metrics?](#)
- [Understanding and analyzing DynamoDB response times](#)
- [Creating CloudWatch alarms to monitor DynamoDB](#)

DynamoDB Metrics and dimensions

When you interact with DynamoDB, it sends the following metrics and dimensions to CloudWatch. You can use the following procedures to view the metrics for DynamoDB.

To view metrics (console)

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Metrics**.

 **Note**

You can also select **Usage** namespace to view DynamoDB usage metrics. For more information about usage metrics, see [AWS usage metrics](#).

3. Select the **DynamoDB** namespace.

To view metrics (CLI)

- At a command prompt, use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/DynamoDB"
```

Viewing metrics and dimensions

CloudWatch displays the following metrics for DynamoDB:

DynamoDB metrics

Note

Amazon CloudWatch aggregates these metrics at one-minute intervals:

- ConditionalCheckFailedRequests
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- ReadThrottleEvents
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledRequests
- TransactionConflict
- UserErrors
- WriteThrottleEvents

For all other DynamoDB metrics, the aggregation granularity is five minutes.

Not all statistics, such as *Average* or *Sum*, are applicable for every metric. However, all of these values are available through the Amazon DynamoDB console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics.

In the following list, each metric has a set of valid statistics that are applicable to that metric.

List of Available Metrics

- [AccountMaxReads](#)

- [AccountMaxTableLevelReads](#)
- [AccountMaxTableLevelWrites](#)
- [AccountMaxWrites](#)
- [AccountProvisionedReadCapacityUtilization](#)
- [AccountProvisionedWriteCapacityUtilization](#)
- [AgeOfOldestUnreplicatedRecord](#)
- [ConditionalCheckFailedRequests](#)
- [ConsumedChangeDataCaptureUnits](#)
- [ConsumedReadCapacityUnits](#)
- [ConsumedWriteCapacityUnits](#)
- [FailedToReplicateRecordCount](#)
- [MaxProvisionedTableReadCapacityUtilization](#)
- [MaxProvisionedTableWriteCapacityUtilization](#)
- [OnlineIndexConsumedWriteCapacity](#)
- [OnlineIndexPercentageProgress](#)
- [OnlineIndexThrottleEvents](#)
- [PendingReplicationCount](#)
- [ProvisionedReadCapacityUnits](#)
- [ProvisionedWriteCapacityUnits](#)
- [ReadThrottleEvents](#)
- [ReplicationLatency](#)
- [ReturnedBytes](#)
- [ReturnedItemCount](#)
- [ReturnedRecordsCount](#)
- [SuccessfulRequestLatency](#)
- [SystemErrors](#)
- [TimeToLiveDeletedItemCount](#)
- [ThrottledPutRecordCount](#)
- [ThrottledRequests](#)
- [TransactionConflict](#)

- [UserErrors](#)
- [WriteThrottleEvents](#)

AccountMaxReads

The maximum number of read capacity units that can be used by an account. This limit does not apply to on-demand tables or global secondary indexes.

Units: Count

Valid Statistics:

- Maximum – The maximum number of read capacity units that can be used by an account.

AccountMaxTableLevelReads

The maximum number of read capacity units that can be used by a table or global secondary index of an account. For on-demand tables this limit caps the maximum read request units a table or a global secondary index can use.

Units: Count

Valid Statistics:

- Maximum – The maximum number of read capacity units that can be used by a table or global secondary index of the account.

AccountMaxTableLevelWrites

The maximum number of write capacity units that can be used by a table or global secondary index of an account. For on-demand tables this limit caps the maximum write request units a table or a global secondary index can use.

Units: Count

Valid Statistics:

- Maximum – The maximum number of write capacity units that can be used by a table or global secondary index of the account.

AccountMaxWrites

The maximum number of write capacity units that can be used by an account. This limit does not apply to on-demand tables or global secondary indexes.

Units: Count

Valid Statistics:

- Maximum – The maximum number of write capacity units that can be used by an account.

AccountProvisionedReadCapacityUtilization

The percentage of provisioned read capacity units utilized by an account.

Units: Percent

Valid Statistics:

- Maximum – The maximum percentage of provisioned read capacity units utilized by the account.
- Minimum – The minimum percentage of provisioned read capacity units utilized by the account.
- Average – The average percentage of provisioned read capacity units utilized by the account.
The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

AccountProvisionedWriteCapacityUtilization

The percentage of provisioned write capacity units utilized by an account.

Units: Percent

Valid Statistics:

- Maximum – The maximum percentage of provisioned write capacity units utilized by the account.
- Minimum – The minimum percentage of provisioned write capacity units utilized by the account.
- Average – The average percentage of provisioned write capacity units utilized by the account.
The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

AgeOfOldestUnreplicatedRecord

The elapsed time since a record yet to be replicated to the Kinesis data stream first appeared in the DynamoDB table.

Units: Milliseconds

Dimensions: TableName, DelegatedOperation

Valid Statistics:

- Maximum.
- Minimum.
- Average.

ConditionalCheckFailedRequests

The number of failed attempts to perform conditional writes. The PutItem, UpdateItem, and DeleteItem operations let you provide a logical condition that must evaluate to true before the operation can proceed. If this condition evaluates to false, ConditionalCheckFailedRequests is incremented by one. ConditionalCheckFailedRequests is also incremented by one for PartiQL Update and Delete statements where a logical condition is provided and that condition evaluates to false.

 **Note**

A failed conditional write will result in an HTTP 400 error (Bad Request). These events are reflected in the ConditionalCheckFailedRequests metric, but not in the UserErrors metric.

Units: Count

Dimensions: TableName

Valid Statistics:

- Minimum
- Maximum
- Average

- SampleCount
- Sum

ConsumedChangeDataCaptureUnits

The number of consumed change data capture units.

Units: Count

Dimensions: TableName, DelegatedOperation

Valid Statistics:

- Minimum
- Maximum
- Average

ConsumedReadCapacityUnits

The number of read capacity units consumed over the specified time period for both provisioned and on-demand capacity, so you can track how much of your throughput is used. You can retrieve the total consumed read capacity for a table and all of its global secondary indexes, or for a particular global secondary index. For more information, see [Read/Write Capacity Mode](#).

The TableName dimension returns the ConsumedReadCapacityUnits for the table, but not for any global secondary indexes. To view ConsumedReadCapacityUnits for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

Note

In Amazon DynamoDB, the consumed capacity metric is reported to CloudWatch at one-minute intervals as an average value. This means that short, intense spikes in capacity consumption lasting just a second may not be accurately reflected in the CloudWatch graph, potentially leading to a lower apparent consumption rate for that minute.

Use the Sum statistic to calculate the consumed throughput. For example, get the Sum value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average ConsumedReadCapacityUnits per second. You can compare the calculated value to the provisioned throughput value that you provide DynamoDB.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Minimum – The minimum number of read capacity units consumed by any individual request to the table or index.
- Maximum – The maximum number of read capacity units consumed by any individual request to the table or index.
- Average – The average per-request read capacity consumed.

 **Note**

The Average value is influenced by periods of inactivity where the sample value will be zero.

- Sum – The total read capacity units consumed. This is the most useful statistic for the ConsumedReadCapacityUnits metric.
- SampleCount – The number of read requests to DynamoDB, even if no read capacity was consumed.

 **Note**

The SampleCount value is influenced by periods of inactivity where the sample value will be zero.

ConsumedWriteCapacityUnits

The number of write capacity units consumed over the specified time period for both provisioned and on-demand capacity, so you can track how much of your throughput is used. You can retrieve the total consumed write capacity for a table and all of its global secondary indexes, or for a particular global secondary index. For more information, see [Read/Write Capacity Mode](#).

The TableName dimension returns the ConsumedWriteCapacityUnits for the table, but not for any global secondary indexes. To view ConsumedWriteCapacityUnits for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

Note

Use the Sum statistic to calculate the consumed throughput. For example, get the Sum value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average ConsumedWriteCapacityUnits per second (recognizing that this average does not highlight any large but brief spikes in write activity that occurred during that minute). You can compare the calculated value to the provisioned throughput value that you provide DynamoDB.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Minimum – The minimum number of write capacity units consumed by any individual request to the table or index.
- Maximum – The maximum number of write capacity units consumed by any individual request to the table or index.
- Average – The average per-request write capacity consumed.

Note

The Average value is influenced by periods of inactivity where the sample value will be zero.

- Sum – The total write capacity units consumed. This is the most useful statistic for the ConsumedWriteCapacityUnits metric.
- SampleCount – The number of write requests to DynamoDB, even if no write capacity was consumed.

Note

The SampleCount value is influenced by periods of inactivity where the sample value will be zero.

FailedToReplicateRecordCount

The number of records that DynamoDB failed to replicate to your Kinesis data stream.

Units: Count

Dimensions: TableName, DelegatedOperation

Valid Statistics:

- Sum

MaxProvisionedTableReadCapacityUtilization

The percentage of provisioned read capacity utilized by the highest provisioned read table or global secondary index of an account.

Units: Percent

Valid Statistics:

- Maximum – The maximum percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of an account.
- Minimum – The minimum percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of an account.
- Average – The average percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

MaxProvisionedTableWriteCapacityUtilization

The percentage of provisioned write capacity utilized by the highest provisioned write table or global secondary index of an account.

Units: Percent

Valid Statistics:

- Maximum – The maximum percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of an account.

- Minimum – The minimum percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of an account.
- Average – The average percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

OnlineIndexConsumedWriteCapacity

The number of write capacity units consumed when adding a new global secondary index to a table. If the write capacity of the index is too low, incoming write activity during the backfill phase might be throttled. This can increase the time it takes to create the index. You should monitor this statistic while the index is being built to determine whether the write capacity of the index is underprovisioned.

You can adjust the write capacity of the index using the `UpdateTable` operation, even while the index is still being built.

The `ConsumedWriteCapacityUnits` metric for the index does not include the write throughput consumed during index creation.

Note

This metric may not be emitted if the new global secondary index's backfill phase completes quickly (less than a few minutes), which may occur if the base table has few or no items to backfill in the index.

Units: Count

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount

- Sum

OnlineIndexPercentageProgress

The percentage of completion when a new global secondary index is being added to a table. DynamoDB must first allocate resources for the new index, and then backfill attributes from the table into the index. For large tables, this process might take a long time. You should monitor this statistic to view the relative progress as DynamoDB builds the index.

Units: Count

Dimensions: TableName , GlobalSecondaryIndexName

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

OnlineIndexThrottleEvents

The number of write throttle events that occur when adding a new global secondary index to a table. These events indicate that the index creation will take longer to complete, because incoming write activity is exceeding the provisioned write throughput of the index.

You can adjust the write capacity of the index using the `UpdateTable` operation, even while the index is still being built.

The `WriteThrottleEvents` metric for the index does not include any throttle events that occur during index creation.

Units: Count

Dimensions: TableName , GlobalSecondaryIndexName

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

PendingReplicationCount

Metric for [Global tables version 2017.11.29 \(Legacy\)](#) (global tables only). The number of item updates that are written to one replica table, but that have not yet been written to another replica in the global table.

Units: Count

Dimensions: TableName, ReceivingRegion

Valid Statistics:

- Average
- Sample Count
- Sum

ProvisionedReadCapacityUnits

The number of provisioned read capacity units for a table or a global secondary index. The TableName dimension returns the ProvisionedReadCapacityUnits for the table, but not for any global secondary indexes. To view ProvisionedReadCapacityUnits for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

Units:Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Minimum – The lowest setting for provisioned read capacity. If you use UpdateTable to increase read capacity, this metric shows the lowest value of provisioned ReadCapacityUnits during this time period.

- Maximum – The highest setting for provisioned read capacity. If you use `UpdateTable` to decrease read capacity, this metric shows the highest value of provisioned `ReadCapacityUnits` during this time period.
- Average – The average provisioned read capacity. The `ProvisionedReadCapacityUnits` metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

ProvisionedWriteCapacityUnits

The number of provisioned write capacity units for a table or a global secondary index.

The `TableName` dimension returns the `ProvisionedWriteCapacityUnits` for the table, but not for any global secondary indexes. To view `ProvisionedWriteCapacityUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

Units: Count

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- Minimum – The lowest setting for provisioned write capacity. If you use `UpdateTable` to increase write capacity, this metric shows the lowest value of provisioned `WriteCapacityUnits` during this time period.
- Maximum – The highest setting for provisioned write capacity. If you use `UpdateTable` to decrease write capacity, this metric shows the highest value of provisioned `WriteCapacityUnits` during this time period.
- Average – The average provisioned write capacity. The `ProvisionedWriteCapacityUnits` metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

ReadThrottleEvents

Requests to DynamoDB that exceed the provisioned read capacity units for a table or a global secondary index.

A single request can result in multiple events. For example, a `BatchGetItem` that reads 10 items is processed as 10 `GetItem` events. For each event, `ReadThrottleEvents` is incremented by one

if that event is throttled. The ThrottledRequests metric for the entire BatchGetItem is not incremented unless *all 10* of the GetItem events are throttled.

The TableName dimension returns the ReadThrottleEvents for the table, but not for any global secondary indexes. To view ReadThrottleEvents for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- SampleCount
- Sum

ReplicationLatency

(This metric is for DynamoDB global tables.) The elapsed time between an updated item appearing in the DynamoDB stream for one replica table, and that item appearing in another replica in the global table.

Units: Milliseconds

Dimensions: TableName, ReceivingRegion

Valid Statistics:

- Average
- Minimum
- Maximum

ReturnedBytes

The number of bytes returned by GetRecords operations (Amazon DynamoDB Streams) during the specified time period.

Units: Bytes

Dimensions: Operation, StreamLabel, TableName

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

ReturnedItemCount

The number of items returned by Query, Scan or ExecuteStatement (select) operations during the specified time period.

The number of items *returned* is not necessarily the same as the number of items that were evaluated. For example, suppose that you requested a Scan on a table or an index that had 100 items, but specified a FilterExpression that narrowed the results so that only 15 items were returned. In this case, the response from Scan would contain a ScanCount of 100 and a Count of 15 returned items.

Units: Count

Dimensions: TableName, Operation

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

ReturnedRecordsCount

The number of stream records returned by GetRecords operations (Amazon DynamoDB Streams) during the specified time period.

Units: Count

Dimensions: Operation, StreamLabel, TableName

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

SuccessfulRequestLatency

The latency of successful requests to DynamoDB or Amazon DynamoDB Streams during the specified time period. SuccessfulRequestLatency can provide two different kinds of information:

- The elapsed time for successful requests (Minimum, Maximum, Sum, or Average).
- The number of successful requests (SampleCount).

SuccessfulRequestLatency reflects activity only within DynamoDB or Amazon DynamoDB Streams, and does not take into account network latency or client-side activity.

Units: Milliseconds

Dimensions: TableName, Operation, StreamLabel

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount

SystemErrors

The requests to DynamoDB or Amazon DynamoDB Streams that generate an HTTP 500 status code during the specified time period. An HTTP 500 usually indicates an internal service error.

Units: Count

Dimensions: TableName, Operation

Valid Statistics:

- Sum
- SampleCount

TimeToLiveDeletedItemCount

The number of items deleted by Time to Live (TTL) during the specified time period. This metric helps you monitor the rate of TTL deletions on your table.

Units: Count

Dimensions: TableName

Valid Statistics:

- Sum

ThrottledPutRecordCount

The number of records that were throttled by your Kinesis data stream due to insufficient Kinesis Data Streams capacity.

Units: Count

Dimensions: TableName, DelegatedOperation

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount

ThrottledRequests

Requests to DynamoDB that exceed the provisioned throughput limits on a resource (such as a table or an index).

ThrottledRequests is incremented by one if any event within a request exceeds a provisioned throughput limit. For example, if you update an item in a table with global secondary indexes, there are multiple events—a write to the table, and a write to each index. If one or more of these events are throttled, then ThrottledRequests is incremented by one.

Note

In a batch request (`BatchGetItem` or `BatchWriteItem`), ThrottledRequests is incremented only if *every* request in the batch is throttled.

If any individual request within the batch is throttled, one of the following metrics is incremented:

- `ReadThrottleEvents` – For a throttled `GetItem` event within `BatchGetItem`.
- `WriteThrottleEvents` – For a throttled `PutItem` or `DeleteItem` event within `BatchWriteItem`.

To gain insight into which event is throttling a request, compare ThrottledRequests with the `ReadThrottleEvents` and `WriteThrottleEvents` for the table and its indexes.

Note

A throttled request will result in an HTTP 400 status code. All such events are reflected in the ThrottledRequests metric, but not in the UserErrors metric.

Units: Count

Dimensions: TableName, Operation

Valid Statistics:

- Sum
- SampleCount

TransactionConflict

Rejected item-level requests due to transactional conflicts between concurrent requests on the same items. For more information, see [Transaction Conflict Handling in DynamoDB](#).

Units: Count

Dimensions: TableName

Valid Statistics:

- Sum – The number of rejected item-level requests due to transaction conflicts.

 **Note**

If multiple item-level requests within a call to `TransactWriteItems` or `TransactGetItems` were rejected, Sum is incremented by one for each item-level Put, Update, Delete, or Get request.

- SampleCount – The number of rejected requests due to transaction conflicts.

 **Note**

If multiple item-level requests within a call to `TransactWriteItems` or `TransactGetItems` are rejected, SampleCount is only incremented by one.

- Min – The minimum number of rejected item-level requests within a call to `TransactWriteItems`, `TransactGetItems`, `PutItem`, `UpdateItem`, or `DeleteItem`.
- Max – The maximum number of rejected item-level requests within a call to `TransactWriteItems`, `TransactGetItems`, `PutItem`, `UpdateItem`, or `DeleteItem`.
- Average – The average number of rejected item-level requests within a call to `TransactWriteItems`, `TransactGetItems`, `PutItem`, `UpdateItem`, or `DeleteItem`.

UserErrors

Requests to DynamoDB or Amazon DynamoDB Streams that generate an HTTP 400 status code during the specified time period. An HTTP 400 usually indicates a client-side error, such as an invalid combination of parameters, an attempt to update a nonexistent table, or an incorrect request signature.

Some examples of exceptions that will log metrics related to `UserErrors` would be:

- `ResourceNotFoundException`
- `ValidationException`
- `TransactionConflict`

All such events are reflected in the `UserErrors` metric, except for the following:

- *ProvisionedThroughputExceededException* – See the `ThrottledRequests` metric in this section.
- *ConditionalCheckFailedException* – See the `ConditionalCheckFailedRequests` metric in this section.

`UserErrors` represents the aggregate of HTTP 400 errors for DynamoDB or Amazon DynamoDB Streams requests for the current AWS Region and the current AWS account.

Units: Count

Valid Statistics:

- Sum
- SampleCount

WriteThrottleEvents

Requests to DynamoDB that exceed the provisioned write capacity units for a table or a global secondary index.

A single request can result in multiple events. For example, a `PutItem` request on a table with three global secondary indexes would result in four events—the table write, and each of the three index writes. For each event, the `WriteThrottleEvents` metric is incremented by one if that event is throttled. For single `PutItem` requests, if any of the events are throttled, `ThrottledRequests` is also incremented by one. For `BatchWriteItem`, the `ThrottledRequests` metric for the entire `BatchWriteItem` is not incremented unless all of the individual `PutItem` or `DeleteItem` events are throttled.

The `TableName` dimension returns the `WriteThrottleEvents` for the table, but not for any global secondary indexes. To view `WriteThrottleEvents` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Sum
- SampleCount

Usage metrics

Usage metrics in CloudWatch allow you to proactively manage usage by visualizing metrics in the CloudWatch console, creating custom dashboards, detecting changes in activity with CloudWatch anomaly detection, and configuring alarms that alert you when usage approaches a threshold.

DynamoDB also integrates these usage metrics with Service Quotas. You can use CloudWatch to manage your account's use of your service quotas. For more information, see [Visualizing your service quotas and setting alarms](#)

List of Available Usage Metrics

- [AccountProvisionedWriteCapacityUnits](#)
- [AccountProvisionedReadCapacityUnits](#)
- [TableCount](#)

AccountProvisionedWriteCapacityUnits

The sum of write capacity units provisioned for all tables and global secondary indexes of an account.

Units: Count

Valid Statistics:

- Minimum – The lowest number of provisioned write capacity units during a time period.
- Maximum – The highest number of provisioned write capacity units during a time period.
- Average - The average number provisioned write capacity units account during a time period.

This metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

AccountProvisionedReadCapacityUnits

The sum of read capacity units provisioned for all tables and global secondary indexes of an account.

Units: Count

Valid Statistics:

- Minimum – The lowest number of provisioned read capacity units during a time period.
- Maximum – The highest number of provisioned read capacity units during a time period.
- Average - The average number provisioned read capacity units account during a time period.

This metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

TableCount

The number of active tables of an account.

Units: Count

Valid Statistics:

- Minimum – The lowest number of tables during a time period.
- Maximum – The highest number of tables during a time period.
- Average - The average number tables during a time period.

Understanding metrics and dimensions for DynamoDB

The metrics for DynamoDB are qualified by the values for the account, table name, global secondary index name, or operation. You can use the CloudWatch console to retrieve DynamoDB data along any of the dimensions in the table below.

List of Available Dimensions

- [DelegatedOperation](#)

- [GlobalSecondaryIndexName](#)
- [Operation](#)
- [OperationType](#)
- [Verb](#)
- [ReceivingRegion](#)
- [StreamLabel](#)
- [TableName](#)

DelegatedOperation

This dimension limits the data to operations DynamoDB performs on your behalf. The following operations are captured:

- Change data capture for Kinesis Data Streams.

GlobalSecondaryIndexName

This dimension limits the data to a global secondary index on a table. If you specify GlobalSecondaryIndexName, you must also specify TableName.

Operation

This dimension limits the data to one of the following DynamoDB operations:

- PutItem
- DeleteItem
- UpdateItem
- GetItem
- BatchGetItem
- Scan
- Query
- BatchWriteItem
- TransactWriteItems
- TransactGetItems

- `ExecuteTransaction`
- `BatchExecuteStatement`
- `ExecuteStatement`

In addition, you can limit the data to the following Amazon DynamoDB Streams operation:

- `GetRecords`

OperationType

This dimension limits the data to one of the following operation types:

- `Read`
- `Write`

This dimension is emitted for `ExecuteTransaction` and `BatchExecuteStatement` requests.

Verb

This dimension limits the data to one of the following DynamoDB PartiQL verbs:

- `Insert: PartiQLInsert`
- `Select: PartiQLSelect`
- `Update: PartiQLUpdate`
- `Delete: PartiQLDelete`

This dimension is emitted for the `ExecuteStatement` operation.

ReceivingRegion

This dimension limits the data to a particular AWS region. It is used with metrics originating from replica tables within a DynamoDB global table.

StreamLabel

This dimension limits the data to a specific stream label. It is used with metrics originating from Amazon DynamoDB Streams `GetRecords` operations.

TableName

This dimension limits the data to a specific table. This value can be any table name in the current region and the current AWS account.

How do I use DynamoDB metrics?

The metrics reported by DynamoDB provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, not a comprehensive list.

How can I?	Relevant metrics
How can I monitor the rate of TTL deletions on my table?	You can monitor <code>TimeToLiveDeletedItemCount</code> over the specified time period, to track the rate of TTL deletions on your table. For an example of a server-less application using the <code>TimeToLiveDeletedItemCount</code> metric, see Automatically archive items to S3 using DynamoDB time to live (TTL) with AWS Lambda and Amazon Data Firehose .
How can I determine how much of my provisioned throughput is being used?	You can monitor <code>ConsumedReadCapacityUnits</code> or <code>ConsumedWriteCapacityUnits</code> over the specified time period, to track how much of your provisioned throughput is being used.
How can I determine which requests exceed the provisioned throughput quotas of a table?	<code>ThrottledRequests</code> is incremented by one if any event within a request exceeds a provisioned throughput quota. Then, to gain insight into which event is throttling a request, compare <code>ThrottledRequests</code> with the <code>ReadThrottleEvents</code> and <code>WriteThrottleEvents</code> metrics for the table and its indexes.
How can I determine if any system errors occurred?	You can monitor <code>SystemErrors</code> to determine if any requests resulted in an HTTP 500 (server error) code. Typically, this metric should be equal to zero. If it isn't, then you might want to investigate.

How can I?	Relevant metrics
	<p>Note</p> <p>You might encounter internal server errors while working with items. These are expected during the lifetime of a table. Any failed requests can be retried immediately.</p>

Understanding and analyzing DynamoDB response times

When analyzing latency, it's generally best to check the average. Occasional spikes in latency aren't a cause for concern. However, if average latency is high then an underlying issue could be responsible.

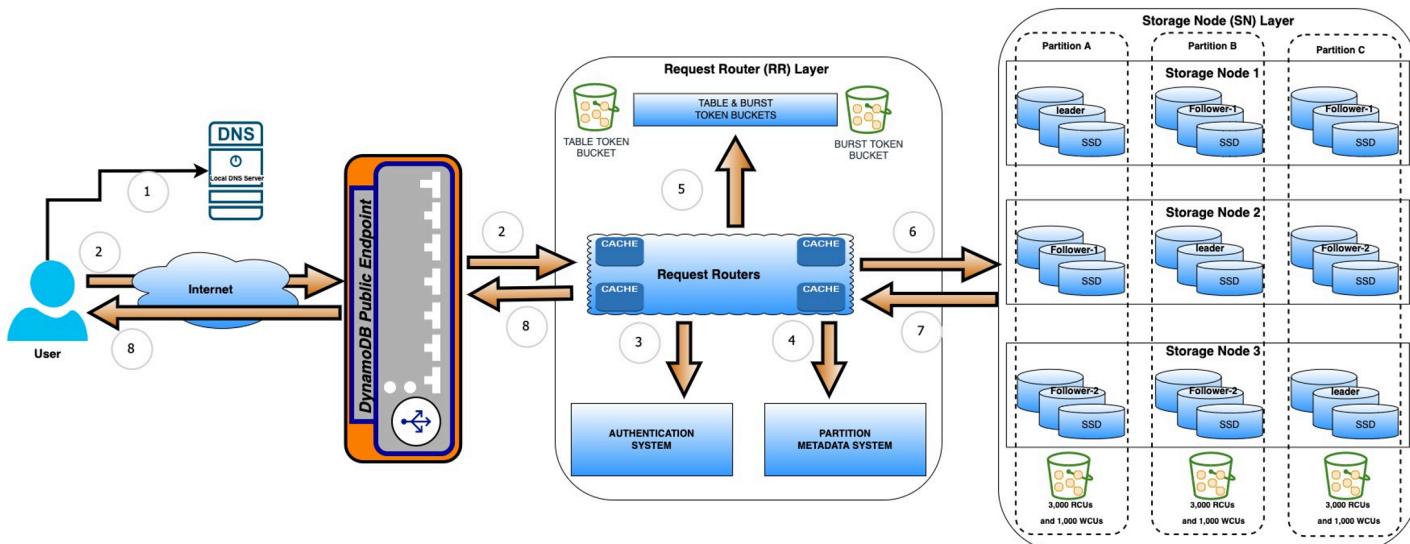
There are two categories of latency, API latency and service side latency. DynamoDB's API latency is measured from steps 1 through 11 in the process below. Service-side latency is measured from the moment an API reaches the Request Router (step 4) to the time the RR takes to send a result back to the application user (step 11). You can analyze service side latency with the Amazon CloudWatch metric `SuccessfulRequestLatency`.

When any application makes any DynamoDB API call to DynamoDB, such as issuing a `GetItem` operation to a DynamoDB table, the following steps occur.

1. The application resolves the DynamoDB public endpoint using local DNS server.
2. The application connects to the IP address resolved in step one, and makes an API call.
3. The DynamoDB public endpoint takes the request and forwards it to a component called Request Router (RR).
4. Once the API request reaches the Request Router (RR), RR does authentication and authorization of the API call. RR also does the throttling checks at this stage.
5. After completing all the checks Request Router (RR) creates the hash of the partition key value which it gets from the API request. Based upon hash value, Request Router (RR) finds the partition information (storage node) details.
6. The Storage Nodes represent the servers where customer table data is stored. A single partition (not to be confused with primary or partition key), consists of a set of 3 Storage

Nodes. Out of these three Storage Nodes, one node acts as a leader for that partition and the remaining two nodes act as follower.

7. If the API call is a write request or if it is strongly consistent read request, then Request Routers (RR) finds the leader node for that partition and forwards the API request to that specific node. In case of eventually consistent read request, Request Routers(RR) forwards the request to leader node or any of the follower nodes for that partition randomly.
8. In normal circumstances, the Request Router reaches the the Storage Node in first attempt. If this attempt fails, RR does multiple retries to reach the storage node. While connecting to storage node, RR always gives enough time to the existing attempt and then tries to connect to the different SN. This is an internal micro-service retry, and not a configurable SDK retry.
9. At this stage the API request reaches to Storage Node (SN) layer and SN starts processing it, reading or writing the data depending of the API call.
10. After successfully processing the API request, the Storage Node (SN) return the results or response code to the RR which originated the request.
11. Finally, Request Router forwards the results to the customer application.



Note

- For most atomic operations, such as `GetItem` and `PutItem`, you can expect an average latency in single-digit milliseconds. Latency for non-atomic operations such as `Query` and `Scan` depends on many factors, including the size of the result set and the complexity of the query conditions and filters.

- DynamoDB doesn't measure the amount of time an application takes to connect with the DynamoDB public endpoint, or the amount of time an application takes to download the results from the public endpoint.

Creating CloudWatch alarms to monitor DynamoDB

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period you specify, and performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods.

Note

You must specify all the required dimensions when creating your CloudWatch alarm, since CloudWatch will not aggregate metrics for a missing dimension. Creating a CloudWatch alarm with a missing dimension will not result in an error, when creating the alarm.

For a list of supported metrics and their required dimensions in DynamoDB, see [Viewing metrics and dimensions](#).

How can I be notified before I consume my entire read capacity?

1. Create an Amazon SNS topic, `arn:aws:sns:us-east-1:123456789012:capacity-alarm`.

For more information, see [Set up Amazon Simple Notification Service](#).

2. Create the alarm. In this example, we assume a provisioned capacity of five read capacity units.

```
aws cloudwatch put-metric-alarm \
  --alarm-name ReadCapacityUnitsLimitAlarm \
  --alarm-description "Alarm when read capacity reaches 80% of my provisioned
  read capacity" \
  --namespace AWS/DynamoDB \
```

```
--metric-name ConsumedReadCapacityUnits \
--dimensions Name=TableName,Value=myTable \
--statistic Sum \
--threshold 240 \
--comparison-operator GreaterThanOrEqualToThreshold \
--period 60 \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-east-1:123456789012:capacity-alarm
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name ReadCapacityUnitsLimitAlarm --state-reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name ReadCapacityUnitsLimitAlarm --state-reason "initializing" --state-value ALARM
```

Note

The alarm is activated whenever the consumed read capacity is at least 4 units per second (80% of provisioned read capacity of 5) for 1 minute (60 seconds). So the threshold is 240 read capacity units (4 units/sec * 60 seconds). Any time the read capacity is updated you should update the alarm calculations appropriately. You can avoid this process by creating alarms through the DynamoDB Console. In this way, the alarms are automatically updated for you.

How can I be notified if any requests exceed the provisioned throughput quotas of a table?

1. Create an Amazon SNS topic, arn:aws:sns:us-east-1:123456789012:requests-exceeding-throughput.

For more information, see [Set up Amazon Simple Notification Service](#).

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name RequestsExceedingThroughputAlarm \
--alarm-description "Alarm when my requests are exceeding provisioned throughput quotas of a table" \
```

```
--namespace AWS/DynamoDB \
--metric-name ThrottledRequests \
--dimensions Name=TableName,Value=myTable
Name=Operation,Value=aDynamoDBOperation \
--statistic Sum \
--threshold 0 \
--comparison-operator GreaterThanThreshold \
--period 300 \
--unit Count \
--evaluation-periods 1 \
--treat-missing-data notBreaching \
--alarm-actions arn:aws:sns:us-east-1:123456789012:requests-exceeding-
throughput
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --
state-reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --
state-reason "initializing" --state-value ALARM
```

How can I be notified if any system errors occurred?

1. Create an Amazon SNS topic, arn:aws:sns:us-east-1:123456789012:notify-on-system-errors.

For more information, see [Set up Amazon Simple Notification Service](#).

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name SystemErrorsAlarm \
--alarm-description "Alarm when system errors occur" \
--namespace AWS/DynamoDB \
--metric-name SystemErrors \
--dimensions Name=TableName,Value=myTable
Name=Operation,Value=aDynamoDBOperation \
--statistic Sum \
--threshold 0 \
--comparison-operator GreaterThanThreshold \
--period 60 \
```

```
--unit Count \
--evaluation-periods 1 \
--treat-missing-data breaching \
--alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
"initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
"initializing" --state-value ALARM
```

Logging DynamoDB operations by using AWS CloudTrail

DynamoDB is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in DynamoDB. CloudTrail captures all API calls for DynamoDB as events. The calls captured include calls from the DynamoDB console and code calls to the DynamoDB API operations, using both PartiQL and the classic API. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for DynamoDB. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to DynamoDB, the IP address from which the request was made, who made the request, when it was made, and additional details.

For robust monitoring and alerting, you can also integrate CloudTrail events with [Amazon CloudWatch Logs](#). To enhance your analysis of DynamoDB service activity and identify changes in activities for an AWS account, you can query AWS CloudTrail logs using [Amazon Athena](#). For example, you can use queries to identify trends and further isolate activity by attributes such as source IP address or user.

To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

Topics

- [DynamoDB information in CloudTrail](#)
- [Understanding DynamoDB log file entries](#)

DynamoDB information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in DynamoDB, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing events with CloudTrail event history](#).

For an ongoing record of events in your AWS account, including events for DynamoDB, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- [Overview for creating a trail](#)
- [CloudTrail supported services and integrations](#)
- [Configuring Amazon SNS notifications for CloudTrail](#)
- [Receiving CloudTrail log files from multiple regions](#) and [Receiving CloudTrail log files from multiple accounts](#)

Control plane events in CloudTrail

The following API actions are logged by default as events in CloudTrail files:

Amazon DynamoDB

- [CreateBackup](#)
- [CreateGlobalTable](#)
- [CreateTable](#)
- [DeleteBackup](#)
- [DeleteTable](#)
- [DescribeBackup](#)
- [DescribeContinuousBackups](#)
- [DescribeGlobalTable](#)
- [DescribeLimits](#)
- [DescribeTable](#)

- [DescribeTimeToLive](#)
- [ListBackups](#)
- [ListTables](#)
- [ListTagsOfResource](#)
- [ListGlobalTables](#)
- [RestoreTableFromBackup](#)
- [RestoreTableToPointInTime](#)
- [TagResource](#)
- [UntagResource](#)
- [UpdateGlobalTable](#)
- [UpdateTable](#)
- [UpdateTimeToLive](#)
- [DescribeReservedCapacity](#)
- [DescribeReservedCapacityOfferings](#)
- [PurchaseReservedCapacityOfferings](#)
- [DescribeScalableTargets](#)
- [RegisterScalableTarget](#)

DynamoDB Streams

- [DescribeStream](#)
- [ListStreams](#)

DynamoDB Accelerator (DAX)

- [CreateCluster](#)
- [CreateParameterGroup](#)
- [CreateSubnetGroup](#)
- [DecreaseReplicationFactor](#)
- [DeleteCluster](#)
- [DeleteParameterGroup](#)
- [DeleteSubnetGroup](#)

- [DescribeClusters](#)
- [DescribeDefaultParameters](#)
- [DescribeEvents](#)
- [DescribeParameterGroups](#)
- [DescribeParameters](#)
- [DescribeSubnetGroups](#)
- [IncreaseReplicationFactor](#)
- [ListTags](#)
- [RebootNode](#)
- [TagResource](#)
- [UntagResource](#)
- [UpdateCluster](#)
- [UpdateParameterGroup](#)
- [UpdateSubnetGroup](#)

DynamoDB data plane events in CloudTrail

To enable logging of the following API actions in CloudTrail files, you'll need to enable logging of data plane API activity in CloudTrail. See [Logging data events for trails](#) for more information.

Data plane events can be filtered by resource type, for granular control over which DynamoDB API calls you want to selectively log and pay for in CloudTrail. For example, by specifying AWS::DynamoDB::Stream as a resource type, you can log only calls to the DynamoDB streams APIs. For tables with streams enabled, the resource field in the data plane event contains both AWS::DynamoDB::Stream and AWS::DynamoDB::Table. If you specify AWS::DynamoDB::Table as a resource type, it will log both DynamoDB table and DynamoDB streams events by default. You can add an additional [filter](#) to exclude the streams events, if you don't want the streams events to be logged. For more information, see [DataResource](#) in the CloudTrail [AWS CloudTrail API reference](#).

Amazon DynamoDB

- [BatchExecuteStatement](#)
- [BatchGetItem](#)

- [BatchWriteItem](#)
- [DeleteItem](#)
- [ExecuteStatement](#)
- [ExecuteTransaction](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [TransactGetItems](#)
- [TransactWriteItems](#)
- [UpdateItem](#)

 **Note**

DynamoDB Time to Live data plane actions are not logged by CloudTrail

DynamoDB Streams

- [GetRecords](#)
- [GetShardIterator](#)

Understanding DynamoDB log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

Note

Non key attribute values will be redacted in the CloudTrail logs of actions using the PartiQL API, and will not appear in logs of actions using the classic API.

For more information, see the [CloudTrail userIdentity element](#).

The following examples demonstrate CloudTrail logs of these event types:

Amazon DynamoDB

- [UpdateTable](#)
- [DeleteTable](#)
- [CreateCluster](#)
- [PutItem \(Successful\)](#)
- [UpdateItem \(Unsuccessful\)](#)
- [TransactWriteItems \(Successful\)](#)
- [TransactWriteItems \(With TransactionCanceledException\)](#)
- [ExecuteStatement](#)
- [BatchExecuteStatement](#)

DynamoDB Streams

- [GetRecords](#)

UpdateTable

```
{  
    "Records": [  
        {  
            "eventVersion": "1.03",  
            "userIdentity": {  
                "type": "AssumedRole",  
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",  
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",  
                "accountId": "111122223333",  
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
                "sessionContext": {  
                    "sessionIssuer": {  
                        "type": "AWS",  
                        "principalId": "ASIAQK1TUVV4OOGJLWQ",  
                        "arn": "arn:aws:sts::111122223333:aws-iam",  
                        "accountId": "111122223333",  
                        "userName": "root",  
                        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
                        "sessionName": "root",  
                        "sessionDuration": 3600  
                    },  
                    "sessionToken": "AQAB...  
                }  
            }  
        }  
    ]  
}
```

```
"sessionContext": {
    "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2015-05-28T18:06:01Z"
    },
    "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::444455556666:role/admin-role",
        "accountId": "444455556666",
        "userName": "bob"
    }
},
},
"eventTime": "2015-05-04T02:14:52Z",
"eventSource": "dynamodb.amazonaws.com",
"eventName": "UpdateTable",
"awsRegion": "us-west-2",
"sourceIPAddress": "192.0.2.0",
"userAgent": "console.aws.amazon.com",
"requestParameters": {
    "provisionedThroughput": {
        "writeCapacityUnits": 25,
        "readCapacityUnits": 25
    }
},
},
"responseElements": {
    "tableDescription": {
        "tableName": "Music",
        "attributeDefinitions": [
            {
                "attributeType": "S",
                "attributeName": "Artist"
            },
            {
                "attributeType": "S",
                "attributeName": "SongTitle"
            }
        ],
        "itemCount": 0,
        "provisionedThroughput": {
            "writeCapacityUnits": 10,
            "numberOfDecreasesToday": 0,
            "readCapacityUnits": 10,
        }
    }
}
```

```
        "lastIncreaseDateTime": "May 3, 2015 11:34:14 PM"
    },
    "creationDateTime": "May 3, 2015 11:34:14 PM",
    "keySchema": [
        {
            "attributeName": "Artist",
            "keyType": "HASH"
        },
        {
            "attributeName": "SongTitle",
            "keyType": "RANGE"
        }
    ],
    "tableStatus": "UPDATING",
    "tableSizeBytes": 0
}
},
"requestID": "AALNP0J2L244N5015PKISJ1KUFVV4KQNS05AEMVJF66Q9ASUAAJG",
"eventID": "eb834e01-f168-435f-92c0-c36278378b6e",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
}
]
}
```

DeleteTable

```
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },

```

```
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::444455556666:role/admin-role",
            "accountId": "444455556666",
            "userName": "bob"
        }
    },
},
"eventTime": "2015-05-04T13:38:20Z",
"eventSource": "dynamodb.amazonaws.com",
"eventName": "DeleteTable",
"awsRegion": "us-west-2",
"sourceIPAddress": "192.0.2.0",
"userAgent": "console.aws.amazon.com",
"requestParameters": {
    "tableName": "Music"
},
"responseElements": {
    "tableDescription": {
        "tableName": "Music",
        "itemCount": 0,
        "provisionedThroughput": {
            "writeCapacityUnits": 25,
            "numberOfDecreasesToday": 0,
            "readCapacityUnits": 25
        },
        "tableStatus": "DELETING",
        "tableSizeBytes": 0
    }
},
"requestID": "4KBNVRGD25RG1KE09UT4V3FQDJVV4KQNS05AEMVJF66Q9ASUAAJG",
"eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
}
]
```

CreateCluster

{

```
"Records": [
    {
        "eventVersion": "1.05",
        "userIdentity": {
            "type": "IAMUser",
            "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
            "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
            "accountId": "111122223333",
            "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
            "userName": "bob"
        },
        "eventTime": "2019-12-17T23:17:34Z",
        "eventSource": "dax.amazonaws.com",
        "eventName": "CreateCluster",
        "awsRegion": "us-west-2",
        "sourceIPAddress": "192.0.2.0",
        "userAgent": "aws-cli/1.16.304 Python/3.6.9
Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.13.40",
        "requestParameters": {
            "sSESpecification": {
                "enabled": true
            },
            "clusterName": "daxcluster",
            "nodeType": "dax.r4.large",
            "replicationFactor": 3,
            "iamRoleArn": "arn:aws:iam::111122223333:role/
DAXServiceRoleForDynamoDBAccess"
        },
        "responseElements": {
            "cluster": {
                "securityGroups": [
                    {
                        "securityGroupIdentifier": "sg-1af6e36e",
                        "status": "active"
                    }
                ],
                "parameterGroup": {
                    "nodeIdsToReboot": [],
                    "parameterGroupName": "default.dax1.0",
                    "parameterApplyStatus": "in-sync"
                },
                "clusterDiscoveryEndpoint": {
                    "port": 8111
                },
                "tags": [
                    {
                        "key": "aws:cloudTrail:resourceArn",
                        "value": "arn:aws:dynamodb:us-west-2:111122223333:table/test-table"
                    }
                ]
            }
        }
    }
]
```

```
        "clusterArn": "arn:aws:dax:us-west-2:111122223333:cache/daxcluster",
        "status": "creating",
        "subnetGroup": "default",
        "sSDescription": {
            "status": "ENABLED",
            "kMSMasterKeyArn": "arn:aws:kms:us-west-2:111122223333:key/764898e4-adb1-46d6-a762-e2f4225b4fc4"
        },
        "iamRoleArn": "arn:aws:iam::111122223333:role/DAXServiceRoleForDynamoDBAccess",
        "clusterName": "daxcluster",
        "activeNodes": 0,
        "totalNodes": 3,
        "preferredMaintenanceWindow": "thu:13:00-thu:14:00",
        "nodeType": "dax.r4.large"
    }
},
{
    "requestID": "585adc5f-ad05-4e27-8804-70ba1315f8fd",
    "eventID": "29158945-28da-4e32-88e1-56d1b90c1a0c",
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
]
```

PutItem (Successful)

```
{
    "Records": [
        {
            "eventVersion": "1.06",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                }
            }
        }
    ]
}
```

```
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::444455556666:role/admin-role",
            "accountId": "444455556666",
            "userName": "bob"
        }
    }
},
"eventTime": "2019-01-19T15:41:54Z",
"eventSource": "dynamodb.amazonaws.com",
"eventName": "PutItem",
"awsRegion": "us-west-2",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
"requestParameters": {
    "tableName": "Music",
    "key": {
        "Artist": "No One You Know",
        "SongTitle": "Scared of My Shadow"
    },
    "item": [
        "Artist",
        "SongTitle",
        "AlbumTitle"
    ],
    "returnConsumedCapacity": "TOTAL"
},
"responseElements": null,
"requestID": "4KBNVRGD25RG1KE09UT4V3FQDJVV4KQNS05AEMVJF66Q9ASUAAJG",
"eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
"readOnly": false,
"resources": [
{
    "accountId": "111122223333",
    "type": "AWS::DynamoDB::Table",
    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
}
],
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"managementEvent": false,
"recipientAccountId": "111122223333",
```

```
        "eventCategory": "Data"
    }
]
}
```

UpdateItem (Unsuccessful)

```
{
  "Records": [
    {
      "eventVersion": "1.07",
      "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
          "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::444455556666:role/admin-role",
            "accountId": "444455556666",
            "userName": "bob"
          },
          "attributes": {
            "creationDate": "2020-09-03T22:14:13Z",
            "mfaAuthenticated": "false"
          }
        }
      },
      "eventTime": "2020-09-03T22:27:15Z",
      "eventSource": "dynamodb.amazonaws.com",
      "eventName": "UpdateItem",
      "awsRegion": "us-west-2",
      "sourceIPAddress": "192.0.2.0",
      "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
      "errorCode": "ConditionalCheckFailedException",
      "errorMessage": "The conditional request failed",
      "requestParameters": {
        "tableName": "Music",
        "key": {
```

```
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    "updateExpression": "SET #Y = :y, #AT = :t",
    "expressionAttributeNames": {
        "#Y": "Year",
        "#AT": "AlbumTitle"
    },
    "conditionExpression": "attribute_not_exists(#Y)",
    "returnConsumedCapacity": "TOTAL"
},
"responseElements": null,
"requestID": "4KBNVRGD25RG1KE09UT4V3FQDJVV4KQNS05AEMVJF66Q9ASUAAJG",
"eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
"readOnly": false,
"resources": [
    {
        "accountId": "111122223333",
        "type": "AWS::DynamoDB::Table",
        "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
    }
],
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"managementEvent": false,
"recipientAccountId": "111122223333",
"eventCategory": "Data"
}
]
```

TransactWriteItems (Successful)

```
{  
  "Records": [  
    {  
      "eventVersion": "1.07",  
      "userIdentity": {  
        "type": "AssumedRole",  
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",  
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",  
        "accountId": "111122223333",  
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
        "sessionName": "aws-sdk-nodejs-1592441183453" } ] }  
}
```

```
"sessionContext": {
    "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::444455556666:role/admin-role",
        "accountId": "444455556666",
        "userName": "bob"
    },
    "attributes": {
        "creationDate": "2020-09-03T22:14:13Z",
        "mfaAuthenticated": "false"
    }
},
{
    "eventTime": "2020-09-03T21:48:12Z",
    "eventSource": "dynamodb.amazonaws.com",
    "eventName": "TransactWriteItems",
    "awsRegion": "us-west-1",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
    "requestParameters": {
        "requestItems": [
            {
                "operation": "Put",
                "tableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Call Me Today"
                },
                "items": [
                    "Artist",
                    "SongTitle",
                    "AlbumTitle"
                ],
                "conditionExpression": "#AT = :A",
                "expressionAttributeNames": {
                    "#AT": "AlbumTitle"
                },
                "returnValuesOnConditionCheckFailure": "ALL_OLD"
            },
            {
                "operation": "Update",
                "tableName": "Music",
```

```
"key": {
    "Artist": "No One You Know",
    "SongTitle": "Call Me Tomorrow"
},
"updateExpression": "SET #AT = :newval",
"ConditionExpression": "attribute_not_exists(Rating)",
"ExpressionAttributeNames": {
    "#AT": "AlbumTitle"
},
"returnValuesOnConditionCheckFailure": "ALL_OLD"
},
{
    "operation": "Delete",
    "TableName": "Music",
    "key": {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Yesterday"
    },
    "conditionExpression": "#P between :lo and :hi",
    "expressionAttributeNames": {
        "#P": "Price"
    },
    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
},
{
    "operation": "ConditionCheck",
    "TableName": "Music",
    "Key": {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Now"
    },
    "ConditionExpression": "#P between :lo and :hi",
    "ExpressionAttributeNames": {
        "#P": "Price"
    },
    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
}
],
"returnConsumedCapacity": "TOTAL",
"returnItemCollectionMetrics": "SIZE"
},
"responseElements": null,
"requestID": "45EN320M6TQSMV2MI6504L5TNFVV4KQNS05AEMVJF66Q9ASUAAJG",
"eventID": "4f1cc78b-5c94-4174-a6ad-3ee78605381c",
```

```
        "readOnly": false,
        "resources": [
            {
                "accountId": "111122223333",
                "type": "AWS::DynamoDB::Table",
                "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
            }
        ],
        "eventType": "AwsApiCall",
        "apiVersion": "2012-08-10",
        "managementEvent": false,
        "recipientAccountId": "111122223333",
        "eventCategory": "Data"
    }
]
}
```

TransactWriteItems (With TransactionCanceledException)

```
{
    "Records": [
        {
            "eventVersion": "1.06",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "awsRegion": "us-east-1",
            "dynamodb": {
                "RequestResponses": [
                    {
                        "PutItem": {
                            "Item": {
                                "SongName": "The Sound of Silence",
                                "Artist": "Simon & Garfunkel",
                                "Year": 1965
                            }
                        }
                    }
                ]
            }
        }
    ]
}
```

```
"eventTime": "2019-02-01T00:42:34Z",
"eventSource": "dynamodb.amazonaws.com",
"eventName": "TransactWriteItems",
"awsRegion": "us-west-2",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.16.93 Python/3.4.7
Linux/4.9.119-0.1.ac.277.71.329.metal1.x86_64 botocore/1.12.83",
"errorCode": "TransactionCanceledException",
"errorMessage": "Transaction cancelled, please refer cancellation reasons
for specific reasons [ConditionalCheckFailed, None]",
"requestParameters": {
    "requestItems": [
        {
            "operation": "Put",
            "tableName": "Music",
            "key": {
                "Artist": "No One You Know",
                "SongTitle": "Call Me Today"
            },
            "items": [
                "Artist",
                "SongTitle",
                "AlbumTitle"
            ],
            "conditionExpression": "#AT = :A",
            "expressionAttributeNames": {
                "#AT": "AlbumTitle"
            },
            "returnValuesOnConditionCheckFailure": "ALL_OLD"
        },
        {
            "operation": "Update",
            "tableName": "Music",
            "key": {
                "Artist": "No One You Know",
                "SongTitle": "Call Me Tomorrow"
            },
            "updateExpression": "SET #AT = :newval",
            "ConditionExpression": "attribute_not_exists(Rating)",
            "ExpressionAttributeNames": {
                "#AT": "AlbumTitle"
            },
            "returnValuesOnConditionCheckFailure": "ALL_OLD"
        },
    ]
}
```

```
{  
    "operation": "Delete",  
    "TableName": "Music",  
    "key": {  
        "Artist": "No One You Know",  
        "SongTitle": "Call Me Yesterday"  
    },  
    "conditionExpression": "#P between :lo and :hi",  
    "expressionAttributeNames": {  
        "#P": "Price"  
    },  
    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"  
},  
{  
    "operation": "ConditionCheck",  
    "TableName": "Music",  
    "Key": {  
        "Artist": "No One You Know",  
        "SongTitle": "Call Me Now"  
    },  
    "ConditionExpression": "#P between :lo and :hi",  
    "ExpressionAttributeNames": {  
        "#P": "Price"  
    },  
    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"  
}  
],  
"returnConsumedCapacity": "TOTAL",  
"returnItemCollectionMetrics": "SIZE"  
},  
"responseElements": null,  
"requestID": "A0GTQEKLBB9VD8E05REA5A3E1VVV4KQNS05AEMVJF66Q9ASUAAJG",  
"eventID": "43e437b5-908a-46af-84e6-e27fffb9c5cd",  
"readOnly": false,  
"resources": [  
    {  
        "accountId": "111122223333",  
        "type": "AWS::DynamoDB::Table",  
        "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"  
    }  
],  
"eventType": "AwsApiCall",  
"apiVersion": "2012-08-10",  
"managementEvent": false,
```

```
        "recipientAccountId": "111122223333",
        "eventCategory": "Data"
    }
]
}
```

ExecuteStatement

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-03-03T23:06:45Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "ExecuteStatement",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.7 Python/3.6.13
Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 botocore/1.20.7",
            "requestParameters": {
                "statement": "SELECT * FROM Music WHERE Artist = 'No One You Know' AND
SongTitle = 'Call Me Today' AND nonKeyAttr = ***(Redacted)"
            },
        }
    ],
}
```

```
"responseElements": null,  
"requestID": "V7G2KCSFLP830RB7MMFG6RIAD3VV4KQNS05AEMVJF66Q9ASUAAJG",  
"eventID": "0b5c4779-e169-4227-a1de-6ed01dd18ac7",  
"readOnly": false,  
"resources": [  
    {  
        "accountId": "111122223333",  
        "type": "AWS::DynamoDB::Table",  
        "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"  
    }  
],  
"eventType": "AwsApiCall",  
"apiVersion": "2012-08-10",  
"managementEvent": false,  
"recipientAccountId": "111122223333",  
"eventCategory": "Data"  
}  
]  
}
```

BatchExecuteStatement

```
{  
    "Records": [  
        {  
            "eventVersion": "1.08",  
            "userIdentity": {  
                "type": "AssumedRole",  
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",  
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",  
                "accountId": "111122223333",  
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
                "sessionContext": {  
                    "sessionIssuer": {  
                        "type": "Role",  
                        "principalId": "AKIAI44QH8DHBEEXAMPLE",  
                        "arn": "arn:aws:iam::444455556666:role/admin-role",  
                        "accountId": "444455556666",  
                        "userName": "bob"  
                    },  
                    "attributes": {  
                        "creationDate": "2020-09-03T22:14:13Z",  
                        "mfaAuthenticated": "false"  
                    }  
                }  
            }  
        }  
    ]  
}
```

```
        }
    },
},
"eventTime": "2021-03-03T23:24:48Z",
"eventSource": "dynamodb.amazonaws.com",
"eventName": "BatchExecuteStatement",
"awsRegion": "us-west-2",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.19.7 Python/3.6.13
Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 botocore/1.20.7",
"requestParameters": {
    "requestItems": [
        {
            "statement": "UPDATE Music SET Album = *** (Redacted) WHERE
Artist = 'No One You Know' AND SongTitle = 'Call Me Today'"
        },
        {
            "statement": "INSERT INTO Music VALUE {'Artist' :
*** (Redacted), 'SongTitle' : *** (Redacted), 'Album' : *** (Redacted)}"
        }
    ]
},
"responseElements": null,
"requestID": "23PE7ED291UD65P9SMS6TISNVBVV4KQNS05AEMVJF66Q9ASUAAJG",
"eventID": "f863f966-b741-4c36-b15e-f867e829035a",
"readOnly": false,
"resources": [
    {
        "accountId": "111122223333",
        "type": "AWS::DynamoDB::Table",
        "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
    }
],
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"managementEvent": false,
"recipientAccountId": "111122223333",
"eventCategory": "Data"
}
]
}
```

GetRecords

```
{  
    "Records": [  
        {  
            "eventVersion": "1.08",  
            "userIdentity": {  
                "type": "AssumedRole",  
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",  
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",  
                "accountId": "111122223333",  
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
                "sessionContext": {  
                    "sessionIssuer": {  
                        "type": "Role",  
                        "principalId": "AKIAI44QH8DHBEXAMPLE",  
                        "arn": "arn:aws:iam::444455556666:role/admin-role",  
                        "accountId": "444455556666",  
                        "userName": "bob"  
                    },  
                    "attributes": {  
                        "creationDate": "2020-09-03T22:14:13Z",  
                        "mfaAuthenticated": "false"  
                    }  
                }  
            },  
            "eventTime": "2021-04-15T04:15:02Z",  
            "eventSource": "dynamodb.amazonaws.com",  
            "eventName": "GetRecords",  
            "awsRegion": "us-west-2",  
            "sourceIPAddress": "192.0.2.0",  
            "userAgent": "aws-cli/1.19.50 Python/3.6.13  
Linux/4.9.230-0.1.ac.224.84.332.metal1.x86_64 botocore/1.20.50",  
            "requestParameters": {  
                "shardIterator": "arn:aws:dynamodb:us-west-2:123456789012:table/  
Music/stream/2021-04-15T04:02:47.428|1|AAAAAAAAAAH7HF3xwDQHBrvk2UBZ1PKh8bX3F  
+JeH0rFwHCE7dz4VGv1ZoJ5bMxQwkmerA3zwCTL+zSseGLdSXNJP14EwrjLNvDNoZeRSJ/  
n6xc3I4NY0ptR4zR8d7VrjMAD6h5nR12NtxGIgJ/  
dVsUpluWsHyCW3PPbKsMlJSruVRWoitRhSd3S6s1EWEPB0bDC7+  
+ISH5mXrCHOnveyzQK1qNshTSPZ5jWwqRj2VNSXCMTGXv9P01/  
U0bpOUI2cuRTchqUpPSe3ur2sQrRj3K1bmIyCz7P  
+H3CYlugafi8fQ5kipDSkESkIWS605ejzibWKg/3izms1eVIm/  
zLFdEeihCYJ7G8fpHUSLX5JAk3ab68aUXGSFEZLONntgNIhQkcMo00/  
mJlaIgkEdBUyqvZ01vtKUBH5YonIrZqSUhv8Coc+mh24v0g1YI+SPIXlr
```

```
+Ln154BG6AjrmScjHACVXoPDxPsXSJXC4c9HjoC3YSskCPV7uWi0f65/
n7JAT3cskcX2ISaLHwYzJPaMBSftx0geRLm3BnisL32nT8uTj2gF/
PUrEjdyoqTX7EerQpcaekXm0gay5Kh8n4T2uPdM83f356vRpar/
DDp8pLFD0ddb6Yvz7zU2zGdAvTod3IScC1GpTqcjRxaMh1BVZy1TnI9Cs
+7fXMdUF6xYScjR2725icFBNLojSFVDmsfHabXaCEpmeuXZsLbp5CjcPAHa66R8mQ5tSoFjrz0EzeB4uconEXAMPLE=="
},
"responseElements": null,
"requestID": "1M0U1Q80P4LDPT7A7N1A758N2VVV4KQNS05AEMVJF66Q9EXAMPLE",
"eventID": "09a634f2-da7d-4c9e-a259-54aceexample",
"readOnly": true,
"resources": [
{
    "accountId": "111122223333",
    "type": "AWS::DynamoDB::Table",
    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
}
],
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"managementEvent": false,
"recipientAccountId": "111122223333",
"eventCategory": "Data"
}
]
}
```

Logging and monitoring in DynamoDB Accelerator

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon DynamoDB Accelerator (DAX) and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs.

For more information about logging and monitoring in DAX, see [Monitoring DAX](#).

Analyzing data access using CloudWatch contributor insights for DynamoDB

Amazon CloudWatch Contributor Insights for Amazon DynamoDB is a diagnostic tool for identifying the most frequently accessed and throttled keys in your table or index at a glance. This tool uses [CloudWatch contributor insights](#).

By enabling CloudWatch Contributor Insights for DynamoDB on a table or global secondary index, you can view the most accessed and throttled items in those resources.

Note

CloudWatch charges apply for Contributor Insights for DynamoDB. For more information about pricing, see [Amazon CloudWatch pricing](#).

Topics

- [CloudWatch contributor insights for DynamoDB: How it works](#)
- [Getting started with CloudWatch contributor insights for DynamoDB](#)
- [Using IAM with CloudWatch contributor insights for DynamoDB](#)

CloudWatch contributor insights for DynamoDB: How it works

Amazon DynamoDB integrates with [Amazon CloudWatch Contributor Insights](#) to provide information about the most accessed and throttled items in a table or global secondary index. DynamoDB delivers this information to you via CloudWatch Contributor Insights [rules](#), [reports](#), and graphs of report data.

For more information about CloudWatch Contributor Insights, see [Using Contributor Insights to Analyze High-Cardinality Data](#) in the *Amazon CloudWatch User Guide*.

The following sections describe the core concepts and behavior of CloudWatch Contributor Insights for DynamoDB.

Topics

- [CloudWatch contributor insights for DynamoDB rules](#)
- [Understanding CloudWatch contributor insights for DynamoDB graphs](#)

- [Interactions with other DynamoDB features](#)
- [CloudWatch contributor insights for DynamoDB billing](#)

CloudWatch contributor insights for DynamoDB rules

When you enable CloudWatch Contributor Insights for DynamoDB on a table or global secondary index, DynamoDB creates the following [rules](#) on your behalf:

- **Most accessed items (partition key)** — Identifies the partition keys of the most accessed items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-PKC-[resource_name]-[creationtimestamp]

- **Most throttled keys (partition key)** — Identifies the partition keys of the most throttled items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-PKT-[resource_name]-[creationtimestamp]

Note

When you enable Contributor Insights on your DynamoDB table, you're still subject to Contributor Insights rules limits. For more information, see [CloudWatch service quotas](#).

If your table or global secondary index has a sort key, DynamoDB also creates the following rules specific to sort keys:

- **Most accessed keys (partition and sort keys)** — Identifies the partition and sort keys of the most accessed items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-SKC-[resource_name]-[creationtimestamp]

- **Most throttled keys (partition and sort keys)** — Identifies the partition and sort keys of the most throttled items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-SKT-[resource_name]-[creationtimestamp]

Note

- You can't use the CloudWatch console or APIs to directly modify or delete the rules created by CloudWatch Contributor Insights for DynamoDB. Disabling CloudWatch Contributor Insights for DynamoDB on a table or global secondary index automatically deletes the rules created for that table or global secondary index.
- When you use the [GetInsightRuleReport](#) operation with CloudWatch Contributor Insights rules that are created by DynamoDB, only MaxContributorValue and Maximum return useful statistics. The other statistics in this list don't return meaningful values.
- CloudWatch Contributor Insights for DynamoDB has a limit of 25 contributors. Requesting more than 25 contributors will return an error.

You can create CloudWatch Alarms using the CloudWatch Contributor Insights for DynamoDB [rules](#). This allows you to be notified when any item exceed or meets a specific threshold for ConsumedThroughputUnits or ThrottleCount. For more information, see [Setting an Alarm on Contributor Insights Metric Data](#).

Understanding CloudWatch contributor insights for DynamoDB graphs

CloudWatch Contributor Insights for DynamoDB displays two types of graphs on both the DynamoDB and CloudWatch consoles: *Most Accessed Items* and *Most Throttled Items*.

Most accessed items

Use this graph to identify the most accessed items in the table or global secondary index. The graph displays ConsumedThroughputUnits on the y-axis and time on the x-axis. Each of the top N keys is displayed in its own color, with a legend displayed below the x-axis.

DynamoDB measures key access frequency by using ConsumedThroughputUnits, which measures combined read and write traffic. ConsumedThroughputUnits is defined as the following:

- Provisioned — $(3 \times \text{consumed write capacity units}) + \text{consumed read capacity units}$
- On-demand — $(3 \times \text{write request units}) + \text{read request units}$

On the DynamoDB console, each data point in the graph represents the maximum of ConsumedThroughputUnits over a 1-minute period. For example, a graph value of 180,000

`ConsumedThroughputUnits` indicates that the item was accessed continuously at the per-item maximum throughput of 1,000 write request units or 3,000 read request units for a 60-second span within that 1-minute period (3,000 x 60 seconds). In other words, the graphed values represent the highest-traffic minute within each 1-minute period. You can change the time granularity of the `ConsumedThroughputUnits` metric (for example, to view 5-minute metrics instead of 1-minute) on the CloudWatch console.

If you see several closely clustered lines without any obvious outliers, it indicates that your workload is relatively balanced across items over the given time window. If you see isolated points in the graph instead of connected lines, it indicates an item that was frequently accessed only for a brief period.

If your table or global secondary index has a sort key, DynamoDB creates two graphs: one for the most accessed partition keys and one for the most accessed partition + sort key pairs. You can see traffic at the partition key level in the partition key-only graph. You can see traffic at the item level in the partition + sort key graphs.

Most throttled items

Use this graph to identify the most throttled items in the table or global secondary index. The graph displays `ThrottleCount` on the y-axis and time on the x-axis. Each of the top N keys is displayed in its own color, with a legend displayed below the x-axis.

DynamoDB measures throttle frequency using `ThrottleCount`, which is the count of `ProvisionedThroughputExceededException`, `ThrottlingException`, and `RequestLimitExceeded` errors.

Write throttling caused by insufficient write capacity for a global secondary index is not measured. You can use the *Most Accessed Items* graph of the global secondary index to identify imbalanced access patterns that may cause write throttling. For more information, see [Provisioned Throughput Considerations for Global Secondary Indexes](#).

On the DynamoDB console, each data point in the graph represents the count of throttle events over a 1-minute period.

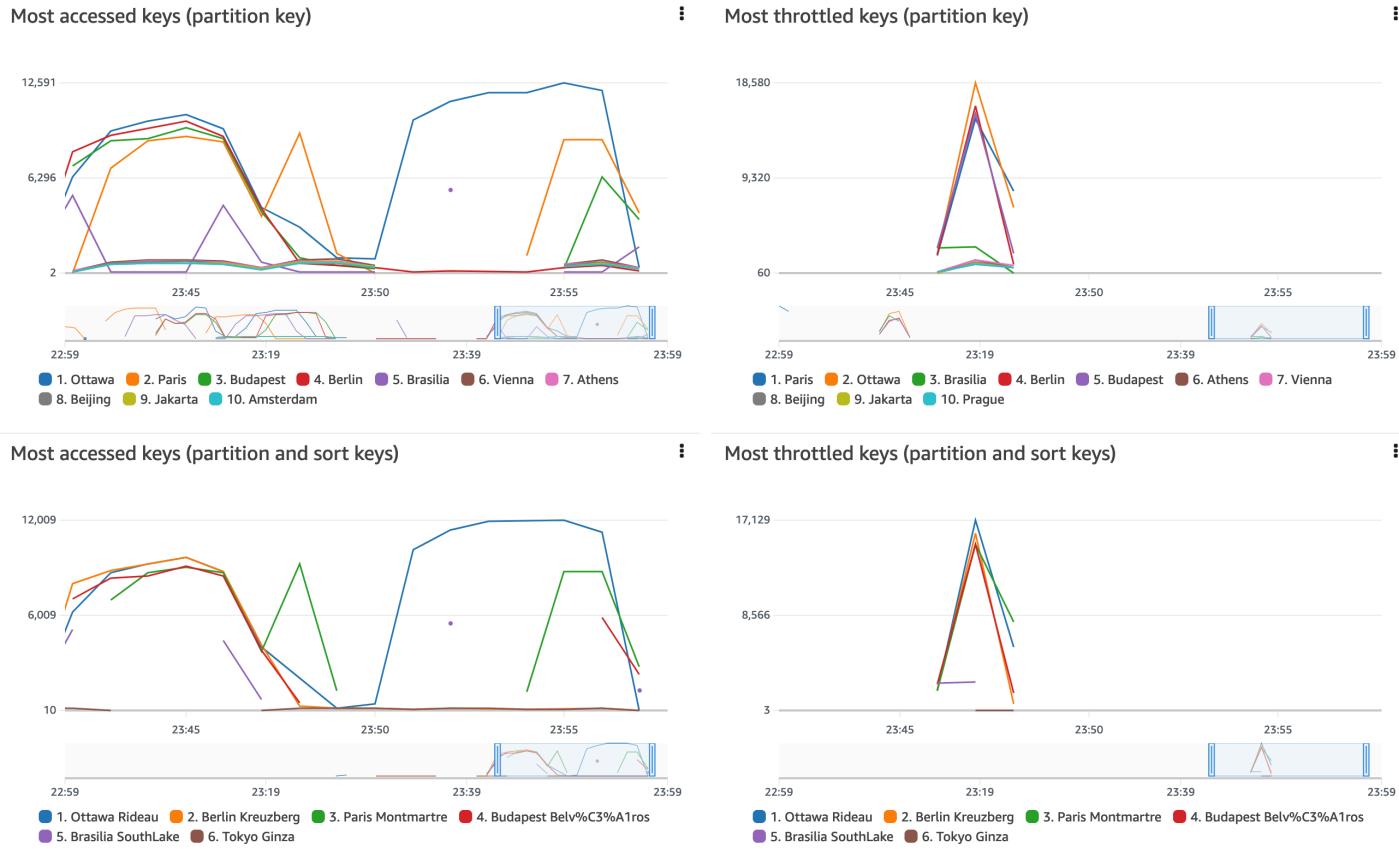
If you see no data in this graph, it indicates that your requests are not being throttled. If you see isolated points in the graph instead of connected lines, it indicates that an item was frequently throttled for a brief period.

If your table or global secondary index has a sort key, DynamoDB creates two graphs: one for most throttled partition keys and one for most throttled partition + sort key pairs. You can see throttle

count at the partition key level in the partition key-only graph, and throttle count at the item-level in the partition + sort key graphs.

Report examples

The following are examples of the reports generated for a table with both a partition key and sort key.



Interactions with other DynamoDB features

The following sections describe how CloudWatch Contributor Insights for DynamoDB behaves and interacts with several other features in DynamoDB.

Global tables

CloudWatch Contributor Insights for DynamoDB monitors global table replicas as distinct tables. The Contributor Insights graphs for a replica in one AWS Region might not show the same patterns as another Region. This is because write data is replicated across all replicas in a global table, but each replica can serve Region-bound read traffic.

DynamoDB Accelerator (DAX)

CloudWatch Contributor Insights for DynamoDB doesn't show DAX cache responses. It only shows responses to accessing a table or a global secondary index.

Note

DynamoDB CCI does not support PartiQL requests.

Encryption at rest

CloudWatch Contributor Insights for DynamoDB doesn't affect how encryption works in DynamoDB. The primary key data that is published in CloudWatch is encrypted with the AWS owned key. However, DynamoDB also supports the AWS managed key and a customer managed key.

CloudWatch Contributor Insights for DynamoDB graphs display the partition key and sort key (if applicable) of frequently accessed items and frequently throttled items in plaintext. If you require the use of AWS Key Management Service (KMS) to encrypt this table's partition key and sort key data with an AWS managed key or customer managed key, you should not enable CloudWatch Contributor Insights for DynamoDB for this table.

If you require your primary key data to be encrypted with the AWS managed key or a customer managed key, you should not enable CloudWatch Contributor Insights for DynamoDB for that table.

Fine-grained access control

CloudWatch Contributor Insights for DynamoDB doesn't function differently for tables with fine-grained access control (FGAC). In other words, any user who has the appropriate CloudWatch permissions can view FGAC-protected primary keys in CloudWatch Contributor Insights graphs.

If the table's primary key contains FGAC-protected data that you don't want published to CloudWatch, you should not enable CloudWatch Contributor Insights for DynamoDB for that table.

Access control

You control access to CloudWatch Contributor Insights for DynamoDB using AWS Identity and Access Management (IAM) by limiting DynamoDB control plane permissions and CloudWatch data

plane permissions. For more information see, [Using IAM with CloudWatch Contributor Insights for DynamoDB](#).

CloudWatch contributor insights for DynamoDB billing

Charges for CloudWatch Contributor Insights for DynamoDB appear in the [CloudWatch](#) section of your monthly bill. These charges are calculated based on the number of DynamoDB events that are processed. For tables and global secondary indexes with CloudWatch Contributor Insights for DynamoDB enabled, each item that is written or read via a [data plane](#) operation represents one event.

If a table or global secondary index includes a sort key, each item that is read or written represents two events. This is because DynamoDB is identifying top contributors from separate time series: one for partitions keys only, and one for partition and sort key pairs.

For example, assume that your application performs the following DynamoDB operations: a `GetItem`, a `PutItem`, and a `BatchWriteItem` that puts five items

- If your table or global secondary index has only a partition key, it results in 7 events (1 for the `GetItem`, 1 for the `PutItem`, and 5 for the `BatchWriteItem`).
- If your table or global secondary index has a partition key and sort key, it results in 14 events (2 for the `GetItem`, 2 for the `PutItem`, and 10 for the `BatchWriteItem`).
- A `Query` operation always results in 1 event, regardless of the number of items returned.

Unlike other DynamoDB features, CloudWatch Contributor Insights for DynamoDB billing *does not* vary based on the following:

- The [capacity mode](#) (provisioned vs. on-demand)
- Whether you perform read or write requests
- The size (KB) of the items read or written

Getting started with CloudWatch contributor insights for DynamoDB

This section describes how to use Amazon CloudWatch Contributor Insights with the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

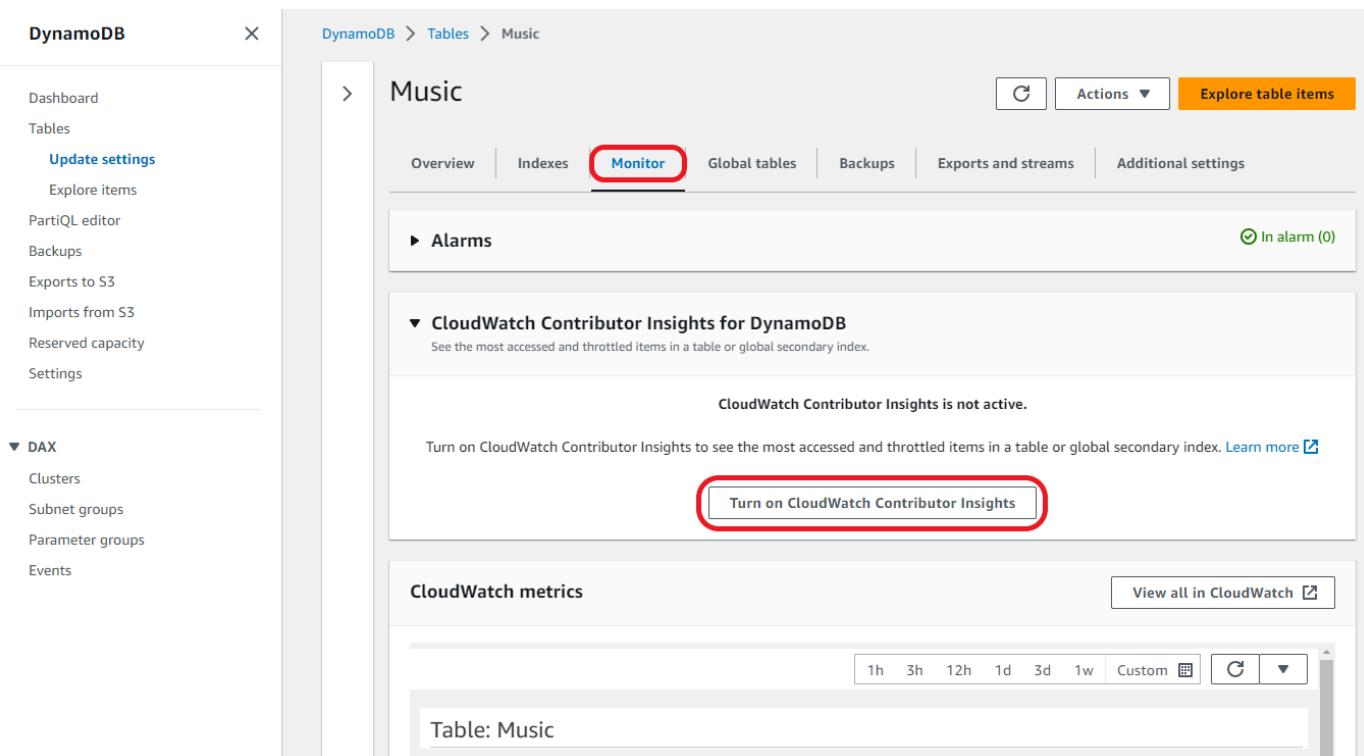
In the following examples, you use the DynamoDB table that is defined in the [Getting started with DynamoDB](#) tutorial.

Topics

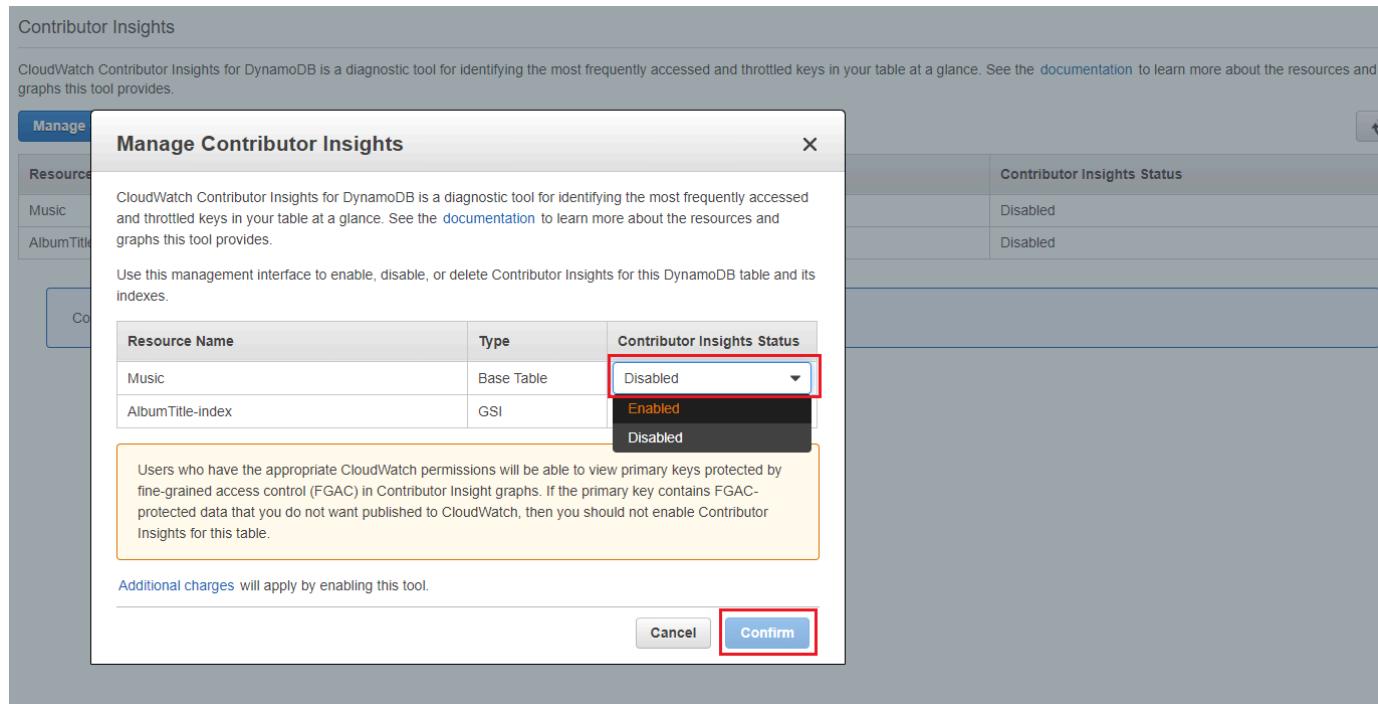
- [Using contributor insights \(console\)](#)
- [Using contributor insights \(AWS CLI\)](#)

Using contributor insights (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the Music table.
4. Choose the **Monitor** tab.
5. Choose **Turn on CloudWatch Contributor Insights**.

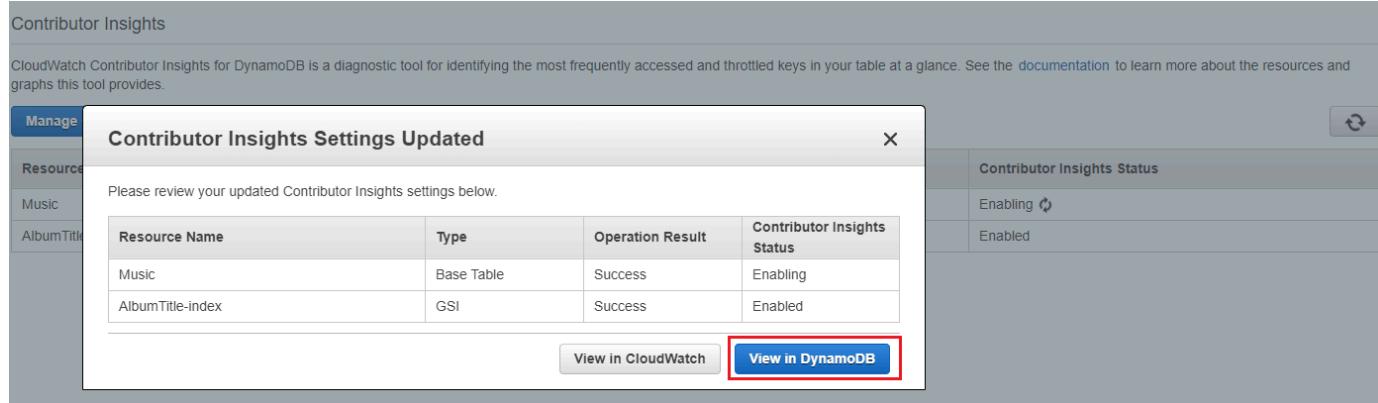


6. In the **Manage Contributor Insights** dialog box, under **Contributor Insights Status**, choose **Enabled** for both the Music base table and the AlbumTitle-index global secondary index. Then choose **Confirm**.

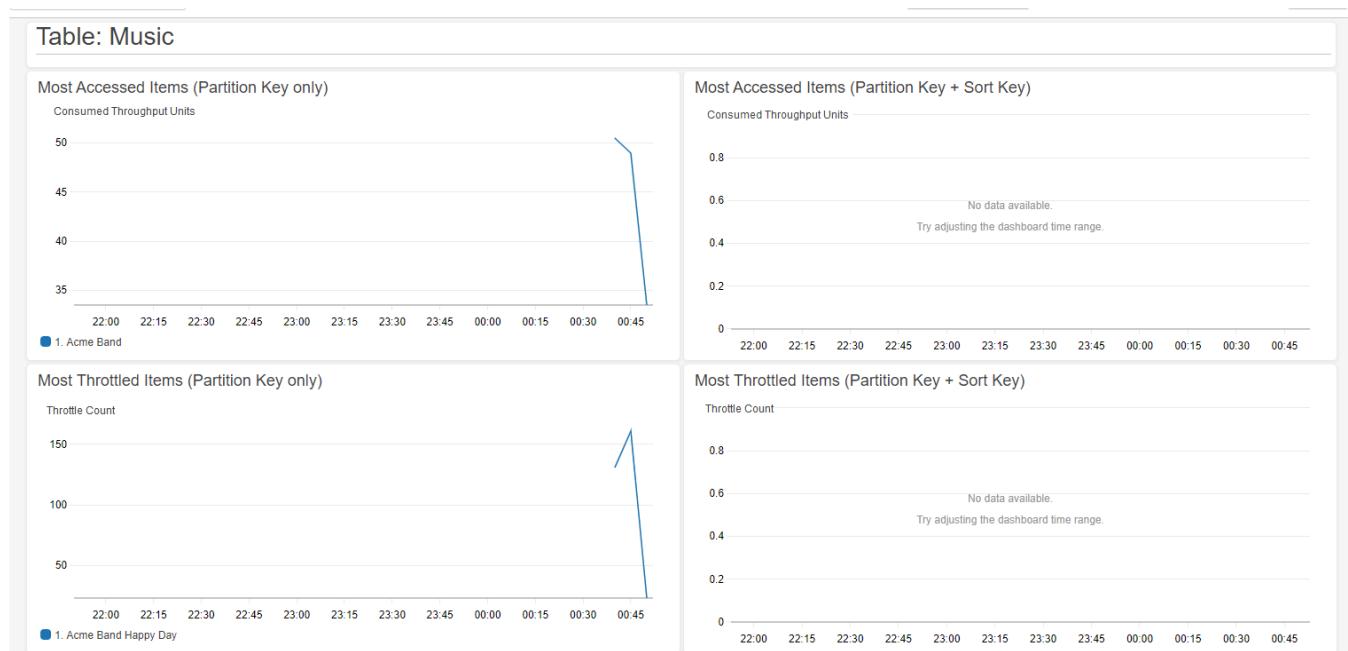


If the operation fails, see [DescribeContributorInsightsFailureException](#) in the *Amazon DynamoDB API Reference* for possible reasons.

7. Choose View in DynamoDB.



8. The Contributor Insights graphs are now visible on the **Contributor Insights** tab for the Music table.



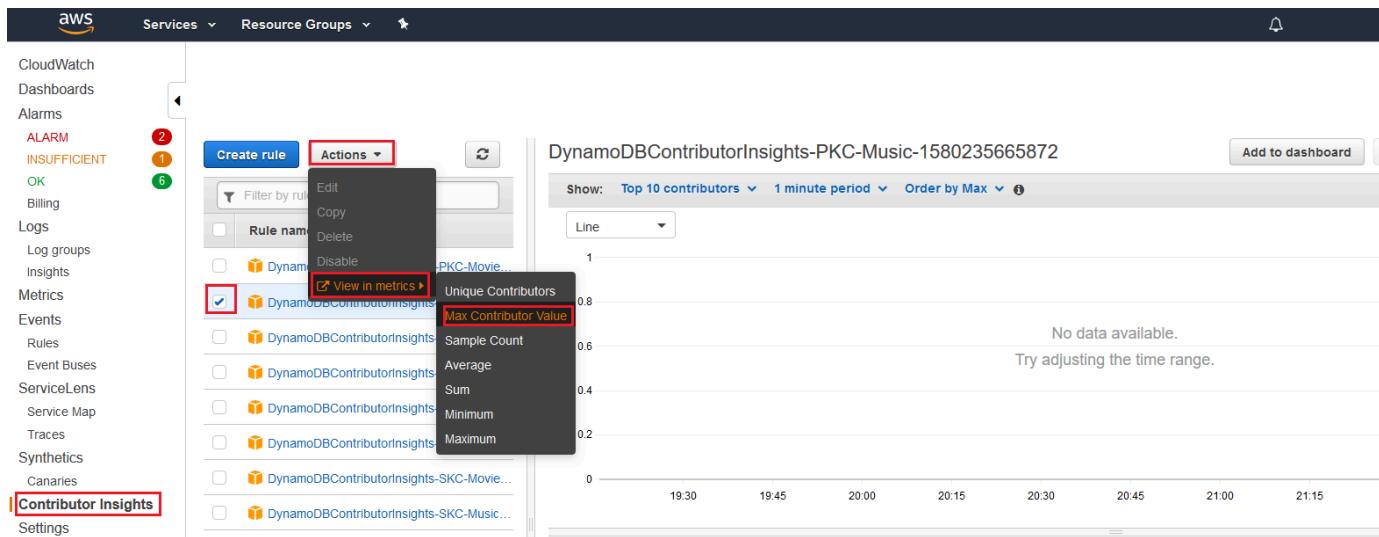
Creating CloudWatch alarms

Follow these steps to create a CloudWatch alarm and be notified when any partition key consumes more than 50,000 [ConsumedThroughputUnits](#).

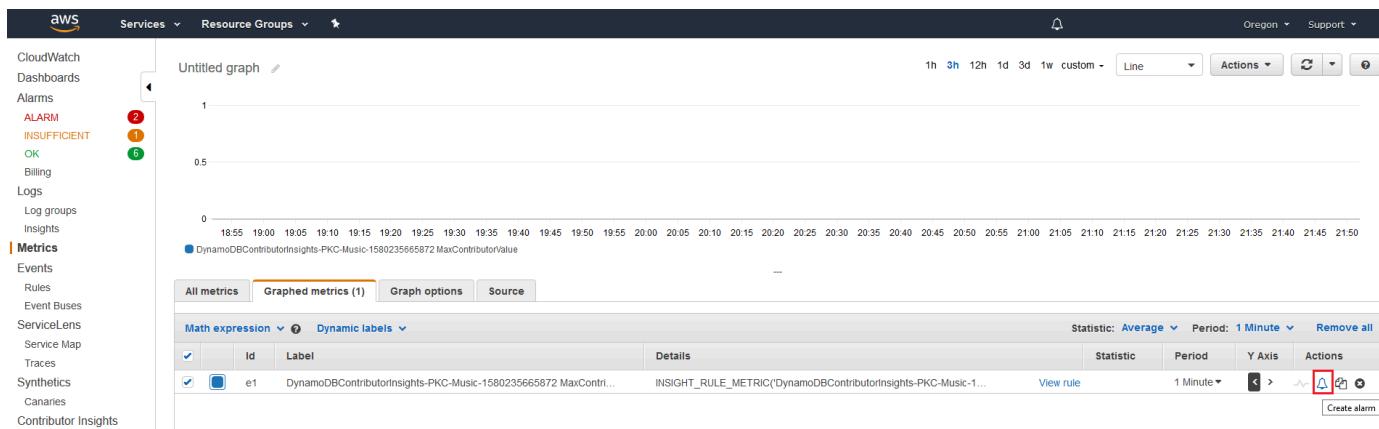
1. Sign in to the AWS Management Console and open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>
2. In the navigation pane on the left side of the console, choose **Contributor Insights**.
3. Choose the **DynamoDBContributorInsights-PKC-Music** rule.
4. Choose the **Actions** drop down.
5. Choose **View in metrics**.
6. Choose **Max Contributor Value**.

Note

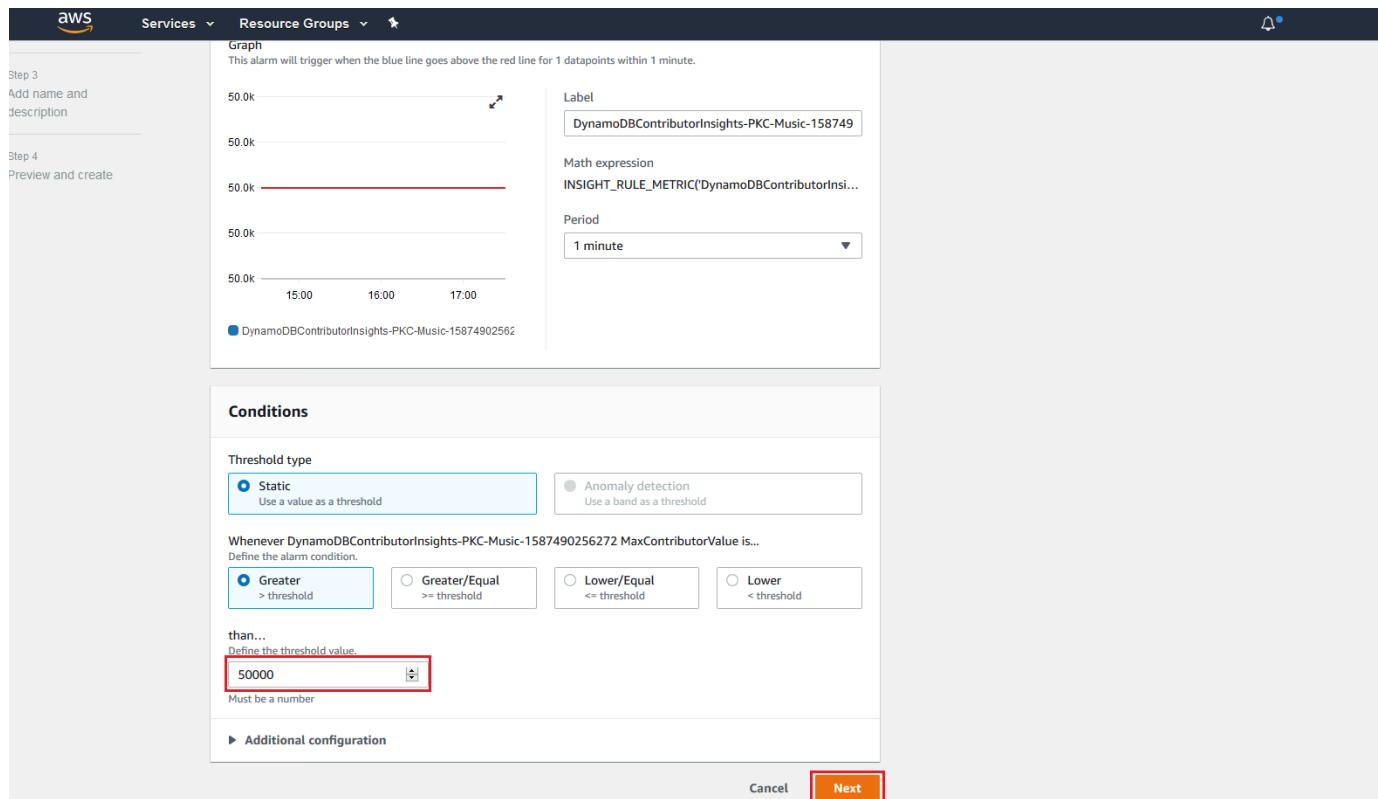
Only Max Contributor Value and Maximum return useful statistics. The other statistics in this list don't return meaningful values.



7. On the Actions column, Choose Create Alarm.



8. Enter a value of 50000 for threshold and choose Next.



- See [Using Amazon CloudWatch alarms](#) for details on how to configure the notification for the alarm.

Using contributor insights (AWS CLI)

- Enable CloudWatch Contributor Insights for DynamoDB on the Music base table.

```
aws dynamodb update-contributor-insights --table-name Music --contributor-insights-action=ENABLE
```

- Enable Contributor Insights for DynamoDB on the AlbumTitle-index global secondary index.

```
aws dynamodb update-contributor-insights --table-name Music --index-name AlbumTitle-index --contributor-insights-action=ENABLE
```

- Get the status and rules for the Music table and all its indexes.

```
aws dynamodb describe-contributor-insights --table-name Music
```

4. Disable CloudWatch Contributor Insights for DynamoDB on the AlbumTitle-index global secondary index.

```
aws dynamodb update-contributor-insights --table-name Music --index-name  
AlbumTitle-index --contributor-insights-action=DISABLE
```

5. Get the status of the Music table and all its indexes.

```
aws dynamodb list-contributor-insights --table-name Music
```

Using IAM with CloudWatch contributor insights for DynamoDB

The first time that you enable Amazon CloudWatch Contributor Insights for Amazon DynamoDB, DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role, `AWSServiceRoleForDynamoDBCloudWatchContributorInsights`, allows DynamoDB to manage CloudWatch Contributor Insights rules on your behalf. Don't delete this service-linked role. If you delete it, all your managed rules will no longer be cleaned up when you delete your table or global secondary index.

For more information about service-linked roles, see [Using service-linked roles in the IAM User Guide](#).

The following permissions are required:

- To enable or disable CloudWatch Contributor Insights for DynamoDB, you must have `dynamodb:UpdateContributorInsights` permission on the table or index.
- To view CloudWatch Contributor Insights for DynamoDB graphs, you must have `cloudwatch:GetInsightRuleReport` permission.
- To describe CloudWatch Contributor Insights for DynamoDB for a given DynamoDB table or index, you must have `dynamodb:DescribeContributorInsights` permission.
- To list CloudWatch Contributor Insights for DynamoDB statuses for each table and global secondary index, you must have `dynamodb>ListContributorInsights` permission.

Example: Enable or disable CloudWatch contributor insights for DynamoDB

The following IAM policy grants permissions to enable or disable CloudWatch Contributor Insights for DynamoDB.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iam:CreateServiceLinkedRole",  
            "Resource": "arn:aws:iam::*:role/aws-service-role/  
contributorinsights.dynamodb.amazonaws.com/  
AWSServiceRoleForDynamoDBCloudWatchContributorInsights",  
            "Condition": {"StringLike": {"iam:AWSPropertyName":  
"contributorinsights.dynamodb.amazonaws.com"}}  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:UpdateContributorInsights"  
            ],  
            "Resource": "arn:aws:dynamodb:*:*:table/*"  
        }  
    ]  
}
```

For tables encrypted by KMS key, the user needs to have kms:Decrypt permissions in order to update Contributor Insights.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iam:CreateServiceLinkedRole",  
            "Resource": "arn:aws:iam::*:role/aws-service-role/  
contributorinsights.dynamodb.amazonaws.com/  
AWSServiceRoleForDynamoDBCloudWatchContributorInsights",  
            "Condition": {"StringLike": {"iam:AWSPropertyName":  
"contributorinsights.dynamodb.amazonaws.com"}}  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:UpdateContributorInsights"  
            ],  
            "Resource": "arn:aws:dynamodb:*:*:table/*"  
        }  
    ]  
}
```

```
        },
        {
            "Effect": "Allow",
            "Resource": "arn:aws:kms:*:*:key/*",
            "Action": [
                "kms:Decrypt"
            ],
        }
    ]
}
```

Example: Retrieve CloudWatch contributor insights rule report

The following IAM policy grants permissions to retrieve CloudWatch Contributor Insights rule report.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetInsightRuleReport"
            ],
            "Resource": "arn:aws:cloudwatch:*:*:insight-rule/
DynamoDBContributorInsights*"
        }
    ]
}
```

Example: Selectively apply CloudWatch contributor insights for DynamoDB permissions based on resource

The following IAM policy grants permissions to allow the `ListContributorInsights` and `DescribeContributorInsights` actions and denies the `UpdateContributorInsights` action for a specific global secondary index.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {

```

```
        "Effect": "Allow",
        "Action": [
            "dynamodb>ListContributorInsights",
            "dynamodb>DescribeContributorInsights"
        ],
        "Resource": "*"
    },
    {
        "Effect": "Deny",
        "Action": [
            "dynamodb>UpdateContributorInsights"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/
Author-index"
    }
]
```

Using service-linked roles for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to CloudWatch Contributor Insights for DynamoDB. Service-linked roles are predefined by CloudWatch Contributor Insights for DynamoDB and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up CloudWatch Contributor Insights for DynamoDB easier because you don't have to manually add the necessary permissions. CloudWatch Contributor Insights for DynamoDB defines the permissions of its service-linked roles, and unless defined otherwise, only CloudWatch Contributor Insights for DynamoDB can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB uses the service-linked role named **AWSServiceRoleForDynamoDBCloudWatchContributorInsights**. The purpose of the service-linked

role is to allow Amazon DynamoDB to manage Amazon CloudWatch Contributor Insights rules created for DynamoDB tables and global secondary indexes, on your behalf.

The `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role trusts the following services to assume the role:

- `contributorinsights.dynamodb.amazonaws.com`

The role permissions policy allows CloudWatch Contributor Insights for DynamoDB to complete the following actions on the specified resources:

- Action: Create and manage Insight Rules on `DynamoDBContributorInsights`

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

Creating a service-linked role for CloudWatch Contributor Insights for DynamoDB

You don't need to manually create a service-linked role. When you enable Contributor Insights in the AWS Management Console, the AWS CLI, or the AWS API, CloudWatch Contributor Insights for DynamoDB creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you enable Contributor Insights, CloudWatch Contributor Insights for DynamoDB creates the service-linked role for you again.

Editing a service-linked role for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB does not allow you to edit the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

Deleting a service-linked role for CloudWatch Contributor Insights for DynamoDB

You don't need to manually delete the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` role. When you disable

Contributor Insights in the AWS Management Console, the AWS CLI, or the AWS API, CloudWatch Contributor Insights for DynamoDB cleans up the resources.

You can also use the IAM console, the AWS CLI or the AWS API to manually delete the service-linked role. To do this, you must first manually clean up the resources for your service-linked role and then you can manually delete it.

 **Note**

If the CloudWatch Contributor Insights for DynamoDB service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the AWSServiceRoleForDynamoDBCloudWatchContributorInsights service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

Using AWS User Notifications User Notifications with Amazon DynamoDB

You can use [User Notifications](#) to set up delivery channels to get notified about DynamoDB events. You receive a notification when an event matches a rule that you specify. You can receive notifications for events through multiple channels, including email, [AWS Chatbot](#) chat notifications, or [AWS Console Mobile Application](#) push notifications. You can also see notifications in the [Console Notifications Center](#). User Notifications supports aggregation, which can reduce the number of notifications you receive during specific events.

Best practices for designing and architecting with DynamoDB

Use this section to quickly find recommendations for maximizing performance and minimizing throughput costs when working with Amazon DynamoDB.

Topics

- [NoSQL design for DynamoDB](#)
- [Using deletion protection to protect your table](#)
- [Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload](#)
- [Best practices for designing and using partition keys effectively](#)
- [Best practices for using sort keys to organize data](#)
- [Best practices for using secondary indexes in DynamoDB](#)
- [Best practices for storing large items and attributes](#)
- [Best practices for handling time series data in DynamoDB](#)
- [Best practices for managing many-to-many relationships](#)
- [Best practices for implementing a hybrid database system](#)
- [Best practices for modeling relational data in DynamoDB](#)
- [Best practices for querying and scanning data](#)
- [Best practices for DynamoDB table design](#)
- [Best practices for DynamoDB global table design](#)
- [Best practices for managing the control plane in DynamoDB](#)
- [Best Practices for Understanding your AWS Billing and Usage Reports](#)
- [Considerations when using AWS PrivateLink for Amazon DynamoDB](#)

NoSQL design for DynamoDB

NoSQL database systems like Amazon DynamoDB use alternative models for data management, such as key-value pairs or document storage. When you switch from a relational database

management system to a NoSQL database system like DynamoDB, it's important to understand the key differences and specific design approaches.

Topics

- [Differences between relational data design and NoSQL](#)
- [Two key concepts for NoSQL design](#)
- [Approaching NoSQL design](#)
- [NoSQL Workbench for DynamoDB](#)

Differences between relational data design and NoSQL

Relational database systems (RDBMS) and NoSQL databases have different strengths and weaknesses:

- In RDBMS, data can be queried flexibly, but queries are relatively expensive and don't scale well in high-traffic situations (see [First steps for modeling relational data in DynamoDB](#)).
- In a NoSQL database such as DynamoDB, data can be queried efficiently in a limited number of ways, outside of which queries can be expensive and slow.

These differences make database design different between the two systems:

- In RDBMS, you design for flexibility without worrying about implementation details or performance. Query optimization generally doesn't affect schema design, but normalization is important.
- In DynamoDB, you design your schema specifically to make the most common and important queries as fast and as inexpensive as possible. Your data structures are tailored to the specific requirements of your business use cases.

Two key concepts for NoSQL design

NoSQL design requires a different mindset than RDBMS design. For an RDBMS, you can go ahead and create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. You can organize each type of data into its own table.

How NoSQL design is different

- By contrast, you shouldn't start designing your schema for DynamoDB until you know the questions it will need to answer. Understanding the business problems and the application use cases up front is essential.
- You should maintain as few tables as possible in a DynamoDB application. Having fewer tables keeps things more scalable, requires less permissions management, and reduces overhead for your DynamoDB application. It can also help keep backup costs lower overall.

Approaching NoSQL design

The first step in designing your DynamoDB application is to identify the specific query patterns that the system must satisfy.

In particular, it is important to understand three fundamental properties of your application's access patterns before you begin:

- **Data size:** Knowing how much data will be stored and requested at one time will help determine the most effective way to partition the data.
- **Data shape:** Instead of reshaping data when a query is processed (as an RDBMS system does), a NoSQL database organizes data so that its shape in the database corresponds with what will be queried. This is a key factor in increasing speed and scalability.
- **Data velocity:** DynamoDB scales by increasing the number of physical partitions that are available to process queries, and by efficiently distributing data across those partitions. Knowing in advance what the peak query loads will be might help determine how to partition data to best use I/O capacity.

After you identify specific query requirements, you can organize data according to general principles that govern performance:

- **Keep related data together.** Research on routing-table optimization 20 years ago found that "locality of reference" was the single most important factor in speeding up response time: keeping related data together in one place. This is equally true in NoSQL systems today, where keeping related data in close proximity has a major impact on cost and performance. Instead of distributing related data items across multiple tables, you should keep related items in your NoSQL system as close together as possible.

As a general rule, you should maintain as few tables as possible in a DynamoDB application.

Exceptions are cases where high-volume time series data are involved, or datasets that have very different access patterns. A single table with inverted indexes can usually enable simple queries to create and retrieve the complex hierarchical data structures required by your application.

- **Use sort order.** Related items can be grouped together and queried efficiently if their key design causes them to sort together. This is an important NoSQL design strategy.
- **Distribute queries.** It is also important that a high volume of queries not be focused on one part of the database, where they can exceed I/O capacity. Instead, you should design data keys to distribute traffic evenly across partitions as much as possible, avoiding "hot spots."
- **Use global secondary indexes.** By creating specific global secondary indexes, you can enable different queries than your main table can support, and that are still fast and relatively inexpensive.

These general principles translate into some common design patterns that you can use to model data efficiently in DynamoDB.

NoSQL Workbench for DynamoDB

[NoSQL Workbench for DynamoDB](#) is a cross-platform, client-side GUI application that you can use for modern database development and operations. It's available for Windows, macOS, and Linux. NoSQL Workbench is a visual development tool that provides data modeling, data visualization, sample data generation, and query development features to help you design, create, query, and manage DynamoDB tables. With NoSQL Workbench for DynamoDB, you can build new data models from, or design models based on, existing data models that satisfy your application's data access patterns. You can also import and export the designed data model at the end of the process. For more information, see [Building data models with NoSQL Workbench](#)

Using deletion protection to protect your table

Deletion protection can keep your table from being accidentally deleted. This section describes some best practices for using deletion protection.

- For all active production tables, the best practice is to turn on the deletion protection setting and protect these tables from accidental deletion. This also applies to global replicas.

- When serving application development use cases, if the table management workflow includes frequently deleting and recreating development and staging tables then the deletion protection setting can be turned off. This will allow intentional deletion of such tables by authorized IAM principals.

For more information about deletion protection, see [Using deletion protection](#).

Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload

This section describes the Amazon DynamoDB Well-Architected Lens, a collection of design principles and guidance for designing well-architected DynamoDB workloads.

Optimizing costs on DynamoDB tables

This section covers best practices on how to optimize costs for your existing DynamoDB tables. You should look at the following strategies to see which cost optimization strategy best suits your needs and approach them iteratively. Each strategy will provide an overview of what might be impacting your costs, what signs to look for, and prescriptive guidance on how to reduce them.

Topics

- [Evaluate your costs at the table level](#)
- [Evaluate your table capacity mode](#)
- [Evaluate your table's auto scaling settings](#)
- [Evaluate your table class selection](#)
- [Identifying your unused resources](#)
- [Evaluate your table usage patterns](#)
- [Evaluate your Streams usage](#)
- [Evaluate your provisioned capacity for right-sized provisioning](#)

Evaluate your costs at the table level

The Cost Explorer tool found within the AWS Management Console allows you to see costs broken down by type, such as read, write, storage and backup charges. You can also see these costs summarized by period such as month or day.

One challenge administrators can face is when the costs of only one particular table need to be reviewed. Some of this data is available via the DynamoDB console or via calls to the `DescribeTable` API, however Cost Explorer does not, by default, allow you to filter or group by costs associated with a specific table. This section will show you how to use tagging to perform individual table cost analysis in Cost Explorer.

Topics

- [How to view the costs of a single DynamoDB table](#)
- [Cost Explorer's default view](#)
- [How to use and apply table tags in Cost Explorer](#)

How to view the costs of a single DynamoDB table

Both the Amazon DynamoDB AWS Management Console and the `DescribeTable` API will show you information about a single table, including the primary key schema, any indexes on the table, and the size and item count of the table and any indexes. The size of the table, plus the size of the indexes, can be used to calculate the monthly storage cost for your table. For example, \$0.25 per GB in the us-east-1 region.

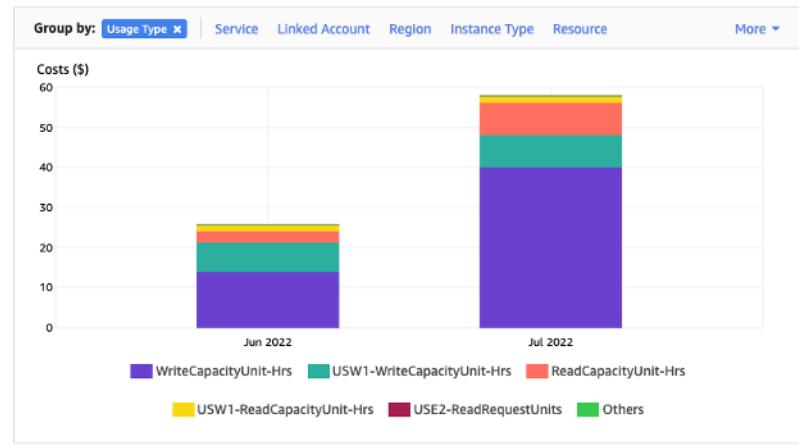
If the table is in provisioned capacity mode, the current RCU and WCU settings are returned as well. These could be used to calculate the current read and write costs for the table, but these costs could change, especially if the table has been configured with Auto Scaling.

Note

If the table is in on-demand capacity mode, then `DescribeTable` will not help estimate throughput costs, as these are billed based on actual, not provisioned usage in any one period.

Cost Explorer's default view

Cost Explorer's default view provides charts showing the cost of consumed resources such as throughput and storage. You can choose to group costs by period, such as totals by month or by day. The costs of storage, reads, writes, and other features can be broken out and compared as well.



How to use and apply table tags in Cost Explorer

By default, Cost Explorer does not provide a summary of the costs for any one specific table, as it will combine the costs of multiple tables into a total. However, you can use [AWS resource tagging](#) to identify each table by a metadata tag. Tags are key-value pairs you can use for a variety of purposes, such as to identify all resources belonging to a project or department. For this example, we'll assume you have a table named **MyTable**.

1. Set a tag with the key of **table_name** and the value of **MyTable**.
2. [Activate the tag within Cost Explorer](#) and then filter on the tag value to gain more visibility into each table's costs.

Note

It may take one or two days for the tag to start appearing in Cost Explorer

You can set metadata tags yourself in the console, or via automation such as the AWS CLI or AWS SDK. Consider requiring a **table_name** tag to be set as part of your organization's new table creation process. For existing tables, there is a Python utility available that will find and apply these tags to all existing tables in a certain region in your account. See [Eponymous Table Tagger on GitHub](#) for more details.

Evaluate your table capacity mode

This section provides an overview of how to select the appropriate capacity mode for your DynamoDB table. Each mode is tuned to meet the needs of a different workload in terms of

responsiveness to change in throughput, as well as how that usage is billed. You must balance these factors when making our decision.

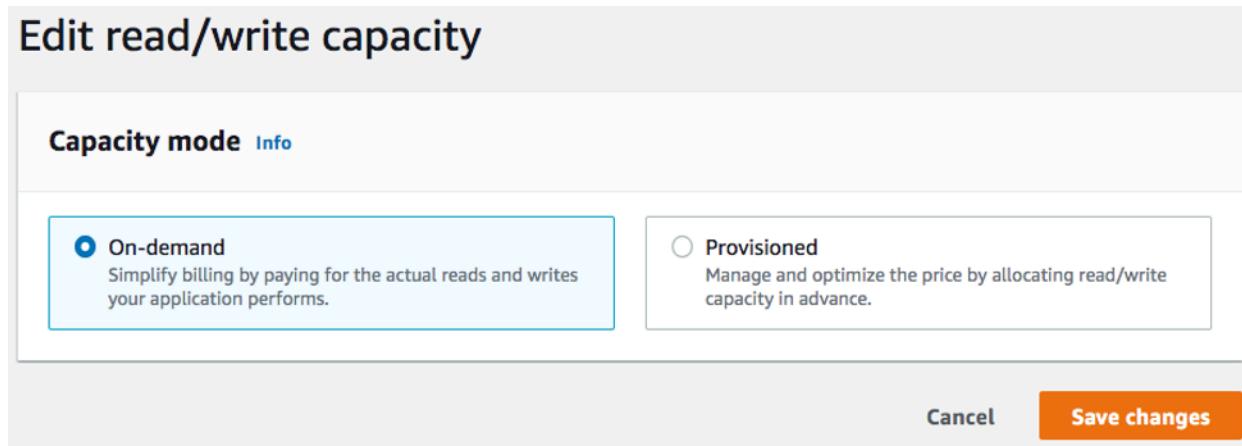
Topics

- [What table capacity modes are available](#)
- [When to select on-demand capacity mode](#)
- [When to select provisioned capacity mode](#)
- [Additional factors to consider when choosing a table capacity mode](#)

What table capacity modes are available

When you create a DynamoDB table, you must select either on-demand or provisioned capacity mode. You can switch between read/write capacity modes once every 24 hours. The only exception to this is if you switch a provisioned mode table to on-demand mode: you can switch back to provisioned mode in the same 24-hour period.

Edit read/write capacity



On-demand capacity mode

The on-demand capacity mode is designed to eliminate the need to plan or provision the capacity of your DynamoDB table. In this mode, your table will instantly accommodate requests to your table without the need to scale any resources up or down (up to twice the previous peak throughput of the table).

On-demand tables are billed by counting the number of actual requests against the table, so you will only pay for what you use rather than what has been provisioned.

Provisioned capacity mode

The provisioned capacity mode is a more traditional model where you can define how much capacity the table has available for requests either directly or with the assistance of auto scaling. Because a specific capacity is provisioned for the table at any given time, billing is based off of the capacity provisioned rather than the number of requests. Going over the allocated capacity can also cause the table to reject requests and reduce the experience of your applications users.

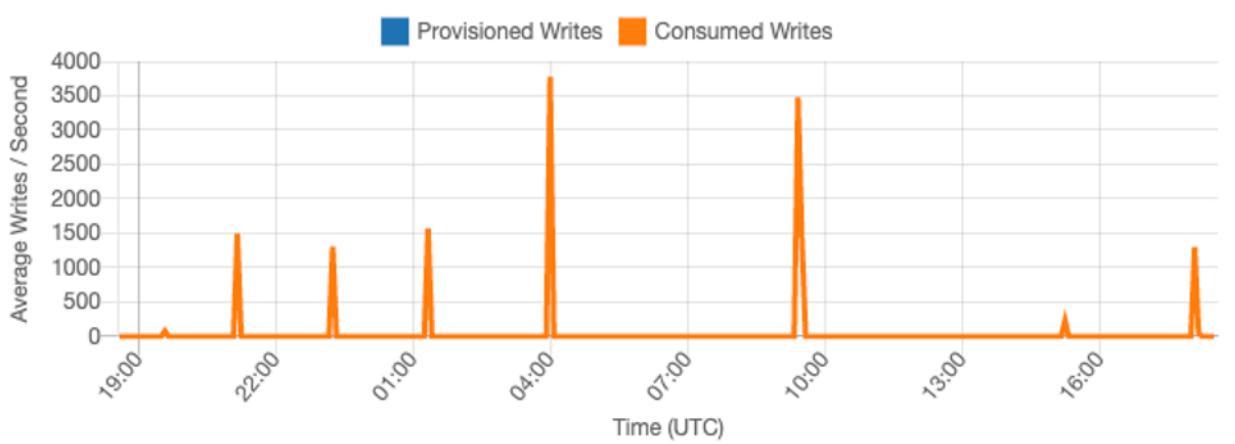
Provisioned capacity mode requires a balance between not overprovisioning or under provisioning the table to keep both throttling low and costs tuned.

When to select on-demand capacity mode

When optimizing for cost, on-demand mode is your best choice when you have a workload similar to the following graph.

The following factors contribute to this type of workload:

- Unpredictable request timing (resulting in traffic spikes)
- Variable volume of requests (resulting from batch workloads)
- Drops to zero or below 18% of the peak for a given hour (resulting from dev or test environments)



For workloads with the above factors, using auto scaling to maintain enough capacity on the table to respond to spikes in traffic will likely lead to the table being overprovisioned and costing more than necessary or the table being under provisioned and requests being unnecessarily throttled.

Because on-demand tables are billed by request, there is nothing further we need to do at the table level to optimize for cost. You should regularly evaluate your on-demand tables to verify

the workload still has the above factors. If the workload has stabilized, consider changing to provisioned mode to further optimize cost.

When to select provisioned capacity mode

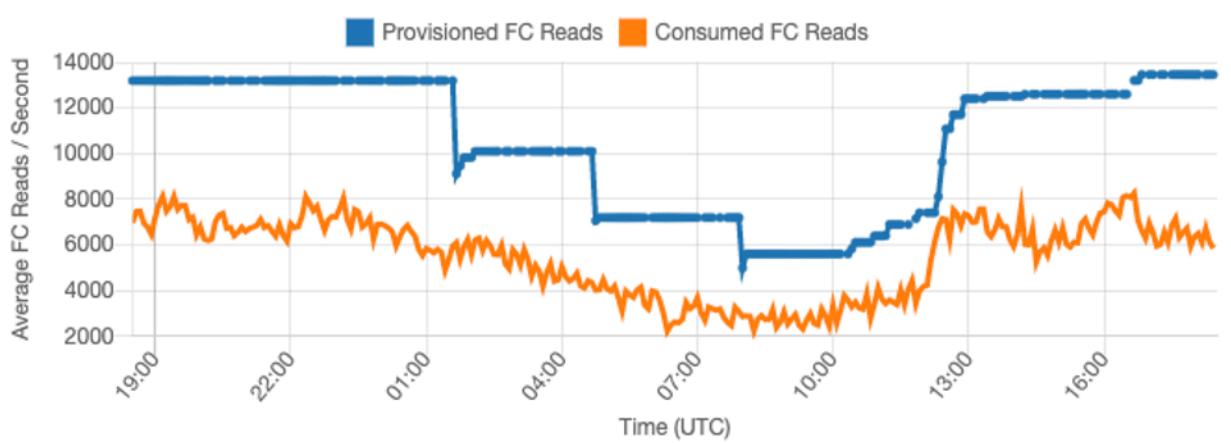
An ideal workload for provisioned capacity mode is one with a more predictable usage pattern like the graph below.

Note

We recommend reviewing metrics at a fine-grained period, such as 14 days or 24 hours, before taking action on your provisioned capacity.

The following factors contribute to this type of workload:

- Predictable/cyclical traffic for a given hour or day
- Limited short term bursts of traffic



Since the traffic volumes within a given hour or day are more stable, we can set the provisioned capacity of the table relatively close to the actual consumed capacity of the table. Cost optimizing a provisioned capacity table is ultimately an exercise in getting the provisioned capacity (blue line) as close to the consumed capacity (orange line) as possible without increasing ThrottledRequests on the table. The space between the two lines is both wasted capacity as well as insurance against a bad user experience due to throttling.

DynamoDB provides auto scaling for provisioned capacity tables which will automatically balance this on your behalf. This lets you track your consumed capacity throughout the day and set the capacity of the table based on a handful of variables.

The screenshot shows the 'Table capacity' configuration section. It includes two options: 'On-demand' (disabled) and 'Provisioned' (selected). The 'Provisioned' option is described as managing and optimizing price by allocating read/write capacity in advance. Below this, the 'Read capacity' section is shown. It includes an 'Auto scaling' setting (set to 'On') with an 'Info' link, a note about dynamically adjusting provisioned throughput, and three input fields: 'Minimum capacity units' (100), 'Maximum capacity units' (500), and 'Target utilization (%)' (70). There is also an 'Initial provisioned units' field with an 'Info' link containing the value 200.

Minimum capacity units

You can set the minimum capacity of a table to limit throttling, but it will not reduce the cost of the table. If your table has periods of low usage followed by a sudden burst of high usage, setting the minimum can prevent auto scaling from setting the table capacity too low.

Maximum capacity units

You can set the maximum capacity of a table to limit a table scaling higher than intended. Consider applying a maximum for Dev or Test tables where large-scale load testing is not desired. You can set a maximum for any table, but be sure to regularly evaluate this setting against the table baseline when using it in Production to prevent accidental throttling.

Target utilization

Setting the target utilization of the table is the primary means of cost optimization for a provisioned capacity table. Setting a lower percent value here will increase how much the table is

overprovisioned, increasing cost, but reducing the risk of throttling. Setting a higher percent value will decrease how much the table is overprovisioned, but increase the risk of throttling.

Additional factors to consider when choosing a table capacity mode

When deciding between the two modes, there are some additional factors worth considering.

Reserved capacity

For provisioned capacity tables, DynamoDB offers the ability to purchase reserved capacity for your read and write capacity (replicated write capacity units (rWCUs) and Standard-IA tables are currently not eligible). If you chose to purchase reservations for this capacity, you can reduce the cost of the table by a significant percent.

When deciding between the two table modes, consider how much this additional discount will affect the cost of the table. In many cases, even a relatively unpredictable workload can be cheaper to run on an overprovisioned provisioned capacity table with reserved capacity.

Improving predictability of your workload

In some situations, a workload may seemingly have both a predictable and unpredictable pattern. While this can be easily supported with an on-demand table, costs will likely be better if the unpredictable patterns in the workload can be improved.

One of the most common causes of these patterns is batch imports. This type of traffic can often exceed the baseline capacity of the table to such a degree that throttling would occur if it were to run. To keep a workload like this running on a provisioned capacity table, consider the following options:

- If the batch occurs at scheduled times, you can schedule an increase to your auto-scaling capacity before it runs
- If the batch occurs randomly, consider trying to extend the time it runs rather than executing as fast as possible
- Add a ramp up period to the import where the velocity of the import starts small but is slowly increased over a few minutes until auto scaling has had the opportunity to start adjusting table capacity

Evaluate your table's auto scaling settings

This section provides an overview of how to evaluate the auto scaling settings on your DynamoDB tables. [Amazon DynamoDB auto scaling](#) is a feature that manages table and global secondary index (GSI) throughput based on your application traffic and your target utilization metric. This ensures your tables or GSIs will have the required capacity required for your application patterns.

The AWS auto scaling service will monitor your current table utilization and compare it to the target utilization value: `TargetValue`. It will notify you if it is time to increase or decrease the capacity allocated.

Topics

- [Understanding your auto scaling settings](#)
- [How to identify tables with low target utilization \(<=50%\)](#)
- [How to address workloads with seasonal variance](#)
- [How to address spiky workloads with unknown patterns](#)
- [How to address workloads with linked applications](#)

Understanding your auto scaling settings

Defining the correct value for the target utilization, initial step, and final values is an activity that requires involvement from your operations team. This allows you to properly define the values based on historical application usage, which will be used to trigger the AWS auto scaling policies. The utilization target is the percentage of your total capacity that needs to be hit during a period of time before the auto scaling rules apply.

When you set a **high utilization target (a target around 90%)** it means your traffic needs to be higher than 90% for a period of time before the auto scaling kicks in. You should not use a high utilization target unless your application is very constant and doesn't receive spikes in traffic.

When you set a very **low utilization (a target less than 50%)** it means your application would need to reach 50% of the provisioned capacity before it triggers an auto scaling policy. Unless your application traffic grows at a very aggressive rate, this usually translates into unused capacity and wasted resources.

How to identify tables with low target utilization (<=50%)

You can use either the AWS CLI or AWS Management Console to monitor and identify the `TargetValues` for your auto scaling policies in your DynamoDB resources:

AWS CLI

1. Return the entire list of resources by running the following command:

```
aws application-autoscaling describe-scaling-policies --service-namespace dynamodb
```

This command will return the entire list of auto scaling policies that are issued to any DynamoDB resource. If you only want to retrieve the resources from a particular table, you can add the `--resource-id` parameter. For example:

```
aws application-autoscaling describe-scaling-policies --service-namespace dynamodb --resource-id "table/<table-name>"
```

2. Return only the auto scaling policies for a particular GSI by running the following command

```
aws application-autoscaling describe-scaling-policies --service-namespace dynamodb --resource-id "table/<table-name>/index/<gsi-name>"
```

The values we're interested in for the auto scaling policies are highlighted below. We want to ensure that the target value is greater than 50% to avoid over-provisioning. You should obtain a result similar to the following:

```
{
    "ScalingPolicies": [
        {
            "PolicyARN": "arn:aws:autoscaling:<region>:<account-id>:scalingPolicy:<uuid>:resource/dynamodb/table/<table-name>/index/<index-name>:policyName/$<full-gsi-name>-scaling-policy",
            "PolicyName": "$<full-gsi-name>",
            "ServiceNamespace": "dynamodb",
            "ResourceId": "table/<table-name>/index/<index-name>",
            "ScalableDimension": "dynamodb:index:WriteCapacityUnits",
            "PolicyType": "TargetTrackingScaling",
            "TargetTrackingScalingPolicyConfiguration": {
                "TargetValue": 70.0,
                "PredefinedMetricSpecification": {
                    "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
                }
            },
            "Alarms": [

```

```
    ...
],
"CreationTime": "2022-03-04T16:23:48.641000+10:00"
},
{
    "PolicyARN": "arn:aws:autoscaling:<region>:<account-id>:scalingPolicy:<uuid>:resource/dynamodb/table/<table-name>/index/<index-name>:policyName/$<full-gsi-name>-scaling-policy",
    "PolicyName": "$<full-gsi-name>",
    "ServiceNamespace": "dynamodb",
    "ResourceId": "table/<table-name>/index/<index-name>",
    "ScalableDimension": "dynamodb:index:ReadCapacityUnits",
    "PolicyType": "TargetTrackingScaling",
    "TargetTrackingScalingPolicyConfiguration": {
        "TargetValue": 70.0,
        "PredefinedMetricSpecification": {
            "PredefinedMetricType": "DynamoDBReadCapacityUtilization"
        }
    },
    "Alarms": [
        ...
    ],
    "CreationTime": "2022-03-04T16:23:47.820000+10:00"
}
]
}
```

AWS Management Console

1. Log into the AWS Management Console and navigate to the CloudWatch service page at [Getting Started with the AWS Management Console](#). Select the appropriate AWS region if necessary.
2. On the left navigation bar, select **Tables**. On the **Tables** page, select the table's **Name**.
3. On the **Table Details** page under the **Additional Settings** tab, review your table's auto scaling settings.

Overview	Indexes	Monitor	Global tables	Backups	Exports and streams	Additional settings
Read/write capacity						
The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.						
Capacity mode	Provisioned					
Table capacity						
Read capacity auto scaling	On	Write capacity auto scaling	On			
Provisioned read capacity units	5	Provisioned write capacity units	5			
Provisioned range for reads	1 - 10	Provisioned range for writes	1 - 10			
Target read capacity utilization	70%	Target write capacity utilization	70%			
▶ Index capacity						

For indexes, expand the **Index Capacity** tab to review the index's auto scaling settings.

Read/write capacity	The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.	
Capacity mode	Provisioned	
Table capacity		
Read capacity auto scaling	On	Write capacity auto scaling
Provisioned read capacity units	5	On
Provisioned range for reads	1 - 10	Provisioned write capacity units
Target read capacity utilization	70%	5
▼ Index capacity		
Index name	Read capacity	Write capacity
GSI1PK-GSI1SK-index	Range: 1 - 10 Auto scaling at 70% Current provisioned units: 5	Range: 1 - 10 Auto scaling at 70% Current provisioned units: 5

If your target utilization values are less than or equal to 50%, you should explore your table utilization metrics to see if they are [under-provisioned or over-provisioned](#).

How to address workloads with seasonal variance

Consider the following scenario: your application is operating under a minimum average value most of the time, but the utilization target is low so your application can react quickly to events that happen at certain hours in the day and you have enough capacity and avoid getting throttled. This scenario is common when you have an application that is very busy during normal office hours (9 AM to 5 PM) but then it works at a base level during after hours. Since some users will start to connect before 9 am, the application uses this low threshold to ramp up quickly to get to the *required* capacity during peak hours.

This scenario could look like this:

- Between 5 PM and 9 AM the ConsumedWriteCapacity units stay between 90 and 100
- Users start to connect to the application before 9 AM and the capacity units increases considerably (the maximum value you've seen is 1500 WCU)
- On average, your application usage varies between 800 to 1200 during working hours

If the previous scenario applies to you, consider using [scheduled auto scaling](#), where your table could still have an application auto scaling rule configured, but with a less aggressive target utilization that only provisions the extra capacity at the specific intervals you require.

You can use AWS CLI to execute the following steps to create a scheduled auto scaling rule that will execute based on the time of day and the day of the week.

1. Register your DynamoDB table or GSI as scalable target with Application Auto Scaling. A scalable target is a resource that Application Auto Scaling can scale out or in.

```
aws application-autoscaling register-scalable-target \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
    --resource-id table/<table-name> \
    --min-capacity 90 \
    --max-capacity 1500
```

2. Set up scheduled actions according to your requirements.

We'll need two rules to cover the scenario: one to scale up and another to scale down. The first rule to scale up the scheduled action:

```
aws application-autoscaling put-scheduled-action \
--service-namespace dynamodb \
--scalable-dimension dynamodb:table:WriteCapacityUnits \
--resource-id table/<table-name> \
--scheduled-action-name my-8-5-scheduled-action \
--scalable-target-action MinCapacity=800,MaxCapacity=1500 \
--schedule "cron(45 8 ? * MON-FRI *)" \
--timezone "Australia/Brisbane"
```

The second rule to scale down the scheduled action:

```
aws application-autoscaling put-scheduled-action \
--service-namespace dynamodb \
--scalable-dimension dynamodb:table:WriteCapacityUnits \
--resource-id table/<table-name> \
--scheduled-action-name my-5-8-scheduled-down-action \
--scalable-target-action MinCapacity=90,MaxCapacity=1500 \
--schedule "cron(15 17 ? * MON-FRI *)" \
--timezone "Australia/Brisbane"
```

- Run the following command to validate both rules have been activated:

```
aws application-autoscaling describe-scheduled-actions --service-namespace dynamodb
```

You should get a result like this:

```
{  
    "ScheduledActions": [  
        {  
            "ScheduledActionName": "my-5-8-scheduled-down-action",  
            "ScheduledActionARN":  
                "arn:aws:autoscaling:<region>:<account>:scheduledAction:<uuid>:resource/dynamodb/  
                table/<table-name>:scheduledActionName/my-5-8-scheduled-down-action",  
            "ServiceNamespace": "dynamodb",  
            "Schedule": "cron(15 17 ? * MON-FRI *)",  
            "Timezone": "Australia/Brisbane",  
            "ResourceId": "table/<table-name>",  
            "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
```

```

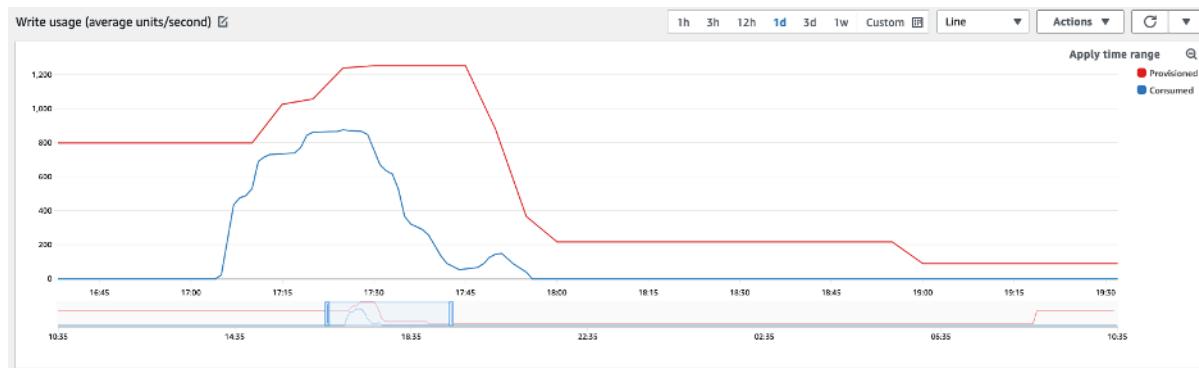
        "ScalableTargetAction": {
            "MinCapacity": 90,
            "MaxCapacity": 1500
        },
        "CreationTime": "2022-03-15T17:30:25.100000+10:00"
    },
    {
        "ScheduledActionName": "my-8-5-scheduled-action",
        "ScheduledActionARN":
            "arn:aws:autoscaling:<region>:<account>:scheduledAction:<uuid>:resource/dynamodb/
            table/<table-name>:scheduledActionName/my-8-5-scheduled-action",
        "ServiceNamespace": "dynamodb",
        "Schedule": "cron(45 8 ? * MON-FRI *)",
        "Timezone": "Australia/Brisbane",
        "ResourceId": "table/<table-name>",
        "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
        "ScalableTargetAction": {
            "MinCapacity": 800,
            "MaxCapacity": 1500
        },
        "CreationTime": "2022-03-15T17:28:57.816000+10:00"
    }
]
}

```

The following picture shows a sample workload that always keeps the 70% target utilization. Notice how the auto scaling rules are still applying and the throughput will not be reduced.



Zooming in, we can see there was a spike in the application that triggered the 70% auto scaling threshold, forcing the autoscaling to kick in and provide the extra capacity required for the table. The scheduled auto scaling action will affect maximum and minimum values, and it is your responsibility to set them up.



How to address spiky workloads with unknown patterns

In this scenario, the application uses a very low utilization target because you don't know the application patterns yet, and you want to ensure your workload is not throttled.

Consider using [on-demand capacity mode](#) instead. On-demand tables are perfect for spiky workloads where you don't know the traffic patterns. With on-demand capacity mode, you pay per request for the data reads and writes your application performs on your tables. You do not need to specify how much read and write throughput you expect your application to perform, as DynamoDB instantly accommodates your workloads as they ramp up or down.

How to address workloads with linked applications

In this scenario, the application depends on other systems, like batch processing scenarios where you can have big spikes in traffic according to events in the application logic.

Consider developing custom auto scaling logic that reacts to those events where you can increase table capacity and TargetValues depending on your specific needs. You could benefit from Amazon EventBridge and use a combination of AWS services like Lambda and Step Functions to react to your specific application needs.

Evaluate your table class selection

This section provides an overview of how to select the appropriate table class for your DynamoDB table. With the launch of the Standard Infrequent-Access (Standard-IA) table class, you now have the ability to optimize a table for either lower storage cost or lower throughput cost.

Topics

- [What table classes are available](#)
- [When to select the DynamoDB Standard table class](#)
- [When to select DynamoDB Standard-IA table class](#)
- [Additional factors to consider when choosing a table class](#)

What table classes are available

When you create a DynamoDB Table, you must select either DynamoDB Standard or DynamoDB Standard-IA for the table class. The table class can be changed twice in a 30-day period, so you can always change it in the future. Selecting either table class has no effect on table performance, availability, reliability, or durability.

Update table class

Table class

Select table class to optimize your table's cost based on your workload requirements and data access patterns.

Choose table class

DynamoDB Standard

The default general-purpose table class. Recommended for the vast majority of tables that store frequently accessed data, with throughput (reads and writes) as the dominant table cost.

DynamoDB Standard-IA

Recommended for tables that store data that is infrequently accessed, with storage as the dominant table cost.

 Table class updates is a background process. The time to update your table class depends on your table traffic, storage size, and other related variables. You can still access your table normally while it is converted. Note that no more than two table class updates on your table are allowed in a 30-day trailing period. [Learn more](#) 

[Cancel](#)

[Save changes](#)

Standard table class

The Standard table class is the default option for new tables. This option maintains the original billing balance of DynamoDB which offers a balance of throughput and storage costs for tables with frequently accessed data.

Standard-IA table class

The Standard-IA table class offers a lower storage cost (~60% lower) for workloads that require long-term storage of data with infrequent updates or reads. Since the class is optimized for infrequent access, reads and writes will be billed at a slightly higher cost (~25% higher) than the Standard table class.

When to select the DynamoDB Standard table class

DynamoDB Standard table class is best suited for tables whose storage cost is approximately 50% or less of the overall monthly cost of the table. This cost balance is indicative of a workload that regularly accesses or updates items already stored within DynamoDB.

When to select DynamoDB Standard-IA table class

DynamoDB Standard-IA table class is best suited for tables whose storage cost is approximately 50% or more of the overall monthly cost of the table. This cost balance is indicative of a workload that creates or reads fewer items per month than it keeps in storage.

A common use for the Standard-IA table class is moving less frequently accessed data to individual Standard-IA tables. For further information, see [Optimizing the storage costs of your workloads with Amazon DynamoDB Standard-IA table class](#).

Additional factors to consider when choosing a table class

When deciding between the two table classes, there are some additional factors worth considering as part of your decision.

Reserved capacity

Purchasing reserved capacity for tables using the Standard-IA table class is currently not supported. When transitioning from a Standard table with reserved capacity to a Standard-IA table without reserved capacity, you may not see a cost benefit.

Identifying your unused resources

This section provides an overview of how to evaluate your unused resources regularly. As your application requirements evolve you should ensure no resources are unused and contributing to unnecessary Amazon DynamoDB costs. The procedures described below will use Amazon CloudWatch metrics to identify unused resources and will help you identify and take action on those resources to reduce costs.

You can monitor DynamoDB using CloudWatch, which collects and processes raw data from DynamoDB into readable, near real-time metrics. These statistics are retained for a period of time, so that you can access historical information to better understand your utilization. By default, DynamoDB metric data is sent to CloudWatch automatically. For more information, see [What is Amazon CloudWatch?](#) and [Metrics retention](#) in the *Amazon CloudWatch User Guide*.

Topics

- [How to identify unused resources](#)
- [Identifying unused table resources](#)
- [Cleaning up unused table resources](#)

- [Identifying unused GSI resources](#)
- [Cleaning up unused GSI resources](#)
- [Cleaning up unused global tables](#)
- [Cleaning up unused backups or point-in-time recovery \(PITR\)](#)

How to identify unused resources

To identify unused tables or indexes, we'll look at the following CloudWatch metrics over a period of 30 days to understand if there are any active reads or writes on the table or any reads on the global secondary indexes (GSIs):

[ConsumedReadCapacityUnits](#)

The number of read capacity units consumed over the specified time period, so you can track how much consumed capacity you have used. You can retrieve the total consumed read capacity for a table and all of its global secondary indexes, or for a particular global secondary index.

[ConsumedWriteCapacityUnits](#)

The number of write capacity units consumed over the specified time period, so you can track how much consumed capacity you have used. You can retrieve the total consumed write capacity for a table and all of its global secondary indexes, or for a particular global secondary index.

Identifying unused table resources

Amazon CloudWatch is a monitoring and observability service which provides the DynamoDB table metrics you'll use to identify unused resources. CloudWatch metrics can be viewed through the AWS Management Console as well as through the AWS Command Line Interface.

AWS Command Line Interface

To view your tables metrics through the AWS Command Line Interface, you can use the following commands.

1. First, evaluate your table's reads:

```
aws cloudwatch get-metric-statistics --metric-name ConsumedReadCapacityUnits --start-time <start-time> --end-time <end-time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
```

```
dimensions Name=TableName,Value=<table-name>
```

To avoid falsely identifying a table as unused, evaluate metrics over a longer period. Choose an appropriate start-time and end-time range, such as **30 days**, and an appropriate period, such as **86400**.

In the returned data, any **Sum** above **0** indicates that the table you are evaluating received read traffic during that period.

The following result shows a table receiving read traffic in the evaluated period:

```
{
    "Timestamp": "2022-08-25T19:40:00Z",
    "Sum": 36023355.0,
    "Unit": "Count"
},
{
    "Timestamp": "2022-08-12T19:40:00Z",
    "Sum": 38025777.5,
    "Unit": "Count"
},
```

The following result shows a table not receiving read traffic in the evaluated period:

```
{
    "Timestamp": "2022-08-01T19:50:00Z",
    "Sum": 0.0,
    "Unit": "Count"
},
{
    "Timestamp": "2022-08-20T19:50:00Z",
    "Sum": 0.0,
    "Unit": "Count"
},
```

2. Next, evaluate your table's writes:

```
aws cloudwatch get-metric-statistics --metric-name ConsumedWriteCapacityUnits --start-time <start-time> --end-time <end-time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
```

To avoid falsely identifying a table as unused, you will want to evaluate metrics over a longer period. Choose an appropriate start-time and end-time range, such as **30 days**, and an appropriate period, such as **86400**.

In the returned data, any **Sum** above **0** indicates that the table you are evaluating received read traffic during that period.

The following result shows a table receiving write traffic in the evaluated period:

```
{  
    "Timestamp": "2022-08-19T20:15:00Z",  
    "Sum": 41014457.0,  
    "Unit": "Count"  
,  
{  
    "Timestamp": "2022-08-18T20:15:00Z",  
    "Sum": 40048531.0,  
    "Unit": "Count"  
,
```

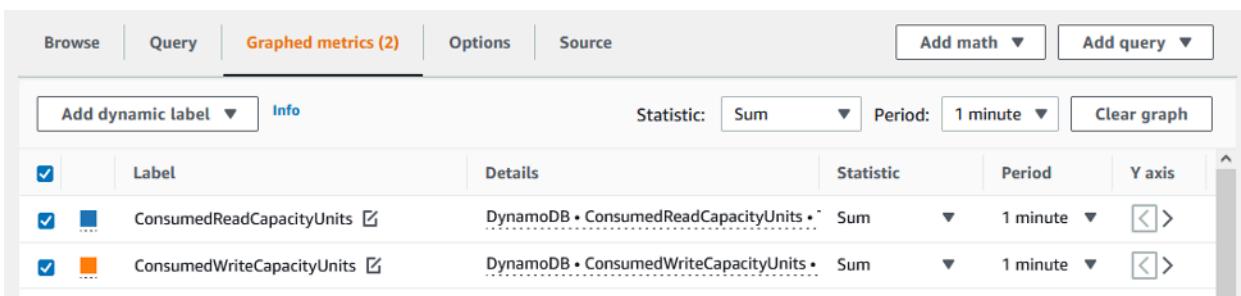
The following result shows a table not receiving write traffic in the evaluated period:

```
{  
    "Timestamp": "2022-07-31T20:15:00Z",  
    "Sum": 0.0,  
    "Unit": "Count"  
,  
{  
    "Timestamp": "2022-08-19T20:15:00Z",  
    "Sum": 0.0,  
    "Unit": "Count"  
,
```

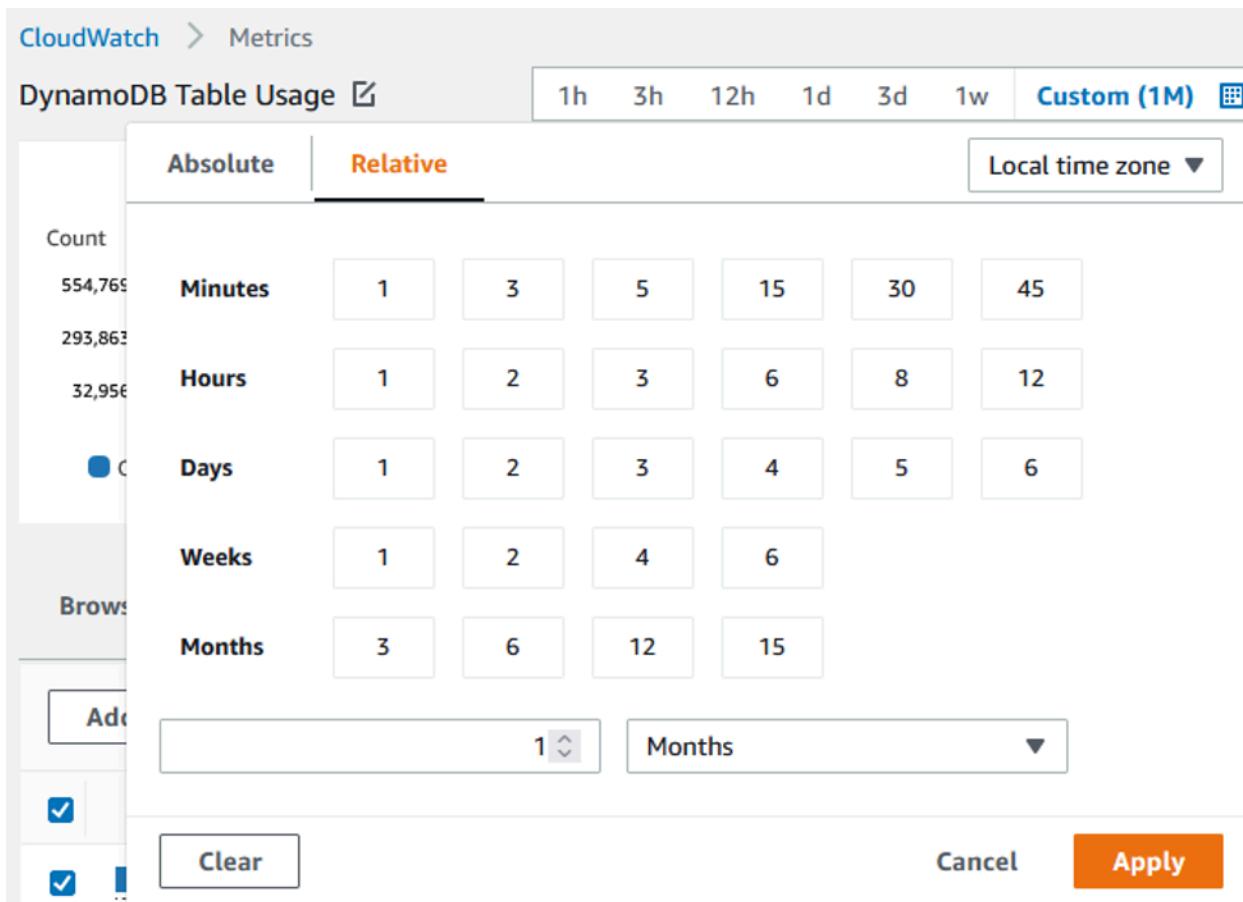
AWS Management Console

The following steps will allow you to evaluate your resources utilization through the AWS Management Console.

1. Log into the AWS console and navigate to the CloudWatch service page at <https://console.aws.amazon.com/cloudwatch/>. Select the appropriate AWS region in the top right of the console, if necessary.
2. On the left navigation bar, locate the Metrics section and select **All metrics**.
3. The action above will open a dashboard with two panels. In the top panel you will see currently graphed metrics. In the bottom you will select the metrics available to graph. Select DynamoDB in the bottom panel.
4. In the DynamoDB metrics selection panel select the **Table Metrics** category to show the metrics for your tables in the current region.
5. Identify your table name by scrolling down the menu, then select the metrics `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` for your table.
6. Select the **Graphed metrics (2)** tab and adjust the **Statistic** column to **Sum**.

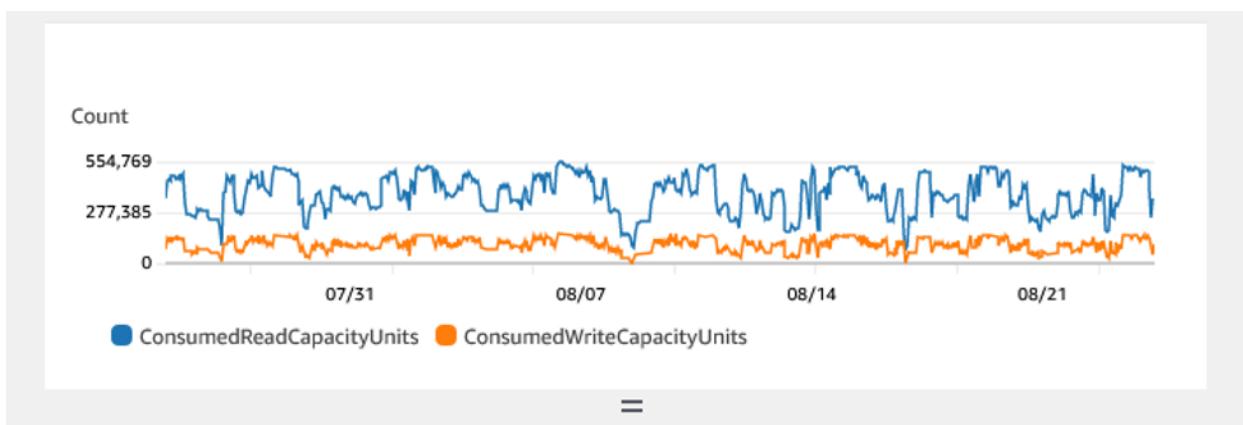


7. To avoid falsely identifying a table as unused, you'll want to evaluate metrics over a longer period. At the top of the graph panel choose an appropriate time frame, such as 1 month, to evaluate your table. Select **Custom**, select **1 Months** in the dropdowns, and choose **Apply**.

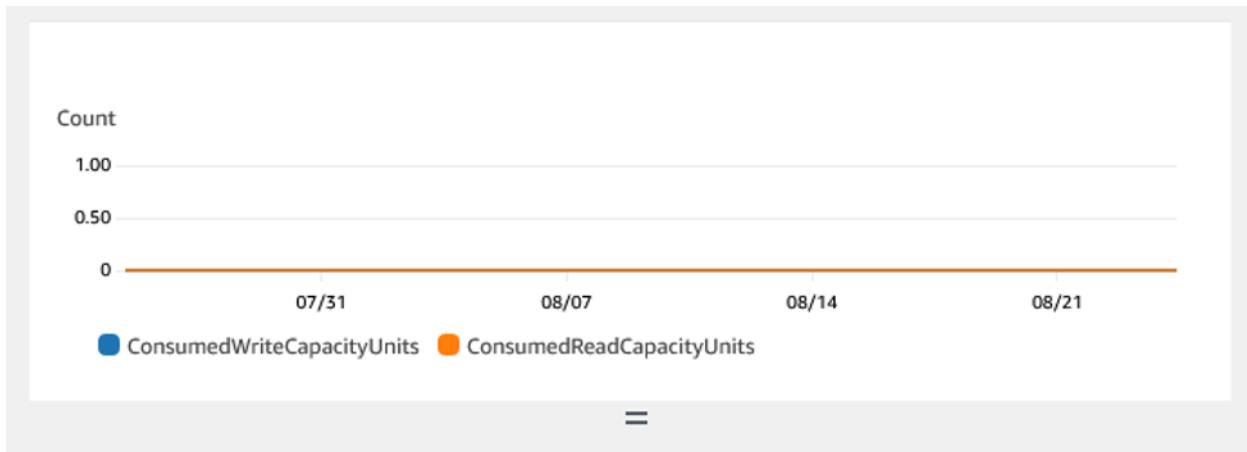


- Evaluate the graphed metrics for your table to determine if it is being used. Metrics that have gone above **0** indicate that a table has been used during the evaluated time period. A flat graph at **0** for both read and write indicates a table that is unused.

The following image shows a table with read traffic:



The following image shows a table without read traffic:



Cleaning up unused table resources

If you have identified unused table resources, you can reduce their ongoing costs in the following ways.

Note

If you have identified an unused table but would still like to keep it available in case it needs to be accessed in the future, consider switching it to on-demand mode. Otherwise, you can consider backing up and deleting the table entirely.

Capacity modes

DynamoDB charges for reading, writing, and storing data in your DynamoDB tables.

DynamoDB has [two capacity modes](#), which come with specific billing options for processing reads and writes on your tables: on-demand and provisioned. The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.

For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform. DynamoDB charges you for the reads and writes that your application performs on your tables in terms of read request units and write request units. If there is no activity on your table/index you do not pay for throughput but you'll still incur a storage charge.

Table class

DynamoDB also offers [two table classes](#) designed to help you optimize for cost. The DynamoDB Standard table class is the default and is recommended for most workloads. The DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) table class is optimized for tables where storage is the dominant cost.

If there is no activity on your table or index, storage is likely to be the dominant cost and changing table class will offer a significant savings.

Deleting tables

If you've discovered an unused table and would like to delete it, you may wish to make a backup or export of the data first.

Backups taken through AWS Backup can leverage cold storage tiering, further reducing costs. Refer to the [Using AWS Backup with DynamoDB](#) documentation for information on how enable backups through AWS Backup as well as the [Managing backup plans](#) documentation for information on how to use lifecycle to move your backup to cold storage.

Alternatively, you may choose to export your table's data to S3. To do so, refer to the [Export to Amazon S3](#) documentation. Once your data is exported, if you wish to leverage S3 Glacier Instant Retrieval, S3 Glacier Flexile Retrieval, or S3 Glacier Deep Archive to further reduce costs, see [Managing your storage lifecycle](#).

After your table has been backed up, you may choose to delete it either through the AWS Management Console or through the AWS Command Line Interface.

Identifying unused GSI resources

The steps for identifying an unused global secondary are similar to those for identifying an unused table. Since DynamoDB replicates items written to your base table into your GSI if they contain the attribute used as the GSI's partition key, an unused GSI is still likely to have ConsumedWriteCapacityUnits above 0 if its base table is in use. As a result, you'll be evaluating only the ConsumedReadCapacityUnits metric to determine if your GSI is unused.

To view your GSI metrics through the AWS AWS CLI, you can use the following commands to evaluate your tables reads:

```
aws cloudwatch get-metric-statistics --metric-name  
ConsumedReadCapacityUnits --start-time <start-time> --end-time <end-
```

```
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
Name=GlobalSecondaryIndexName,Value=<index-name>
```

To avoid falsely identifying a table as unused, you will want to evaluate metrics over a longer period. Choose an appropriate start-time and end-time range, such as 30 days, and an appropriate period, such as 86400.

In the returned data, any Sum above 0 indicates that the table you are evaluating received read traffic during that period.

The following result shows a GSI receiving read traffic in the evaluated period:

```
{
  "Timestamp": "2022-08-17T21:20:00Z",
  "Sum": 36319167.0,
  "Unit": "Count"
},
{
  "Timestamp": "2022-08-11T21:20:00Z",
  "Sum": 1869136.0,
  "Unit": "Count"
},
```

The following result shows a GSI receiving minimal read traffic in the evaluated period:

```
{
  "Timestamp": "2022-08-28T21:20:00Z",
  "Sum": 0.0,
  "Unit": "Count"
},
{
  "Timestamp": "2022-08-15T21:20:00Z",
  "Sum": 2.0,
  "Unit": "Count"
},
```

The following result shows a GSI receiving no read traffic in the evaluated period:

```
{
  "Timestamp": "2022-08-17T21:20:00Z",
```

```
        "Sum": 0.0,
        "Unit": "Count"
    },
{
    "Timestamp": "2022-08-11T21:20:00Z",
    "Sum": 0.0,
    "Unit": "Count"
},
```

Cleaning up unused GSI resources

If you've identified an unused GSI, you can choose to delete it. Since all data present in a GSI is also present in the base table, additional backup is not necessary before deleting a GSI. If in the future the GSI is once again needed, it may be added back to the table.

If you have identified an infrequently used GSI, you should consider design changes in your application that would allow you to delete it or reduce its cost. For example, while DynamoDB scans should be used sparingly because they can consume large amounts of system resources, they may be more cost effective than a GSI if the access pattern it supports is used very infrequently.

Additionally, if a GSI is required to support an infrequent access pattern consider projecting a more limited set of attributes. While this may require subsequent queries against the base table to support your infrequent access patterns, it can potentially offer a significant reduction in storage and write costs.

Cleaning up unused global tables

Amazon DynamoDB global tables provide a fully managed solution for deploying a multi-Region, multi-active database, without having to build and maintain your own replication solution.

Global tables are ideal for providing low-latency access to data close to users and as well as a secondary region for disaster recovery.

If the global tables option is enabled for a resource in an effort to provide low-latency access to data but is not part of your disaster recovery strategy, validate that both replicas are actively serving read traffic by evaluating their CloudWatch metrics. If one replica does not serve read traffic, it may be an unused resource.

If global tables are part of your disaster recovery strategy, one replica not receiving read traffic may be expected under an active/standby pattern.

Cleaning up unused backups or point-in-time recovery (PITR)

DynamoDB offers two styles of backup. Point-in-time recovery provides continuous backups for 35 days to help you protect against accidental writes or deletes while on-demand backup allows for snapshot creation which can be saved long term. Both types of backups have costs associated with them.

Refer to the documentation for [Using On-Demand backup and restore for DynamoDB](#) and [Point-in-time recovery for DynamoDB](#) to determine if your tables have backups enabled that may no longer be needed.

Evaluate your table usage patterns

This section provides an overview of how to evaluate if you are efficiently using your DynamoDB tables. There are certain usage patterns which are not optimal for DynamoDB, and they allow room for optimization from both a performance and cost perspective.

Topics

- [Perform fewer strongly-consistent read operations](#)
- [Perform fewer transactions for read operations](#)
- [Perform fewer scans](#)
- [Shorten attribute names](#)
- [Enable Time to Live \(TTL\)](#)
- [Replace global tables with cross-Region backups](#)

Perform fewer strongly-consistent read operations

DynamoDB allows you to configure [read consistency](#) on a per-request basis. Read requests are eventually consistent by default. Eventually consistent reads are charged at 0.5 RCU for upto 4 KB of data.

Most parts of distributed workloads are flexible and can tolerate eventual consistency. However, there can be access patterns requiring strongly consistent reads. Strongly consistent reads are charged at 1 RCU for upto 4 KB of data, essentially doubling your read costs. DynamoDB provides you with the flexibility to use both consistency models on the same table.

You can evaluate your workload and application code to confirm if strongly consistent reads are used only where required.

Perform fewer transactions for read operations

DynamoDB allows you to group certain actions in an all-or-nothing manner, which means you have the ability to execute ACID transactions with DynamoDB. However, as is the case with relational databases, it is not necessary to follow this approach for every action.

A [transactional read operation](#) of up to 4 KB consumes 2 RCUs as opposed to the default 0.5 RCUs for reading the same amount of data in an eventually consistent manner. The costs are doubled in write operations which means, a transactional write of up to 1 KB equates to 2 WCUs.

To determine if all operations on your tables are transactions, CloudWatch metrics for the table can be filtered down to the transaction APIs. If transaction APIs are the only graphs available under the `SuccessfulRequestLatency` metrics for the table, this would confirm that every operation is a transaction for this table. Alternatively, if the overall capacity utilization trend matches the transaction API trend, consider revisiting the application design as it seems dominated by transactional APIs.

Perform fewer scans

The extensive use of Scan operations generally stems from the need to run analytical queries on the DynamoDB data. Running frequent Scan operations on large table can be inefficient and expensive.

A better alternative is to use the [Export to S3](#) feature and choosing a point in time to export the table state to S3. AWS offers services like Athena which can then be used to run analytical queries on the data without consuming any capacity from the table.

The frequency for Scan operations can be determined using the `SampleCount` statistic under the `SuccessfulRequestLatency` metric for Scan. If Scan operations are indeed very frequent, the access patterns and data model should be re-evaluated.

Shorten attribute names

The total size of an item in DynamoDB is the sum of its attribute name lengths and values. Having long attribute names not only contributes towards storage costs, but it might also lead to higher RCU/WCU consumption. We recommend that you choose shorter attribute names rather than long ones. Having shorter attribute names can help limit the item size within the next 4KB/1KB boundary after which you would consume additional RCU/WCU to access data.

Enable Time to Live (TTL)

[Time to Live \(TTL\)](#) can identify items older than the expiry time that you have set on an item and remove them from the table. If your data grows over time and older data becomes irrelevant, enabling TTL on the table can help trim your data down and save on storage costs.

Another useful aspect of TTL is that the expired items occur on your DynamoDB streams, so rather than just removing the data from your data, it is possible to consume those items from the stream and archive them to a lower cost storage tier. Additionally, deleting items via TTL comes at no additional cost — it does not consume capacity, and there's no overhead of designing a clean up application.

Replace global tables with cross-Region backups

[Global tables](#) allow you to maintain multiple active replica tables in different Regions — they can all accept write operations and replicate data across each other. It is easy to set up replicas and the synchronization is managed for you. The replicas converge to a consistent state using a last writer wins strategy.

If you are using Global tables purely as a part of failover or disaster recovery (DR) strategy, you can replace them with a cross-Region backup copy for relatively lenient recovery point objectives and recovery time objective requirements. If you do not require fast local access and the high availability of five nines, maintaining a global table replica might not be the best approach for failover.

As an alternative, consider using AWS Backup to manage DynamoDB backups. You can schedule regular backups and copy them across Regions to meet DR requirements in a more cost-effective approach compared to using Global tables.

Evaluate your Streams usage

This section provides an overview of how to evaluate your DynamoDB Streams usage. There are certain usage patterns which are not optimal for DynamoDB, and they allow room for optimization from both a performance and cost perspective.

You have two native streaming integrations for streaming and event-driven use cases:

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis Data Streams](#)

This page will just focus on cost optimization strategies for these options. If you'd like to instead find out how to choose between the two options, see [Streaming options for change data capture](#).

Topics

- [Optimizing costs for DynamoDB Streams](#)
- [Optimizing costs for Kinesis Data Streams](#)
- [Cost optimization strategies for both types of Streams usage](#)

Optimizing costs for DynamoDB Streams

As mentioned in the [pricing page](#) for DynamoDB Streams, regardless of the table's throughput capacity mode, DynamoDB charges on the number of read requests made towards the table's DynamoDB Stream. Read requests made towards a DynamoDB Stream are different from the read requests made towards a DynamoDB table.

Each read request in terms of the stream is in the form of a GetRecords API call that can return up to 1000 records or 1 MB worth of records in the response, whichever is reached first. None of the [other DynamoDB Stream APIs](#) are charged and DynamoDB Streams are not charged for being idle. In other words, if no read requests are made to a DynamoDB Stream, no charges will be incurred for having a DynamoDB Stream enabled on a table.

Here are a few consumer applications for DynamoDB Streams:

- AWS Lambda function(s)
- Amazon Kinesis Data Streams-based applications
- Customer consumer applications built using an AWS SDK

Read requests made by AWS Lambda-based consumers of DynamoDB Streams are free, whereas calls made by consumers of any other kind are charged. Every month, the first 2,500,000 read requests made by non-Lambda consumers are also free of cost. This applies to all read requests made to any DynamoDB Streams in an AWS Account for each AWS Region.

Monitoring your DynamoDB Streams usage

DynamoDB Streams charges on the billing console are grouped together for all DynamoDB Streams across the AWS Region in an AWS Account. Currently, tagging DynamoDB Streams is not supported, so cost allocation tags cannot be used to identify granular costs for DynamoDB Streams. The volume of GetRecords calls can be obtained at the DynamoDB Stream level

to compute the charges per stream. The volume is represented by the DynamoDB Stream's CloudWatch metric `SuccessfulRequestLatency` and its `SampleCount` statistic. This metric will also include `GetRecords` calls made by global tables to perform on-going replication as well as calls made by AWS Lambda consumers, both of which are not charged. For information on other CloudWatch metrics published by DynamoDB Streams, see [DynamoDB Metrics and dimensions](#).

Using AWS Lambda as the consumer

Evaluate if using AWS Lambda functions as the consumers for DynamoDB Streams is feasible because that can eliminate costs associated with reading from the DynamoDB Stream. On the other hand, DynamoDB Streams Kinesis Adapter or SDK based consumer applications will be charged on the number of `GetRecords` calls they make towards the DynamoDB Stream.

Lambda function invocations will be charged based on standard Lambda pricing, however no charges will be incurred by DynamoDB Streams. Lambda will poll shards in your DynamoDB Stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

Tuning DynamoDB Streams Kinesis Adapter-based consumer applications

Since read requests made by non-Lambda based consumers are charged for DynamoDB Streams, it is important to find a balance between the near real-time requirement and the number of times the consumer application must poll the DynamoDB Stream.

The frequency of polling DynamoDB Streams using a DynamoDB Streams Kinesis Adapter based application is determined by the configured `idleTimeBetweenReadsInMillis` value. This parameter determines the amount of time in milliseconds that the consumer should wait before processing a shard in case the previous `GetRecords` call made to the same shard did not return any records. By default, this value for this parameter is 1000 ms. If near real-time processing is not required, this parameter could be increased to have the consumer application make fewer `GetRecords` calls and optimize on DynamoDB Streams calls.

Optimizing costs for Kinesis Data Streams

When a Kinesis Data Stream is set as the destination to deliver change data capture events for a DynamoDB table, the Kinesis Data Stream may need separate sizing management which will affect the overall costs. DynamoDB charges in terms of Change Data capture Units (CDUs) where each unit is a made of up a 1 KB DynamoDB item size attempted by the DynamoDB service to the destination Kinesis Data Stream.

In addition to charges by the DynamoDB service, standard Kinesis Data Stream charges will be incurred. As mentioned in the [pricing page](#), the service pricing differs based on the capacity mode - provisioned and on-demand, which are distinct from DynamoDB table capacity modes and are user-defined. At a high level, Kinesis Data Streams charges an hourly rate based on the capacity mode, as well as on data ingested into the stream by DynamoDB service. There may be additional charges like data retrieval (for on-demand mode), extended data retention (beyond default 24 hours), and enhanced fan-out consumer retrievals depending on the user configuration for the Kinesis Data Stream.

Monitoring your Kinesis Data Streams usage

Kinesis Data Streams for DynamoDB publishes metrics from DynamoDB in addition to standard Kinesis Data Stream CloudWatch Metrics. It may be possible that a Put attempt by the DynamoDB service is throttled by the Kinesis service because of insufficient Kinesis Data Streams capacity, or by dependent components like a AWS KMS service that may be configured to encrypt the Kinesis Data Stream data at rest.

To learn more about CloudWatch metrics published by DynamoDB service for the Kinesis Data Stream, see [Monitoring change data capture with Kinesis Data Streams](#). In order to avoid additional costs of service retries due to throttles, it is important to right size the Kinesis Data Stream in case of Provisioned Mode.

Choosing the right capacity mode for Kinesis Data Streams

Kinesis Data Streams are supported in two capacity modes – provisioned mode and on-demand mode.

- If the workload involving Kinesis Data Stream has predictable application traffic, traffic that is consistent or ramps gradually, or traffic that can be forecasted accurately, then Kinesis Data Streams' **provisioned mode** is suitable and will be more cost efficient
- If the workload is new, has unpredictable application traffic, or you prefer not to manage capacity, then Kinesis Data Streams' **on-demand mode** is suitable and will be more cost efficient

A best practice to optimize costs would be to evaluate if the DynamoDB table associated with the Kinesis Data Stream has a predictable traffic pattern that can leverage provisioned mode of Kinesis Data Streams. If the workload is new, you could use on-demand mode for the Kinesis Data Streams for a few initial weeks, review the CloudWatch metrics to understand traffic patterns, and then switch the same Stream to provisioned mode based on the nature of the workload. In the case of

provisioned mode, estimation on number shards can be made by following shard management considerations for Kinesis Data Streams.

Evaluate your consumer applications using Kinesis Data Streams for DynamoDB

Since Kinesis Data Streams don't charge on the number of GetRecords calls like DynamoDB Streams, consumer applications could make as many number of calls as possible, provided the frequency is under the throttling limits for GetRecords. In terms of on-demand mode for Kinesis Data Streams, data reads are charged on a per GB basis. For provisioned mode Kinesis Data Streams, reads are not charged if the data is less than 7 days old. In the case of Lambda functions as Kinesis Data Streams consumers, Lambda polls each shard in your Kinesis Stream for records at a base rate of once per second.

Cost optimization strategies for both types of Streams usage

Event filtering for AWS Lambda consumers

Lambda event filtering allows you to discard events based on a filter criteria from being available in the Lambda function invocation batch. This optimizes Lambda costs for processing or discarding unwanted stream records within the consumer function logic. To learn more about configuring event filtering and writing your filtering criteria, see [Lambda event filtering](#).

Tuning AWS Lambda consumers

Costs could be further be optimized by tuning Lambda configuration parameters like increasing the BatchSize to process more per invocation, enabling BisectBatchOnFunctionError to prevent processing duplicates (which incurs additional costs), and setting MaximumRetryAttempts to not run into too many retries. By default, failed consumer Lambda invocations are retried infinitely until the record expires from the stream, which is around 24 hours for DynamoDB Streams and configurable from 24 hours to up to 1 year for Kinesis Data Streams. Additional Lambda configuration options available including the ones mentioned above for DynamoDB Stream consumers are in the [AWS Lambda developer guide](#).

Evaluate your provisioned capacity for right-sized provisioning

This section provides an overview of how to evaluate if you have right-sized provisioning on your DynamoDB tables. As your workload evolves, you should modify your operational procedures appropriately, especially when your DynamoDB table is configured in provisioned mode and you have the risk to over-provision or under-provision your tables.

The procedures described below require statistical information that should be captured from the DynamoDB tables that are supporting your production application. To understand your application behavior, you should define a period of time that is significant enough to capture the data seasonality from your application. For example, if your application shows weekly patterns, using a three week period should give you enough room for analysing application throughput needs.

If you don't know where to start, use at least one month's worth of data usage for the calculations below.

While evaluating capacity, DynamoDB tables can configure **Read Capacity Units (RCUs)** and **Write Capacity Units (WCUs)** independently. If your tables have any Global Secondary Indexes (GSI) configured, you will need to specify the throughput that it will consume, which will be also independent from the RCUs and WCUs from the base table.

 **Note**

Local Secondary Indexes (LSI) consume capacity from the base table.

Topics

- [How to retrieve consumption metrics on your DynamoDB tables](#)
- [How to identify under-provisioned DynamoDB tables](#)
- [How to identify over-provisioned DynamoDB tables](#)

How to retrieve consumption metrics on your DynamoDB tables

To evaluate the table and GSI capacity, monitor the following CloudWatch metrics and select the appropriate dimension to retrieve either table or GSI information:

Read Capacity Units	Write Capacity Units
ConsumedReadCapacityUnits	ConsumedWriteCapacityUnits
ProvisionedReadCapacityUnits	ProvisionedWriteCapacityUnits
ReadThrottleEvents	WriteThrottleEvents

You can do this either through the AWS CLI or the AWS Management Console.

AWS CLI

Before we retrieve the table consumption metrics, we'll need to start by capturing some historical data points using the CloudWatch API.

Start by creating two files: `write-calc.json` and `read-calc.json`. These files will represent the calculations for a table or GSI. You'll need to update some of the fields, as indicated in the table below, to match your environment.

Field Name	Definition	Example
<code><table-name></code>	The name of the table that you will be analysing	SampleTable
<code><period></code>	The period of time that you will be using to evaluate the utilization target, based in seconds	For a 1-hour period you should specify: 3600
<code><start-time></code>	The beginning of your evaluation interval, specified in ISO8601 format	2022-02-21T23:00:00
<code><end-time></code>	The end of your evaluation interval, specified in ISO8601 format	2022-02-22T06:00:00

The write calculations file will retrieve the number of WCU provisioned and consumed in the time period for the date range specified. It will also generate a utilization percentage that will be used for analysis. The full content of the `write-calc.json` file should look like this:

```
{  
  "MetricDataQueries": [  
    {  
      "Id": "provisionedWCU",  
      "MetricStat": {  
        "Metric": {
```

```
"Namespace": "AWS/DynamoDB",
"MetricName": "ProvisionedWriteCapacityUnits",
"Dimensions": [
    {
        "Name": "TableName",
        "Value": "<table-name>"
    }
],
},
"Period": <period>,
"Stat": "Average"
},
"Label": "Provisioned",
"ReturnData": false
},
{
    "Id": "consumedWCU",
    "MetricStat": {
        "Metric": {
            "Namespace": "AWS/DynamoDB",
            "MetricName": "ConsumedWriteCapacityUnits",
            "Dimensions": [
                {
                    "Name": "TableName",
                    "Value": "<table-name>"""
                }
            ],
            "Period": <period>,
            "Stat": "Sum"
        },
        "Label": "",
        "ReturnData": false
    },
    {
        "Id": "m1",
        "Expression": "consumedWCU/PERIOD(consumedWCU)",
        "Label": "Consumed WCUs",
        "ReturnData": false
    },
    {
        "Id": "utilizationPercentage",
        "Expression": "100*(m1/provisionedWCU)",
        "Label": "Utilization Percentage",
    }
]
```

```
        "ReturnData": true
    }
],
"StartTime": "<start-time>",
"EndTime": "<end-time>",
"ScanBy": "TimestampDescending",
"MaxDatapoints": 24
}
```

The read calculations file uses a similar file. This file will retrieve how many RCUs were provisioned and consumed during the time period for the date range specified. The contents of the `read-calc.json` file should look like this:

```
{
    "MetricDataQueries": [
        {
            "Id": "provisionedRCU",
            "MetricStat": {
                "Metric": {
                    "Namespace": "AWS/DynamoDB",
                    "MetricName": "ProvisionedReadCapacityUnits",
                    "Dimensions": [
                        {
                            "Name": "TableName",
                            "Value": "<table-name>"
                        }
                    ]
                },
                "Period": <period>,
                "Stat": "Average"
            },
            "Label": "Provisioned",
            "ReturnData": false
        },
        {
            "Id": "consumedRCU",
            "MetricStat": {
                "Metric": {
                    "Namespace": "AWS/DynamoDB",
                    "MetricName": "ConsumedReadCapacityUnits",
                    "Dimensions": [
                        {
                            "Name": "TableName",
                            "Value": "<table-name>"
                        }
                    ]
                },
                "Period": <period>,
                "Stat": "Average"
            },
            "Label": "Consumed",
            "ReturnData": false
        }
    ]
}
```

```
        "Value": "<table-name>"  
    }  
]  
,  
"Period": <period>,  
"Stat": "Sum"  
,  
"Label": "",  
"ReturnData": false  
,  
{  
    "Id": "m1",  
    "Expression": "consumedRCU/PERIOD(consumedRCU)",  
    "Label": "Consumed RCUs",  
    "ReturnData": false  
,  
{  
    "Id": "utilizationPercentage",  
    "Expression": "100*(m1/provisionedRCU)",  
    "Label": "Utilization Percentage",  
    "ReturnData": true  
}  
,  
"StartTime": "<start-time>",  
"EndTime": "<end-time>",  
"ScanBy": "TimestampDescending",  
"MaxDatapoints": 24  
}
```

Once you've created the files, you can start retrieving utilization data.

1. To retrieve the write utilization data, issue the following command:

```
aws cloudwatch get-metric-data --cli-input-json file://write-calc.json
```

2. To retrieve the read utilization data, issue the following command:

```
aws cloudwatch get-metric-data --cli-input-json file://read-calc.json
```

The result for both queries will be a series of data points in JSON format that will be used for analysis. Your result will depend on the number of data points you specified, the period, and your own specific workload data. It could look something like this:

```
{  
    "MetricDataResults": [  
        {  
            "Id": "utilizationPercentage",  
            "Label": "Utilization Percentage",  
            "Timestamps": [  
                "2022-02-22T05:00:00+00:00",  
                "2022-02-22T04:00:00+00:00",  
                "2022-02-22T03:00:00+00:00",  
                "2022-02-22T02:00:00+00:00",  
                "2022-02-22T01:00:00+00:00",  
                "2022-02-22T00:00:00+00:00",  
                "2022-02-21T23:00:00+00:00"  
            ],  
            "Values": [  
                91.55364583333333,  
                55.066631944444445,  
                2.6114930555555556,  
                24.9496875,  
                40.94725694444445,  
                25.61819444444444,  
                0.0  
            ],  
            "StatusCode": "Complete"  
        }  
    ],  
    "Messages": []  
}
```

Note

If you specify a short period and a long time range, you might need to modify the `MaxDatapoints` which is by default set to 24 in the script. This represents one data point per hour and 24 per day.

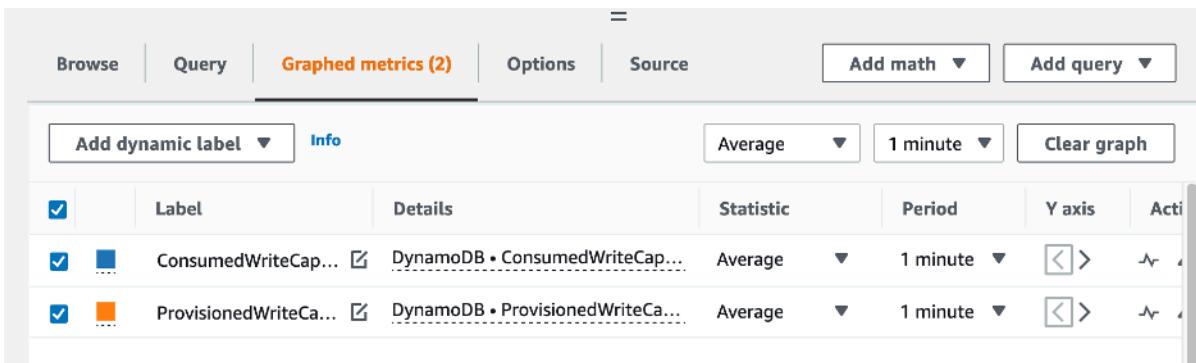
AWS Management Console

1. Log into the AWS Management Console and navigate to the CloudWatch service page at [Getting Started with the AWS Management Console](#). Select the appropriate AWS region if necessary.
2. Locate the Metrics section on the left navigation bar and select **All metrics**.
3. This will open a dashboard with two panels. The top panel will show you the graphic, and the bottom panel will have the metrics you want to graph. Select the DynamoDB panel.
4. Select the **Table Metrics** category from the sub panels. This will show you the tables in your current Region.
5. Identify your table name by scrolling down the menu and selecting the write operation metrics: ConsumedWriteCapacityUnits and ProvisionedWriteCapacityUnits

Note

This example talks about write operation metrics, but you can also use these steps to graph the read operation metrics.

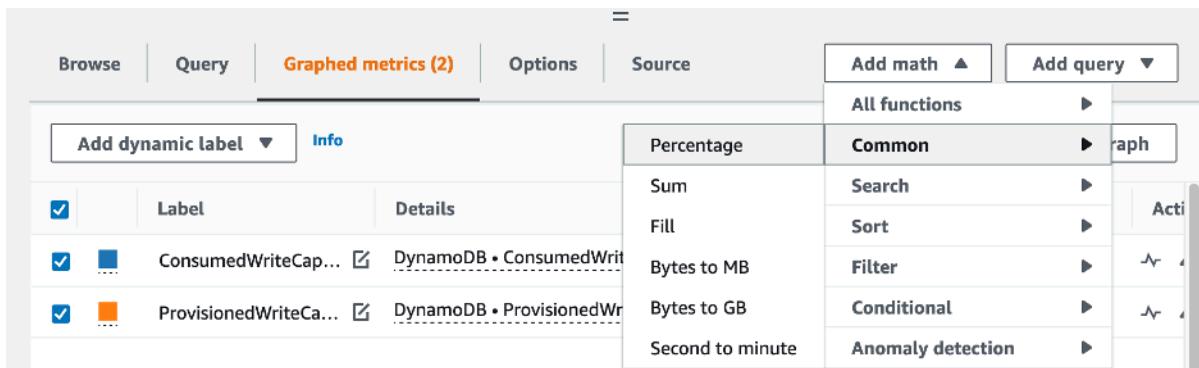
6. Select the **Graphed metrics (2)** tab to modify the formulas. By default CloudWatch will select the statistical function **Average** for the graphs.



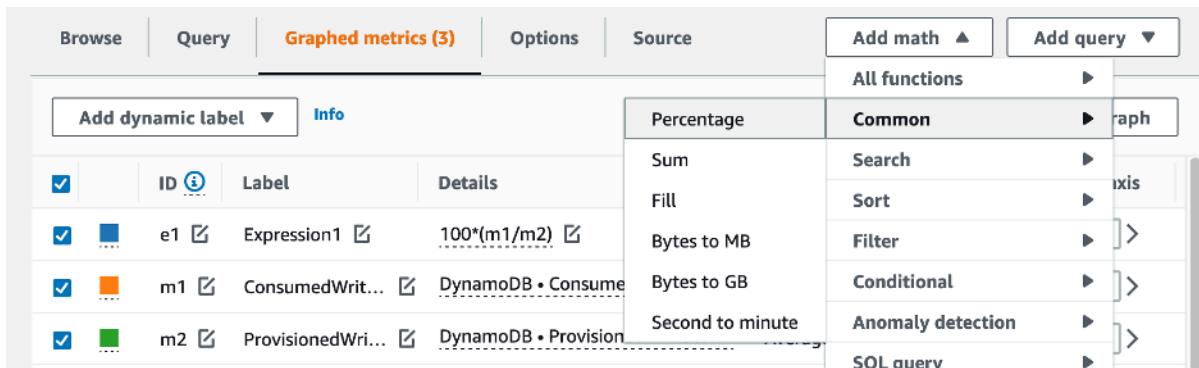
Label	Details	Statistic	Period	Y axis	Action
ConsumedWriteCap...	DynamoDB • ConsumedWriteCap...	Average	1 minute	< >	...
ProvisionedWriteCa...	DynamoDB • ProvisionedWriteCa...	Average	1 minute	< >	...

7. While having both graphed metrics selected (the checkbox on the left) select the menu **Add math**, followed by **Common**, and then select the **Percentage** function. Repeat the procedure twice.

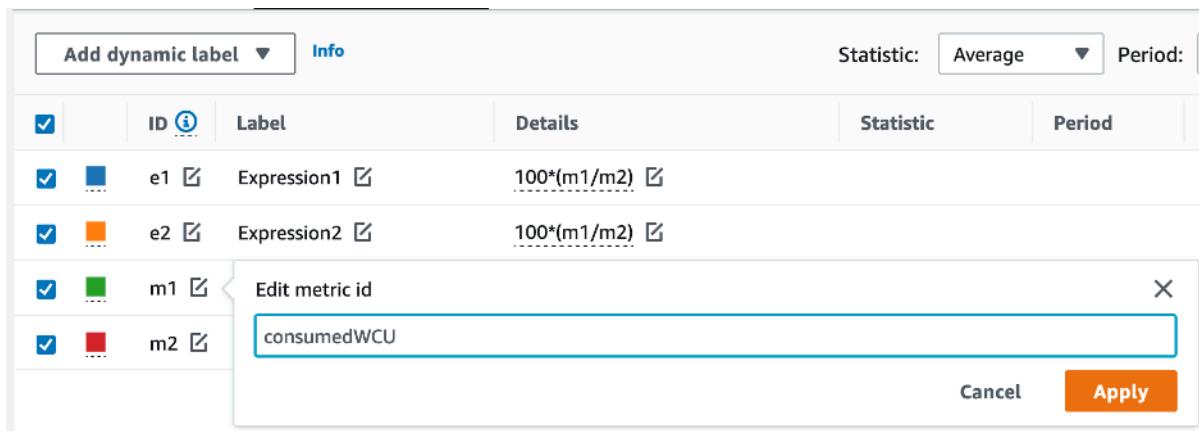
First time selecting the **Percentage** function:



Second time selecting the **Percentage** function:



- At this point you should have four metrics in the bottom menu. Let's work on the `ConsumedWriteCapacityUnits` calculation. To be consistent, we need to match the names for the ones we used in the AWS CLI section. Click on the **m1** ID and change this value to **consumedWCU**.



The screenshot shows the 'Graphed metrics' section of the CloudWatch Metrics console. A modal dialog is open over the list of metrics, titled 'Edit metric label'. Inside the dialog, the current label 'consumedWCU' is displayed in a text input field. Below the input field are two buttons: 'Cancel' and 'Apply'. At the top of the dialog, there are dropdown menus for 'Statistic' set to 'Average' and 'Period' set to '1 minute'. The main list of metrics shows four entries: 'e1' (Expression1), 'e2' (Expression2), 'consumedWCU' (ConsumedWriteCapacity), and 'm2' (ProvisionedWriteCapacity). The 'consumedWCU' entry has its details expanded, showing the metric name 'DynamoDB • ConsumedWriteCapacity' and the statistic 'Average'.

9. Change the statistic from **Average** to **Sum**. This action will automatically create another metric called **ANOMALY_DETECTION_BAND**. For the scope of this procedure, let's can ignore it by removing the checkbox on the newly generated **ad1 metric**.

This screenshot shows the same 'Graphed metrics' section after changing the statistic for 'consumedWCU' to 'Sum'. A dropdown menu for 'Statistic' is open, with 'Sum' selected. The other options in the menu are 'Average', 'Minimum', and 'Maximum'. The rest of the interface remains the same, with the list of metrics and their details visible.

This screenshot shows the 'Graphed metrics' section after completing the steps. The list now includes five metrics: 'e1' (Expression1), 'e2' (Expression2), 'consumedWCU' (ConsumedWriteCapacity, Statistic: Sum), 'ad1' (consumedWCU (e...), ANOMALY_DETECTION_BAND(...)), and 'm2' (ProvisionedWriteCapacity, Statistic: Average). The 'ad1' metric is listed with a note indicating it is a dynamic label. The 'Statistic' dropdown at the top is set to '(multiple)'. The 'Period' dropdown is set to '1 minute'.

10. Repeat step 8 to rename the **m2 ID** to **provisionedWCU**. Leave the statistic set to **Average**.

The screenshot shows the 'Graphed metrics' section of the CloudWatch Metrics console. A modal window titled 'Edit metric label' is open, allowing users to change the label for selected metrics. In the 'Label' field, the value 'provisionedWCU' is entered. The 'Apply' button is visible at the bottom of the modal.

ID	Label	Details	Statistic	Period	Y axis	Actions	
<input checked="" type="checkbox"/>	e1	Expression1	100*(consumedWCU/provision...				
<input checked="" type="checkbox"/>	e2	Expression2	100*(consumedWCU/provision...				
<input checked="" type="checkbox"/>	consumedWCU	DynamoDB + ConsumedWriteCapacity	Sum	1 minute			
<input type="checkbox"/>	ad1	consumedWCU (e...	ANOMALY_DETECTION_BAND(...				
<input checked="" type="checkbox"/>	provisionedWCU	DynamoDB + ProvisionedWriteCapacit...	Average	1 minute			

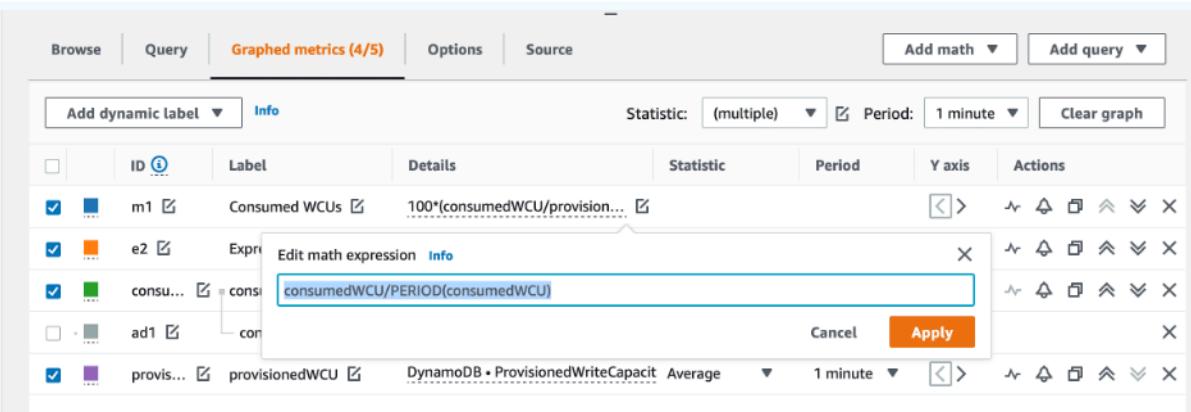
11. Select the **Expression1** label and update the value to **m1** and the label to **Consumed WCUs**.

The screenshot shows the 'Graphed metrics' section of the CloudWatch Metrics console. A modal window titled 'Edit metric label' is open, allowing users to change the label for selected metrics. In the 'Label' field, the value 'Consumed WCUs' is entered. The 'Apply' button is visible at the bottom of the modal. A tooltip 'Expression1' is shown near the 'Expression1' label in the list.

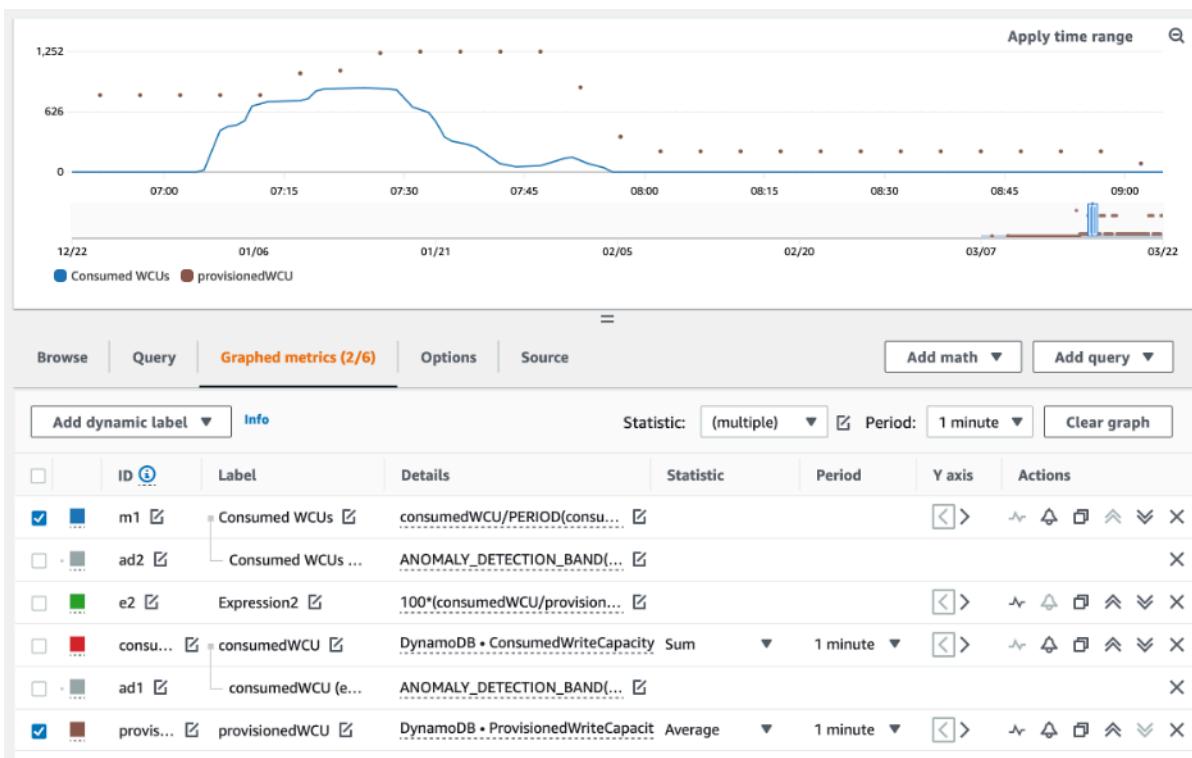
ID	Label	Details	Statistic	Period	Y axis	Actions
<input checked="" type="checkbox"/>	m1	Expression1	Consumed WCUs			
<input checked="" type="checkbox"/>	e2	Expression2				
<input checked="" type="checkbox"/>	consumedWCU	DynamoDB + ConsumedWriteCapacity	Sum	1 minute		
<input type="checkbox"/>	ad1	consumedWCU (e...	ANOMALY_DETECTION_BAND(...			
<input checked="" type="checkbox"/>	provisionedWCU	DynamoDB + ProvisionedWriteCapacit...	Average	1 minute		

Note

Make sure you have only selected **m1** (checkbox on the left) and **provisionedWCU** to properly visualize the data. Update the formula by clicking in **Details** and changing the formula to **consumedWCU/PERIOD(consumedWCU)**. This step might also generate another **ANOMALY_DETECTION_BAND** metric, but for the scope of this procedure we can ignore it.



12. You should have now have two graphics: one that indicates your provisioned WCUs on the table and another that indicates the consumed WCUs. The shape of the graphic might be different from the one below, but you can use it as reference:



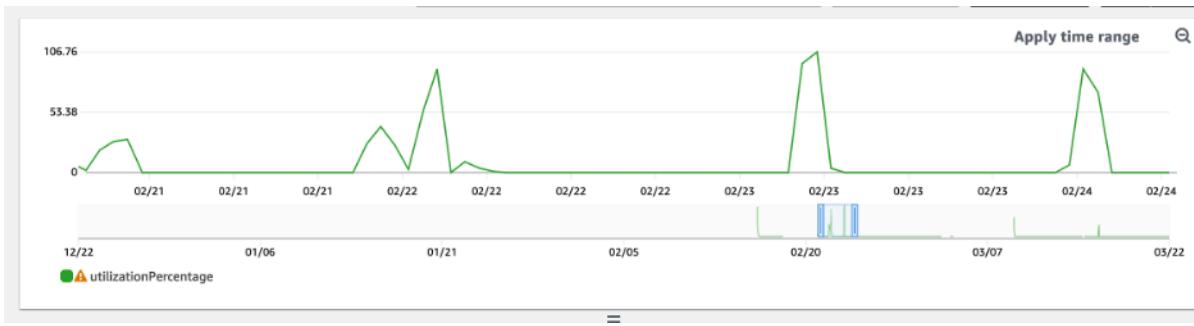
13. Update the percentage formula by selecting the Expression2 graphic (e2). Rename the labels and IDs to **utilizationPercentage**. Rename the formula to match **100*(m1/provisionedWCUs)**.

The screenshot shows the AWS CloudWatch Metrics console interface. At the top, there are tabs for 'Browse', 'Query', and 'Graphed metrics (2/6)', with 'Graphed metrics (2/6)' being the active tab. Below the tabs are buttons for 'Add math' and 'Add query'. The main area displays a table of metrics. One row for 'utilizationPercentage' is selected, and a modal dialog titled 'Edit metric label' is open over it, containing the text 'utilizationPercentage'. The bottom part of the interface shows the 'Edit math expression' dialog with the expression '100*(consumedWCU/provisionedWCU)' entered.

14. Remove the checkbox from all the metrics but **utilizationPercentage** to visualize your utilization patterns. The default interval is set to 1 minute, but feel free to modify it as you need.



Here is view of a longer period of time as well as a bigger period of 1 hour. You can see there are some intervals where the utilization was higher than 100%, but this particular workload has longer intervals with zero utilization.



At this point, you might have different results from the pictures in this example. It all depends on the data from your workload. Intervals with more than 100% utilization are prone to throttling events. DynamoDB offers [burst capacity](#), but as soon as the burst capacity is done anything above 100% will be throttled.

How to identify under-provisioned DynamoDB tables

For most workloads, a table is considered under-provisioned when it constantly consumes more than 80% of their provisioned capacity.

[Burst capacity](#) is a DynamoDB feature that allow customers to temporarily consume more RCUs/WCUs than originally provisioned (more than the per-second provisioned throughput that was defined in the table). The burst capacity was created to absorb sudden increases in traffic due to special events or usage spikes. This burst capacity doesn't last forever. As soon as the unused RCUs and WCUs are depleted, you will get throttled if you try to consume more capacity than provisioned. When your application traffic is getting close to the 80% utilization rate, your risk of throttling is significantly higher.

The 80% utilization rate rule varies from the seasonality of your data and your traffic growth. Consider the following scenarios:

- If your traffic has been **stable** at ~90% utilization rate for the last 12 months, your table has just the right capacity
- If your application traffic is **growing** at a rate of 8% monthly in less than 3 months, you will arrive at 100%
- If your application traffic is **growing** at a rate of 5% in a little more than 4 months, you will still arrive at 100%

The results from the queries above provide a picture of your utilization rate. Use them as a guide to further evaluate other metrics that can help you choose to increase your table capacity as required (for example: a monthly or weekly growth rate). Work with your operations team to define what is a good percentage for your workload and your tables.

There are special scenarios where the data is skewed when we analyse it on a daily or weekly basis. For example, with seasonal applications that have spikes in usage during working hours (but then drops to almost zero outside of working hours), you could benefit by [scheduling auto scaling](#) where you specify the hours of the day (and the days of the week) to increase the provisioned capacity and when to reduce it. Instead of aiming for higher capacity so you can cover the busy hours, you can also benefit from [DynamoDB table auto scaling](#) configurations if your seasonality is less pronounced.

 **Note**

When you create a DynamoDB auto scaling configuration for your base table, remember to include another configuration for any GSI that is associated with the table.

How to identify over-provisioned DynamoDB tables

The query results obtained from the scripts above provide the data points required to perform some initial analysis. If your data set presents values lower than 20% utilization for several intervals, your table might be over-provisioned. To further define if you need to reduce the number of WCUs and RCUS, you should revisit the other readings in the intervals.

When your tables contain several low usage intervals, you can really benefit from using auto scaling policies, either by scheduling auto scaling or just configuring the default auto scaling policies for the table that are based on utilization.

If you have a workload with low utilization to high throttle ratio (**Max(ThrottleEvents)**/**Min(ThrottleEvents)** in the interval), this could happen when you have a very spiky workload where traffic increases a lot during some days (or hours), but in general the traffic is consistently low. In these scenarios it might be beneficial to use [scheduled auto scaling](#).

The AWS [Well-Architected Framework](#) helps cloud architects build secure, high-performing, resilient, and efficient infrastructure for a variety of applications and workloads. Built around six pillars—operational excellence, security, reliability, performance efficiency, cost optimization, and

sustainability—AWS Well-Architected provides a consistent approach for customers and partners to evaluate architectures and implement scalable designs.

The AWS [Well-Architected Lenses](#) extend the guidance offered by AWS Well-Architected to specific industry and technology domains. The Amazon DynamoDB Well-Architected Lens focuses on DynamoDB workloads. It provides best practices, design principles and questions to assess and review a DynamoDB workload. Completing an Amazon DynamoDB Well-Architected Lens review will provide you with education and guidance around recommended design principles as it relates to each of the AWS Well-Architected pillars. This guidance is based on our experience working with customers across various industries, segments, sizes and geographies.

As a direct outcome of the Well-Architected Lens review, you will receive a summary of actionable recommendations to optimize and improve your DynamoDB workload.

Conducting the Amazon DynamoDB Well-Architected Lens review

The DynamoDB Well-Architected Lens review is usually performed by an AWS Solutions Architect together with the customer, but can also be performed by the customer as a self-service. While we recommend reviewing all six of the Well-Architected Pillars as part of the Amazon DynamoDB Well-Architected Lens, you can also decide to prioritize your focus on one or more pillars first.

Additional information and instructions for conducting an Amazon DynamoDB Well-Architected Lens review are available in [this video](#) and the [DynamoDB Well-Architected Lens GitHub page](#).

The pillars of the Amazon DynamoDB Well-Architected Lens

The Amazon DynamoDB Well-Architected Lens is built around six pillars:

Performance efficiency pillar

The performance efficiency pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve.

The primary DynamoDB design principles for this pillar revolve around [modeling the data](#), [choosing partition keys](#) and [sort keys](#), and [defining secondary indexes](#) based on the application access patterns. Additional considerations include choosing the optimal throughput mode for the workload, AWS SDK tuning and, when appropriate, using an optimal caching strategy. To learn more about these design principles, watch this [deep dive video](#) about the performance efficiency pillar of the DynamoDB Well-Architected Lens.

Cost optimization pillar

The cost optimization pillar focuses on avoiding unnecessary costs.

Key topics include understanding and controlling where money is being spent, selecting the most appropriate and right number of resource types, analyzing spend over time, designing your data models to optimize the cost for application-specific access patterns, and scaling to meet business needs without overspending.

The key cost optimization design principles for DynamoDB revolve around choosing the most appropriate capacity mode and table class for your tables and avoiding over-provisioning capacity by either using the on-demand capacity mode, or provisioned capacity mode with autoscaling. Additional considerations include efficient data modeling and querying to reduce the amount of consumed capacity, reserving portions of the consumed capacity at discounted price, minimizing item size, identifying and removing unused resources and using [TTL](#) to automatically delete aged-out data at no cost. To learn more about these design principles, watch this [deep dive video](#) about the cost optimization pillar of the DynamoDB Well-Architected Lens.

See [Cost optimization](#) for additional information on cost optimization best practices for DynamoDB.

Operational excellence pillar

The operational excellence pillar focuses on running and monitoring systems to deliver business value, and continually improving processes and procedures. Key topics include automating changes, responding to events, and defining standards to manage daily operations.

The main operational excellence design principles for DynamoDB include monitoring DynamoDB metrics through Amazon CloudWatch and AWS Config and automatically alert and remediate when predefined thresholds are breached, or non compliant rules are detected. Additional considerations are defining DynamoDB resources via infrastructure as a code and leveraging tags for better organization, identification and cost accounting of your DynamoDB resources. To learn more about these design principles, watch this [deep dive video](#) about the operational excellence pillar of the DynamoDB Well-Architected Lens.

Reliability pillar

The reliability pillar focuses on ensuring a workload performs its intended function correctly and consistently when it's expected to. A resilient workload quickly recovers from failures to meet

business and customer demand. Key topics include distributed system design, recovery planning, and how to handle change.

The essential reliability design principles for DynamoDB revolve around choosing the backup strategy and retention based on your RPO and RTO requirements, using DynamoDB global tables for multi-regional workloads, or cross-region disaster recovery scenarios with low RTO, implementing retry logic with exponential backoff in the application by configuring and using these capabilities in the AWS SDK, and monitoring DynamoDB metrics through Amazon CloudWatch and automatically alerting and remediating when predefined thresholds are breached. To learn more about these design principles, watch this [deep dive video](#) about the reliability pillar of the DynamoDB Well-Architected Lens.

Security pillar

The security pillar focuses on protecting information and systems. Key topics include confidentiality and integrity of data, identifying and managing who can do what with privilege management, protecting systems, and establishing controls to detect security events.

The main security design principles for DynamoDB are encrypting data in transit with HTTPS, choosing the type of keys for data at rest encryption and defining the IAM roles and policies to authenticate, authorize and provide fine grain access to DynamoDB resources. Additional considerations include auditing DynamoDB control plane and data plane operations through AWS CloudTrail. To learn more about these design principles, watch this [deep dive video](#) about the security pillar of the DynamoDB Well-Architected Lens.

See [Security](#) for additional information on security for DynamoDB.

Sustainability pillar

The sustainability pillar focuses on minimizing the environmental impacts of running cloud workloads. Key topics include a shared responsibility model for sustainability, understanding impact, and maximizing utilization to minimize required resources and reduce downstream impacts.

The main sustainability design principles for DynamoDB include identifying and removing unused DynamoDB resources, avoiding over-provisioning though the usage of on-demand capacity mode or provisioned capacity-mode with autoscaling, efficient querying to reduce the amount of capacity being consumed and reduction of the storage footprint by compressing data and by deleting aged-out data through the use of TTL. To learn more about these design principles, watch this [deep dive video](#) about the sustainability pillar of the DynamoDB Well-Architected Lens.

Best practices for designing and using partition keys effectively

The primary key that uniquely identifies each item in an Amazon DynamoDB table can be simple (a partition key only) or composite (a partition key combined with a sort key).

Generally speaking, you should design your application for uniform activity across all logical partition keys in the table and its secondary indexes. You can determine the access patterns that your application requires, and read and write units that each table and secondary index requires.

By default, every partition in the table will strive to deliver the full capacity of 3,000 RCU and 1,000 WCU. The total throughput across all partitions in the table may be constrained by the provisioned throughput in provisioned mode, or by the table level throughput limit in on-demand mode. See [Service Quotas](#) for more information.

Topics

- [Using burst capacity effectively](#)
- [Understanding DynamoDB adaptive capacity](#)
- [Designing partition keys to distribute your workload](#)
- [Using write sharding to distribute workloads evenly](#)
- [Distributing write activity efficiently during data upload](#)

Using burst capacity effectively

DynamoDB provides some flexibility for your throughput provisioning with *burst capacity*.

Whenever you're not fully using your available throughput, DynamoDB reserves a portion of that unused capacity for later *bursts* of throughput to handle usage spikes.

DynamoDB currently retains up to 5 minutes (300 seconds) of unused read and write capacity. During an occasional burst of read or write activity, these extra capacity units can be consumed quickly—even faster than the per-second provisioned throughput capacity that you've defined for your table.

DynamoDB can also consume burst capacity for background maintenance and other tasks without prior notice.

Note that these burst capacity details might change in the future.

Understanding DynamoDB adaptive capacity

Adaptive capacity is a feature that enables DynamoDB to run imbalanced workloads indefinitely. It minimizes throttling due to throughput exceptions. It also helps you reduce costs by enabling you to provision only the throughput capacity that you need.

Adaptive capacity is enabled automatically for every DynamoDB table, at no additional cost. You don't need to explicitly enable or disable it.

Topics

- [Boost throughput capacity to high-traffic partitions](#)
- [Isolate frequently accessed items](#)

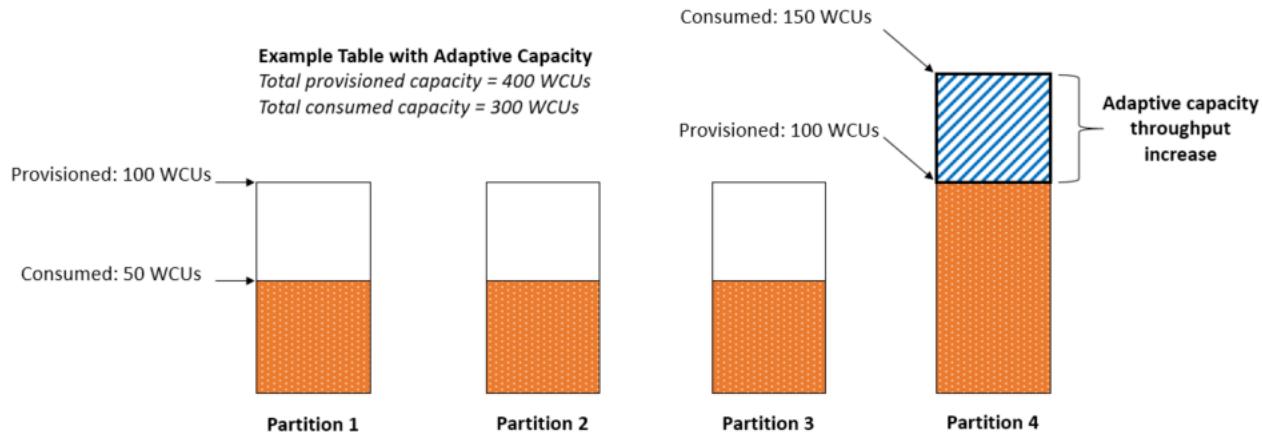
Boost throughput capacity to high-traffic partitions

It's not always possible to distribute read and write activity evenly. When data access is imbalanced, a "hot" partition can receive a higher volume of read and write traffic compared to other partitions. As read and write operations on a partition are managed independently, throttling will occur if a single partition receives more than 3000 read operation or more than 1000 write operations.

To better accommodate uneven access patterns, DynamoDB adaptive capacity enables your application to continue reading and writing to hot partitions without being throttled, provided that traffic does not exceed your table's total provisioned capacity or the partition maximum capacity. Adaptive capacity works by automatically and instantly increasing throughput capacity for partitions that receive more traffic.

The following diagram illustrates how adaptive capacity works. The example table is provisioned with 400 WCUs evenly shared across four partitions, allowing each partition to sustain up to 100 WCUs per second. Partitions 1, 2, and 3 each receives write traffic of 50 WCU/sec. Partition 4 receives 150 WCU/sec. This hot partition can accept write traffic while it still has unused burst capacity, but eventually it throttles traffic that exceeds 100 WCU/sec.

DynamoDB adaptive capacity responds by increasing partition 4's capacity so that it can sustain the higher workload of 150 WCU/sec without being throttled.



Isolate frequently accessed items

If your application drives disproportionately high traffic to one or more items, adaptive capacity rebalances your partitions such that frequently accessed items don't reside on the same partition. This isolation of frequently accessed items reduces the likelihood of request throttling due to your workload exceeding the throughput quota on a single partition. You can also break up an item collection into segments by sort key, as long as the item collection isn't traffic that is tracked by a monotonic increase or decrease of the sort key.

If your application drives consistently high traffic to a single item, adaptive capacity might rebalance your data so that a partition contains only that single, frequently accessed item. In this case, DynamoDB can deliver throughput up to the partition maximum of 3,000 RCU and 1,000 WCUs to that single item's primary key. Adaptive capacity will not split item collections across multiple partitions of the table when there is a [local secondary index](#) on the table.

Designing partition keys to distribute your workload

The partition key portion of a table's primary key determines the logical partitions in which a table's data is stored. This in turn affects the underlying physical partitions. A partition key design that doesn't distribute I/O requests effectively can create "hot" partitions that result in throttling and use your provisioned I/O capacity inefficiently.

The optimal usage of a table's provisioned throughput depends not only on the workload patterns of individual items, but also on the partition key design. This doesn't mean that you must access all partition key values to achieve an efficient throughput level, or even that the percentage of accessed partition key values must be high. It does mean that the more distinct partition key values that your workload accesses, the more those requests will be spread across the partitioned space.

In general, you will use your provisioned throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values increases.

The following is a comparison of the provisioned throughput efficiency of some common partition key schemas.

Partition key value	Uniformity
User ID, where the application has many users.	Good
Status code, where there are only a few possible status codes.	Bad
Item creation date, rounded to the nearest time period (for example, day, hour, or minute).	Bad
Device ID, where each device accesses data at relatively similar intervals.	Good
Device ID, where even if there are many devices being tracked, one is by far more popular than all the others.	Bad

If a single table has only a small number of partition key values, consider distributing your write operations across more distinct partition key values. In other words, structure the primary key elements to avoid one "hot" (heavily requested) partition key value that slows overall performance.

For example, consider a table with a composite primary key. The partition key represents the item's creation date, rounded to the nearest day. The sort key is an item identifier. On a given day, say 2014-07-09, **all** of the new items are written to that single partition key value (and corresponding physical partition).

If the table fits entirely into a single partition (considering growth of your data over time), and if your application's read and write throughput requirements don't exceed the read and write capabilities of a single partition, your application won't encounter any unexpected throttling as a result of partitioning.

To use NoSQL Workbench for DynamoDB to help visualize your partition key design, see [Building data models with NoSQL Workbench](#).

Using write sharding to distribute workloads evenly

One way to better distribute writes across a partition key space in Amazon DynamoDB is to expand the space. You can do this in several different ways. You can add a random number to the partition key values to distribute the items among partitions. Or you can use a number that is calculated based on something that you're querying on.

Sharding using random suffixes

One strategy for distributing loads more evenly across a partition key space is to add a random number to the end of the partition key values. Then you randomize the writes across the larger space.

For example, for a partition key that represents today's date, you might choose a random number between 1 and 200 and concatenate it as a suffix to the date. This yields partition key values like 2014-07-09.1, 2014-07-09.2, and so on, through 2014-07-09.200. Because you are randomizing the partition key, the writes to the table on each day are spread evenly across multiple partitions. This results in better parallelism and higher overall throughput.

However, to read all the items for a given day, you would have to query the items for all the suffixes and then merge the results. For example, you would first issue a Query request for the partition key value 2014-07-09.1. Then issue another Query for 2014-07-09.2, and so on, through 2014-07-09.200. Finally, your application would have to merge the results from all those Query requests.

Sharding using calculated suffixes

A randomizing strategy can greatly improve write throughput. But it's difficult to read a specific item because you don't know which suffix value was used when writing the item. To make it easier to read individual items, you can use a different strategy. Instead of using a random number to distribute the items among partitions, use a number that you can calculate based upon something that you want to query on.

Consider the previous example, in which a table uses today's date in the partition key. Now suppose that each item has an accessible OrderId attribute, and that you most often need to find items by order ID in addition to date. Before your application writes the item to the table, it could calculate a hash suffix based on the order ID and append it to the partition key date. The calculation might

generate a number between 1 and 200 that is fairly evenly distributed, similar to what the random strategy produces.

A simple calculation would likely suffice, such as the product of the UTF-8 code point values for the characters in the order ID, modulo 200, + 1. The partition key value would then be the date concatenated with the calculation result.

With this strategy, the writes are spread evenly across the partition key values, and thus across the physical partitions. You can easily perform a `GetItem` operation for a particular item and date because you can calculate the partition key value for a specific `OrderId` value.

To read all the items for a given day, you still must `Query` each of the `2014-07-09.N` keys (where `N` is 1–200), and your application then has to merge all the results. The benefit is that you avoid having a single "hot" partition key value taking all of the workload.

 **Note**

For a more efficient strategy specifically designed to handle high-volume time series data, see [Time series data](#).

Distributing write activity efficiently during data upload

Typically, when you load data from other data sources, Amazon DynamoDB partitions your table data on multiple servers. You get better performance if you upload data to all the allocated servers simultaneously.

For example, suppose that you want to upload user messages to a DynamoDB table that uses a composite primary key with `UserID` as the partition key and `MessageID` as the sort key.

When you upload the data, one approach you can take is to upload all message items for each user, one user after another:

UserID	MessageID
U1	1
U1	2
U1	...

UserID	MessageID
U1	... up to 100
U2	1
U2	2
U2	...
U2	... up to 200

The problem in this case is that you are not distributing your write requests to DynamoDB across your partition key values. You are taking one partition key value at a time and uploading all of its items before going to the next partition key value and doing the same.

Behind the scenes, DynamoDB is partitioning the data in your table across multiple servers. To fully use all the throughput capacity that is provisioned for the table, you must distribute your workload across your partition key values. By directing an uneven amount of upload work toward items that all have the same partition key value, you are not fully using all the resources that DynamoDB has provisioned for your table.

You can distribute your upload work by using the sort key to load one item from each partition key value, then another item from each partition key value, and so on:

UserID	MessageID
U1	1
U2	1
U3	1
...	...
U1	2
U2	2
U3	2

User ID	Message ID
...	...

Every upload in this sequence uses a different partition key value, keeping more DynamoDB servers busy simultaneously and improving your throughput performance.

Best practices for using sort keys to organize data

In an Amazon DynamoDB table, the primary key that uniquely identifies each item in the table can be composed of a partition key and a sort key.

Well-designed sort keys have two key benefits:

- They gather related information together in one place where it can be queried efficiently. Careful design of the sort key lets you retrieve commonly needed groups of related items using range queries with operators such as `begins_with`, `between`, `>`, `<`, and so on.
- Composite sort keys let you define hierarchical (one-to-many) relationships in your data that you can query at any level of the hierarchy.

For example, in a table listing geographical locations, you might structure the sort key as follows.

```
[country]#[region]#[state]#[county]#[city]#[neighborhood]
```

This would let you make efficient range queries for a list of locations at any one of these levels of aggregation, from country, to a neighborhood, and everything in between.

Using sort keys for version control

Many applications need to maintain a history of item-level revisions for audit or compliance purposes and to be able to retrieve the most recent version easily. There is an effective design pattern that accomplishes this using sort key prefixes:

- For each new item, create two copies of the item: One copy should have a version-number prefix of zero (such as `v0_`) at the beginning of the sort key, and one should have a version-number prefix of one (such as `v1_`).

- Every time the item is updated, use the next higher version-prefix in the sort key of the updated version, and copy the updated contents into the item with version-prefix zero. This means that the latest version of any item can be located easily using the zero prefix.

For example, a parts manufacturer might use a schema like the one illustrated below.

Primary Key		Data-Item Attributes...							
Partition Key	Sort Key (varies)	Attribute 1		Attribute 2		Attribute 3		Attribute 4	...
Equipment_1	Details	Name:	Biphasic Cardiometer <i>(equipment name)</i>	Factory_ID:	S14_Tukwilla <i>(factory where manufactured)</i>	Line_ID	R_7 <i>(assembly-line ID)</i>		
	v0_Audit	Auditor:	Padma <i>(name of the auditor)</i>	Latest:	3 <i>(most recent audit version)</i>	Time:	2018-04-15T11:00 <i>(audit date and time)</i>	Result	Passed <i>(audit result)</i>
	v1_Audit	Auditor:	Rick <i>(name of the auditor)</i>	Time:	2018-03-14T11:00 <i>(audit date and time)</i>	Result	Open <i>(audit result)</i>	Report:	0943922EKG14 <i>(detailed problem report in S3)</i>
	v2_Audit	Auditor:	George <i>(name of the auditor)</i>	Time:	2018-03-18T11:00 <i>(audit date and time)</i>	Result	Open <i>(audit result)</i>	Report:	0943923EKG15 <i>(detailed problem report in S3)</i>
	v3_Audit	Auditor:	Padma <i>(name of the auditor)</i>	Time:	2018-04-15T11:00 <i>(audit date and time)</i>	Result	Passed <i>(audit result)</i>	Report:	x792 <i>(pass confirmation report)</i>

The Equipment_1 item goes through a sequence of audits by various auditors. The results of each new audit are captured in a new item in the table, starting with version number one, and then incrementing the number for each successive revision.

When each new revision is added, the application layer replaces the contents of the zero-version item (having sort key equal to v0_Audit) with the contents of the new revision.

Whenever the application needs to retrieve for the most recent audit status, it can query for the sort key prefix of v0_.

If the application needs to retrieve the entire revision history, it can query all the items under the item's partition key and filter out the v0_ item.

This design also works for audits across multiple parts of a piece of equipment, if you include the individual part-IDs in the sort key after the sort key prefix.

Best practices for using secondary indexes in DynamoDB

Secondary indexes are often essential to support the query patterns that your application requires. At the same time, overusing secondary indexes or using them inefficiently can add cost and reduce performance unnecessarily.

Contents

- [General guidelines for secondary indexes in DynamoDB](#)
 - [Use indexes efficiently](#)
 - [Choose projections carefully](#)
 - [Optimize frequent queries to avoid fetches](#)
 - [Be aware of item-collection size limits when creating local secondary indexes](#)
- [Take advantage of sparse indexes](#)
 - [Examples of sparse indexes in DynamoDB](#)
- [Using Global Secondary Indexes for materialized aggregation queries](#)
- [Overloading Global Secondary Indexes](#)
- [Using Global Secondary Index write sharding for selective table queries](#)
- [Using Global Secondary Indexes to create an eventually consistent replica](#)

General guidelines for secondary indexes in DynamoDB

Amazon DynamoDB supports two types of secondary indexes:

- **Global secondary index (GSI)**—An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary index has no size limitations and has its own provisioned throughput settings for read and write activity that are separate from those of the table.
- **Local secondary index (LSI)**—An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value. As a result, the total size of indexed items for any one partition key value can't exceed 10 GB. Also, a local secondary index shares provisioned throughput settings for read and write activity with the table it is indexing.

Each table in DynamoDB can have up to 20 global secondary indexes (default quota) and 5 local secondary indexes.

Global secondary indexes are often more useful than local secondary indexes. Determining which type of index to use will also depend on your application's requirements. For a comparison of

global secondary indexes and local secondary indexes, and more information on how to choose between them, see [the section called "Working with indexes"](#).

The following are some general principles and design patterns to keep in mind when creating indexes in DynamoDB:

Topics

- [Use indexes efficiently](#)
- [Choose projections carefully](#)
- [Optimize frequent queries to avoid fetches](#)
- [Be aware of item-collection size limits when creating local secondary indexes](#)

Use indexes efficiently

Keep the number of indexes to a minimum. Don't create secondary indexes on attributes that you don't query often. Indexes that are seldom used contribute to increased storage and I/O costs without improving application performance.

Choose projections carefully

Because secondary indexes consume storage and provisioned throughput, you should keep the size of the index as small as possible. Also, the smaller the index, the greater the performance advantage compared to querying the full table. If your queries usually return only a small subset of attributes, and the total size of those attributes is much smaller than the whole item, project only the attributes that you regularly request.

If you expect a lot of write activity on a table compared to reads, follow these best practices:

- Consider projecting fewer attributes to minimize the size of items written to the index. However, this only applies if the size of projected attributes would otherwise be larger than a single write capacity unit (1 KB). For example, if the size of an index entry is only 200 bytes, DynamoDB rounds this up to 1 KB. In other words, as long as the index items are small, you can project more attributes at no extra cost.
- Avoid projecting attributes that you know will rarely be needed in queries. Every time you update an attribute that is projected in an index, you incur the extra cost of updating the index as well. You can still retrieve non-projected attributes in a Query at a higher provisioned throughput cost, but the query cost may be significantly lower than the cost of updating the index frequently.

- Specify ALL only if you want your queries to return the entire table item sorted by a different sort key. Projecting all attributes eliminates the need for table fetches, but in most cases, it doubles your costs for storage and write activity.

Balance the need to keep your indexes as small as possible against the need to keep fetches to a minimum, as explained in the next section.

Optimize frequent queries to avoid fetches

To get the fastest queries with the lowest possible latency, project all the attributes that you expect those queries to return. In particular, if you query a local secondary index for attributes that are not projected, DynamoDB automatically fetches those attributes from the table, which requires reading the entire item from the table. This introduces latency and additional I/O operations that you can avoid.

Keep in mind that "occasional" queries can often turn into "essential" queries. If there are attributes that you don't intend to project because you anticipate querying them only occasionally, consider whether circumstances might change and you might regret not projecting those attributes after all.

For more information about table fetches, see [Provisioned throughput considerations for Local Secondary Indexes](#).

Be aware of item-collection size limits when creating local secondary indexes

An *item collection* is all the items in a table and its local secondary indexes that have the same partition key. No item collection can exceed 10 GB, so it's possible to run out of space for a particular partition key value.

When you add or update a table item, DynamoDB updates all local secondary indexes that are affected. If the indexed attributes are defined in the table, the local secondary indexes grow too.

When you create a local secondary index, think about how much data will be written to it, and how many of those data items will have the same partition key value. If you expect that the sum of table and index items for a particular partition key value might exceed 10 GB, consider whether you should avoid creating the index.

If you can't avoid creating the local secondary index, you must anticipate the item collection size limit and take action before you exceed it. As a best practice, you should utilize the [ReturnItemCollectionMetrics](#) parameter when writing items to monitor and alert on item

collection sizes that approach the 10GB size limit. Exceeding the maximum item collection size will result in failed write attempts. You can mitigate the item collection size issues by monitoring and alerting on item collection sizes before they impact your application.

 **Note**

Once created, you cannot delete a local secondary index.

For strategies on working within the limit and taking corrective action, see [Item collection size limit](#).

Take advantage of sparse indexes

For any item in a table, DynamoDB writes a corresponding index entry **only if the index sort key value is present in the item**. If the sort key doesn't appear in every table item, or if the index partition key is not present in the item, the index is said to be *sparse*.

Sparse indexes are useful for queries over a small subsection of a table. For example, suppose that you have a table where you store all your customer orders, with the following key attributes:

- Partition key: CustomerId
- Sort key: OrderId

To track open orders, you can insert an attribute named `isOpen` in order items that have not already shipped. Then when the order ships, you can delete the attribute. If you then create an index on `CustomerId` (partition key) and `isOpen` (sort key), only those orders with `isOpen` defined appear in it. When you have thousands of orders of which only a small number are open, it's faster and less expensive to query that index for open orders than to scan the entire table.

Instead of using a type of attribute like `isOpen`, you could use an attribute with a value that results in a useful sort order in the index. For example, you could use an `OrderOpenDate` attribute set to the date on which each order was placed, and then delete it after the order is fulfilled. That way, when you query the sparse index, the items are returned sorted by the date on which each order was placed.

Examples of sparse indexes in DynamoDB

Global secondary indexes are sparse by default. When you create a global secondary index, you specify a partition key and optionally a sort key. Only items in the base table that contain those attributes appear in the index.

By designing a global secondary index to be sparse, you can provision it with lower write throughput than that of the base table, while still achieving excellent performance.

For example, a gaming application might track all scores of every user, but generally only needs to query a few high scores. The following design handles this scenario efficiently:

Table	Primary Key		Data Attributes...			
	Partition Key	Sort Key				
	Player_ID	Game_ID	Attribute 1		Attribute 2	
Rick	Rick	Game_1	Score: 36,750 <i>(game score)</i>	Date: 2017-11-14 <i>(date of game)</i>		
		Game_2	Score: 69,450 <i>(game score)</i>	Date: 2017-12-31 <i>(date of game)</i>		
		Game_3	Score: 135,900 <i>(game score)</i>	Date: 2018-01-19 <i>(date of game)</i>	Award: Champ <i>(type of award)</i>	
Padma	Padma	Game_4	Score: 25,350 <i>(game score)</i>	Date: 2018-01-27 <i>(date of game)</i>		
		Game_5	Score: 69,450 <i>(game score)</i>	Date: 2028-01-19 <i>(date of game)</i>		
		Game_6	Score: 147,300 <i>(game score)</i>	Date: 2018-02-02 <i>(date of game)</i>	Award: Champ <i>(type of award)</i>	
		Game_7	Score: 169,100 <i>(game score)</i>	Date: 2018-03-10 <i>(date of game)</i>	Award: Champ <i>(type of award)</i>	

Here, Rick has played three games and achieved Champ status in one of them. Padma has played four games and achieved Champ status in two of them. Notice that the Award attribute is present only in items where the user achieved an award. The associated global secondary index looks like the following:

GSI	Primary Key	Projected Attributes...			
	Partition Key				
	Award	Player_ID	Game_ID	Score	Date
	Champ	Rick	Game_3	135,900	2018-01-19
		Padma	Game_6	147,300	2018-02-02
		Padma	Game_7	169,100	2018-03-10

The global secondary index contains only the high scores that are frequently queried, which are a small subset of the items in the base table.

Using Global Secondary Indexes for materialized aggregation queries

Maintaining near real-time aggregations and key metrics on top of rapidly changing data is becoming increasingly valuable to businesses for making rapid decisions. For example, a music library might want to showcase its most downloaded songs in near-real time.

Consider the following music library table layout:

Music Library Table

Primary Key		Data-Item Attributes...				
Partition Key	Sort Key	Attribute 1		Attribute 2		Attribute 3
Song-129 <i>(song ID)</i>	Details	Title:	Wild Love <i>(song title)</i>	Artist:	Argyboots <i>(artist or band name)</i>	Downloads: 15,314,822 <i>(lifetime total downloads)</i>
		GSI Primary Key		GSI Secondary Key		...etc.
	Month-2018-01	Month:	2018-01 <i>(download month)</i>	MonthTotal:	1,746,992 <i>(month total downloads)</i>	
		Time:	2018-01-01T00:00:07 <i>(download timestamp)</i>			
		Time:	2018-01-01T00:00:07 <i>(download timestamp)</i>			
	Dld-9349823681	Time:	2018-01-01T00:00:07 <i>(download timestamp)</i>			

The table in this example stores songs with the songID as the partition key. You can enable Amazon DynamoDB Streams on this table and attach a Lambda function to the streams so that as each song is downloaded, an entry is added to the table with Partition-Key=SongID and Sort-Key=DownloadID. As these updates are made, they trigger a Lambda function in DynamoDB Streams. The Lambda function can aggregate and group the downloads by songID.

and update the top-level item, `Partition-Key=songID`, and `Sort-Key=Month`. Keep in mind that if a lambda execution fails just after writing the new aggregated value, it may be retried and aggregate the value more than once, leaving you with an approximate value.

To read the updates in near-real time, with single-digit millisecond latency, use the global secondary index with query conditions `Month=2018-01`, `ScanIndexForward=False`, `Limit=1`.

Another key optimization used here is that the global secondary index is a sparse index and is available only on the items that need to be queried to retrieve the data in real time. The global secondary index can serve additional workflows that need information on the top 10 songs that were popular, or any song downloaded in that month.

Overloading Global Secondary Indexes

Although Amazon DynamoDB has a default quota of 20 global secondary indexes per table, in practice, you can index across far more than 20 data fields. As opposed to a table in a relational database management system (RDBMS), in which the schema is uniform, a table in DynamoDB can hold many different kinds of data items at one time. In addition, the same attribute in different items can contain entirely different kinds of information.

Consider the following example of a DynamoDB table layout that saves a variety of different kinds of data.

Primary Key		Data-Item Attributes...				
Partition Key	Sort Key	Attribute 1		Attribute 2		...
HR-974 <i>(employee ID)</i>	Employee_Name	Data:	Murphy, John <i>(employee name)</i>	Start:	2008-11-08 <i>(start date)</i>	...etc.
	YYYY-Q1	Data:	\$5,477 <i>(order totals in USD)</i>	Name:	Murphy, John <i>(employee name)</i>	
	HR_confidential	Data:	2008-11-08 <i>(hire date)</i>	Name:	Murphy, John <i>(employee name)</i>	...etc.
	Warehouse_01	Data:	Murphy, John <i>(employee name)</i>			
	v0_Job_title	Data:	Operator-1 <i>(job title)</i>	Start:	2008-11-08 <i>(start date)</i>	...etc.
	v1_Job_title	Data:	Operator-2 <i>(job title)</i>	Start:	2016-11-04 <i>(start date)</i>	...etc.
	v2_Job_title	Data:	Supervisor-1 <i>(job title)</i>	Start:	2017-11-01 <i>(start date)</i>	...etc.

The Data attribute, which is common to all the items, has different content depending on its parent item. If you create a global secondary index for the table that uses the table's sort key as its partition key and the Data attribute as its sort key, you can make a variety of different queries using that single global secondary index. These queries might include the following:

- Look up an employee by name in the global secondary index, using Employee_Name as the partition key value and the employee's name (for example Murphy, John) as the sort key value.
- Use the global secondary index to find all employees working in a particular warehouse by searching on a warehouse ID (such as Warehouse_01).
- Get a list of recent hires, querying the global secondary index on HR_confidential as a partition key value and using a range of dates in the sort key value.

Using Global Secondary Index write sharding for selective table queries

Applications frequently need to identify a small subset of items in an Amazon DynamoDB table that meet a certain condition. When these items are distributed randomly across the partition keys of the table, you could resort to a table scan to retrieve them. This option can be expensive, but it works well when a large number of items on the table meet the search condition. However, when the key space is large and the search condition is very selective, this strategy can cause a lot of unnecessary processing.

A better solution can be to query the data. To enable selective queries across the entire key space, you can use write sharding by adding an attribute containing a (0-N) value to every item that you will use for the global secondary index partition key.

The following is an example of a schema that uses this in a Critical-Event workflow:

	Primary Key <i>Partition Key</i>	Data Attributes...								
		Attribute 1		Attribute 2		Attribute 3		Attribute 4		
Event_ID		Time: event timestamp	2018-02-07T08:42:40	State:	INFO (event state)	GSI PK:	(random: 0-N) (random GSI-PK value)	GSI SK:	INFO#2018-02-07T08:42:40 (composite state-time)	...etc.
EID_12345		Time: event timestamp	2018-02-07T08:32:40	State:	CRITICAL (event state)	GSI PK:	(random: 0-N) (random GSI-PK value)	GSI SK:	CRITICAL#2018-02-07T08:32:40 (composite state-time)	...etc.
EID_12346		Time: event timestamp	2018-02-07T08:22:40	State:	WARN (event state)	GSI PK:	(random: 0-N) (random GSI-PK value)	GSI SK:	WARN#2018-02-07T08:22:40 (composite state-time)	...etc.
EID_12347		Time: event timestamp	2018-02-07T08:12:40	State:	INFO (event state)	GSI PK:	(random: 0-N) (random GSI-PK value)	GSI SK:	INFO#2018-02-07T08:12:40 (composite state-time)	...etc.
EID_12348		Time: event timestamp	2018-02-07T08:12:40	State:	INFO (event state)	GSI PK:	(random: 0-N) (random GSI-PK value)	GSI SK:	INFO#2018-02-07T08:12:40 (composite state-time)	...etc.

	Primary Key		Data Attributes...		
	Partition Key	Sort Key	Data Attributes...		
	GSI PK	GSI SK	...		
	[0-N]	INFO#2018-02-07T08:42:40 (composite state-time)	...etc.		
	[0-N]	CRITICAL#2018-02-07T08:32:40 (composite state-time)	...etc.		
	[0-N]	WARN#2018-02-07T08:22:40 (composite state-time)	...etc.		
	[0-N]	INFO#2018-02-07T08:12:40 (composite state-time)	...etc.		

Using this schema design, the event items are distributed across 0–N partitions on the GSI, allowing a scatter read using a sort condition on the composite key to retrieve all items with a given state during a specified time period.

This schema pattern delivers a highly selective result set at minimal cost, without requiring a table scan.

Using Global Secondary Indexes to create an eventually consistent replica

You can use a global secondary index to create an eventually consistent replica of a table. Creating a replica can allow you to do the following:

- **Set different provisioned read capacity for different readers.** For example, suppose that you have two applications: One application handles high-priority queries and needs the highest levels of read performance, whereas the other handles low-priority queries that can tolerate throttling of read activity.

If both of these applications read from the same table, a heavy read load from the low-priority application could consume all the available read capacity for the table. This would throttle the high-priority application's read activity.

Instead, you can create a replica through a global secondary index whose read capacity you can set separate from that of the table itself. You can then have your low-priority app query the replica instead of the table.

- **Eliminate reads from a table entirely.** For example, you might have an application that captures a high volume of clickstream activity from a website, and you don't want to risk having reads interfere with that. You can isolate this table and prevent reads by other applications (see [Using IAM policy conditions for fine-grained access control](#)), while letting other applications read a replica created using a global secondary index.

To create a replica, set up a global secondary index that has the same key schema as the parent table, with some or all of the non-key attributes projected into it. In applications, you can direct some or all read activity to this global secondary index rather than to the parent table. You can then adjust the provisioned read capacity of the global secondary index to handle those reads without changing the parent table's provisioned read capacity.

There is always a short propagation delay between a write to the parent table and the time when the written data appears in the index. In other words, your applications should take into account that the global secondary index replica is only *eventually consistent* with the parent table.

You can create multiple global secondary index replicas to support different read patterns. When you create the replicas, project only the attributes that each read pattern actually requires. An application can then consume less provisioned read capacity to obtain only the data it needs rather than having to read the item from the parent table. This optimization can result in significant cost savings over time.

Best practices for storing large items and attributes

Amazon DynamoDB limits the size of each item that you store in a table to 400 KB (see [Service, account, and table quotas in Amazon DynamoDB](#)). If your application needs to store more data in an item than the DynamoDB size limit permits, you can try compressing one or more large attributes or breaking the item into multiple items (efficiently indexed by sort keys). You can also store the item as an object in Amazon Simple Storage Service (Amazon S3) and store the Amazon S3 object identifier in your DynamoDB item.

As a best practice, you should utilize the [ReturnConsumedCapacity](#) parameter when writing items to monitor and alert on items sizes that approach the 400 KB maximum item size. Exceeding the maximum item size will result in failed write attempts. Monitoring and alerting on item sizes will enable you to mitigate the items size issues before they impact your application.

Compressing large attribute values

Compressing large attribute values can let them fit within item limits in DynamoDB and reduce your storage costs. Compression algorithms such as GZIP or LZO produce binary output that you can then store in a Binary attribute type within the item.

As an example, consider a table that stores messages written by forum users. Such messages often contain long strings of text, which are candidates for compression. While compression can reduce item sizes, the downside is that the compressed attribute values are not useful for filtering.

For sample code that demonstrates how to compress such messages in DynamoDB, see the following:

- [Example: Handling binary type attributes using the AWS SDK for Java document API](#)
- [Example: Handling binary type attributes using the AWS SDK for .NET low-level API](#)

Vertical partitioning

An alternative solution to dealing with large items is to break them down into smaller chunks of data and associating all relevant items by the partition key value. You can then use a sort key string to identify the associated information stored alongside it. By doing this, and having multiple items grouped by the same partition key value, you are creating an [*item collection*](#).

For more information on this approach, see:

- [Use vertical partitioning to scale data efficiently in Amazon DynamoDB](#)
- [Implement vertical partitioning in Amazon DynamoDB using AWS Glue](#)

Storing large attribute values in Amazon S3

As mentioned previously, you can also use Amazon S3 to store large attribute values that cannot fit in a DynamoDB item. You can store them as an object in Amazon S3 and then store the object identifier in your DynamoDB item.

You can also use the object metadata support in Amazon S3 to provide a link back to the parent item in DynamoDB. Store the primary key value of the item as Amazon S3 metadata of the object in Amazon S3. Doing this often helps with maintenance of the Amazon S3 objects.

For example, consider the `ProductCatalog` table in the [Creating tables and loading data for code examples in DynamoDB](#) section. Items in this table store information about item price, description, book authors, and dimensions for other products. If you wanted to store an image of each product that was too large to fit in an item, you could store the images in Amazon S3 instead of in DynamoDB.

When implementing this strategy, keep the following in mind:

- DynamoDB doesn't support transactions that cross Amazon S3 and DynamoDB. Therefore, your application must deal with any failures, which could include cleaning up orphaned Amazon S3 objects.
- Amazon S3 limits the length of object identifiers. So you must organize your data in a way that doesn't generate excessively long object identifiers or violate other Amazon S3 constraints.

For more information about how to use Amazon S3, see the [Amazon Simple Storage Service User Guide](#).

Best practices for handling time series data in DynamoDB

General design principles in Amazon DynamoDB recommend that you keep the number of tables you use to a minimum. For most applications, a single table is all you need. However, for time series data, you can often best handle it by using one table per application per period.

Design pattern for time series data

Consider a typical time series scenario, where you want to track a high volume of events. Your write access pattern is that all the events being recorded have today's date. Your read access pattern might be to read today's events most frequently, yesterday's events much less frequently, and then older events very little at all. One way to handle this is by building the current date and time into the primary key.

The following design pattern often handles this kind of scenario effectively:

- Create one table per period, provisioned with the required read and write capacity and the required indexes.
- Before the end of each period, prebuild the table for the next period. Just as the current period ends, direct event traffic to the new table. You can assign names to these tables that specify the periods they have recorded.
- As soon as a table is no longer being written to, reduce its provisioned write capacity to a lower value (for example, 1 WCU), and provision whatever read capacity is appropriate. Reduce the provisioned read capacity of earlier tables as they age. You might choose to archive or delete the tables whose contents are rarely or never needed.

The idea is to allocate the required resources for the current period that will experience the highest volume of traffic and scale down provisioning for older tables that are not used actively, therefore saving costs. Depending on your business needs, you might consider write sharding to distribute traffic evenly to the logical partition key. For more information, see [Using write sharding to distribute workloads evenly](#).

Time series table examples

The following is a time series data example in which the current table is provisioned at a higher read/write capacity and the older tables are scaled down because they are accessed infrequently.

Current table

Provisioned at: WCU=750 and RCU=300

Primary Key		Attributes		
Partition Key	Sort Key	Radiant Intensity	Wavelength	...
2018-03-15	00:00:00.002	17.372 W/Sr	713 nm	...
2018-03-15	00:00:00.004	17.385 W/Sr	712 nm	...
2018-03-15	00:00:00.005	17.478 W/Sr	708 nm	...
2018-03-15	00:00:00.007	19.172 W/Sr	674 nm	...
...

Previous table

Provisioned at: WCU=1 and RCU=100

Primary Key		Attributes		
Partition Key	Sort Key	Radiant Intensity	Wavelength	...
2018-03-14	00:00:00.001	16.473 W/Sr	512	...
2018-03-14	00:00:00.003	16.489 W/Sr	519	...
2018-03-14	00:00:00.004	16.814 W/Sr	522	...
2018-03-14	00:00:00.006	16.719 W/Sr	506	...
...

Older table

Provisioned at: WCU=1 and RCU=1

Primary Key		Attributes		
Partition Key	Sort Key	Radiant Intensity	Wavelength	...
2018-03-10	00:00:00.001	13.669 W/Sr	456	...
2018-03-10	00:00:00.002	13.522 W/Sr	459	...
2018-03-10	00:00:00.004	13.596 W/Sr	457	...
2018-03-10	00:00:00.005	15.721 W/Sr	425	...
...

Best practices for managing many-to-many relationships

Adjacency lists are a design pattern that is useful for modeling many-to-many relationships in Amazon DynamoDB. More generally, they provide a way to represent graph data (nodes and edges) in DynamoDB.

Adjacency list design pattern

When different entities of an application have a many-to-many relationship between them, the relationship can be modeled as an adjacency list. In this pattern, all top-level entities (synonymous to nodes in the graph model) are represented using the partition key. Any relationships with other entities (edges in a graph) are represented as an item within the partition by setting the value of the sort key to the target entity ID (target node).

The advantages of this pattern include minimal data duplication and simplified query patterns to find all entities (nodes) related to a target entity (having an edge to a target node).

A real-world example where this pattern has been useful is an invoicing system where invoices contain multiple bills. One bill can belong in multiple invoices. The partition key in this example is either an `InvoiceID` or a `BillID`. `BillID` partitions have all attributes specific to bills. `InvoiceID` partitions have an item storing invoice-specific attributes, and an item for each `BillID` that rolls up to the invoice.

The schema looks like the following.

	Primary Key		Data Attributes...		
	Partition Key	Sort Key (and GSI PK)	Dated:	2018-02-07	
Invoice-92551	Inv_ID:	Invoice-92551 <i>(invoice ID)</i>	Dated:	2018-02-07 <i>(date created)</i>	More attributes of this invoice...
	Bill_ID:	Bill-4224663 <i>(bill ID)</i>	Dated:	2017-12-03 <i>(date created)</i>	Attributes of this bill <i>in this invoice..</i>
	Bill_ID:	Bill-4224687 <i>(bill ID)</i>	Dated:	2018-01-09 <i>(date created)</i>	Attributes of this bill <i>in this invoice..</i>
Invoice-92552	Inv_ID:	Invoice-92552 <i>(invoice ID)</i>	Dated:	2018-03-04 <i>(date created)</i>	More attributes of this invoice...
	Bill_ID:	Bill-4224687 <i>(bill ID)</i>	Dated:	2018-01-09 <i>(date created)</i>	Attributes of this bill <i>in this invoice..</i>
Bill-4224663	Bill_ID:	Bill-4224663 <i>(bill ID)</i>	Dated:	2017-12-03 <i>(date created)</i>	More attributes of this bill...
Bill-4224687	Bill_ID:	Bill-4224687 <i>(bill ID)</i>	Dated:	2018-01-09 <i>(date created)</i>	More attributes of this bill...

Using the preceding schema, you can see that all bills for an invoice can be queried using the primary key on the table. To look up all invoices that contain a part of a bill, create a global secondary index on the table's sort key.

The projections for the global secondary index look like the following.

	Primary Key <i>Partition Key</i>	Projected Attributes...		
		Bill_ID:	Bill-4224663 <i>(table primary key)</i>	Attributes of this bill...
GSI	Bill-4224663	Inv_ID:	Invoice-92551 <i>(table primary key)</i>	Attributes of this bill <i>in this invoice</i> ..
		Bill_ID:	Bill-4224687 <i>(table primary key)</i>	Attributes of this bill...
	Bill-4224687	Inv_ID:	Invoice-92551 <i>(table primary key)</i>	Attributes of this bill <i>in this invoice</i> ..
		Inv_ID:	Invoice-92552 <i>(table primary key)</i>	Attributes of this bill <i>in this invoice</i> ..
		Inv_ID:	Invoice-92551 <i>(table primary key)</i>	Attributes of this invoice...
	Invoice-92551	Inv_ID:	Invoice-92552 <i>(table primary key)</i>	Attributes of this invoice...
	Invoice-92552	Inv_ID:	Invoice-92552 <i>(table primary key)</i>	Attributes of this invoice...

Materialized graph pattern

Many applications are built around understanding rankings across peers, common relationships between entities, neighbor entity state, and other types of graph style workflows. For these types of applications, consider the following schema design pattern.

	Primary Key		Attributes		
	PK (NodeId)	SK (TypeTarget, GSI 2 SK)	Data	GSI PK	Graph Projections
1	PERSON 1	DATE 2 BIRTH	1980-12-19	Hash(Person.Data)	
		PERSON 1	Data (GSI1 SK)	GSI PK	
		PERSON 5 FRIEND	John Doe	Hash(Person.Data)	
		PLACE 4 BIRTH	Data	GSI PK	
		SKILL 6	Jane Smith	Hash(Person.Data)	
	2	DATE 2	Data	GSI PK	
		DATE 2	1980-12-19	0	
	3	PLACE 3	Data	GSI PK	
		PLACE 3	UK England London	0	
	4	PLACE 4	Data	GSI PK	
		PLACE 4	USA Texas Austin	0	...
5	PERSON 5	DATE 2 BIRTH	Data	GSI PK	
		PERSON 5	1980-12-19	Hash(Person.Data)	
		PERSON 1 FRIEND	Data	GSI PK	
		PERSON 1 FRIEND	Jane Smith	Hash(Person.Data)	
		PLACE 3 BIRTH	Data	GSI PK	
	6	SKILL 6	UK England London	Hash(Person.Data)	
		SKILL 7	Data	GSI PK	
		SKILL 7	Guitar Advanced	Hash(Person.Data)	
		SKILL 6	Data	GSI PK	
		SKILL 7	Java Developer	0	

	Primary Key		Attributes		
	GSI PK	GSI 1 SK (Data)	NodeID	TypeTarget	Graph Projections
GSI 1 O-N	1980-12-19	Guitar	2	DATE 2	
			NodeID	TypeTarget	
			1		
			NodeID	DATE 2 BIRTH	
			5		
		Guitar Advanced	NodeID	TypeTarget	
			7		
	Jane Smith	Java Developer	NodeID	SKILL 7	
			5		
			NodeID	TypeTarget	
			1	Person 5	
			NodeID	TypeTarget	
O-N	Java Developer Senior	John Doe	6	Person 5 FRIEND	
			NodeID	TypeTarget	
			1	SKILL 6	
			NodeID		
			1		
	UK England London	USA Texas Austin	NodeID	TypeTarget	
			3	PLACE 3	
			NodeID	TypeTarget	
			1	PLACE 3 BIRTH	
			NodeID	TypeTarget	

	Primary Key		Attributes		
	GSI PK	GSI 2 SK (TypeTarget)	NodeID	Data	Graph Projections
GSI 2 O-N	DATE 2		NodeID	1980-12-19	...
			2		
	DATE 2 BIRTH		NodeID		
			1		
			NodeID		
			5		
			NodeID		
	PERSON 1		1		
			NodeID		
	PERSON 1 FRIEND		5		
			NodeID		
	PERSON 5		5		
			NodeID		
	PERSON 5 FRIEND		1		
			NodeID		
	PLACE 3		3		
			NodeID		
	PLACE 3 BIRTH		5		
			NodeID		
	PLACE 4		4		
			NodeID		
	PLACE 4 BIRTH		1		
			NodeID		
	SKILL 6		6	Java Developer	...
			NodeID		
			1		
	SKILL 7		NodeID	Java Developer Senior	...
			7		
			NodeID		
			5		

The preceding schema shows a graph data structure that is defined by a set of data partitions containing the items that define the edges and nodes of the graph. Edge items contain a Target and a Type attribute. These attributes are used as part of a composite key name "TypeTarget" to identify the item in a partition in the primary table or in a second global secondary index.

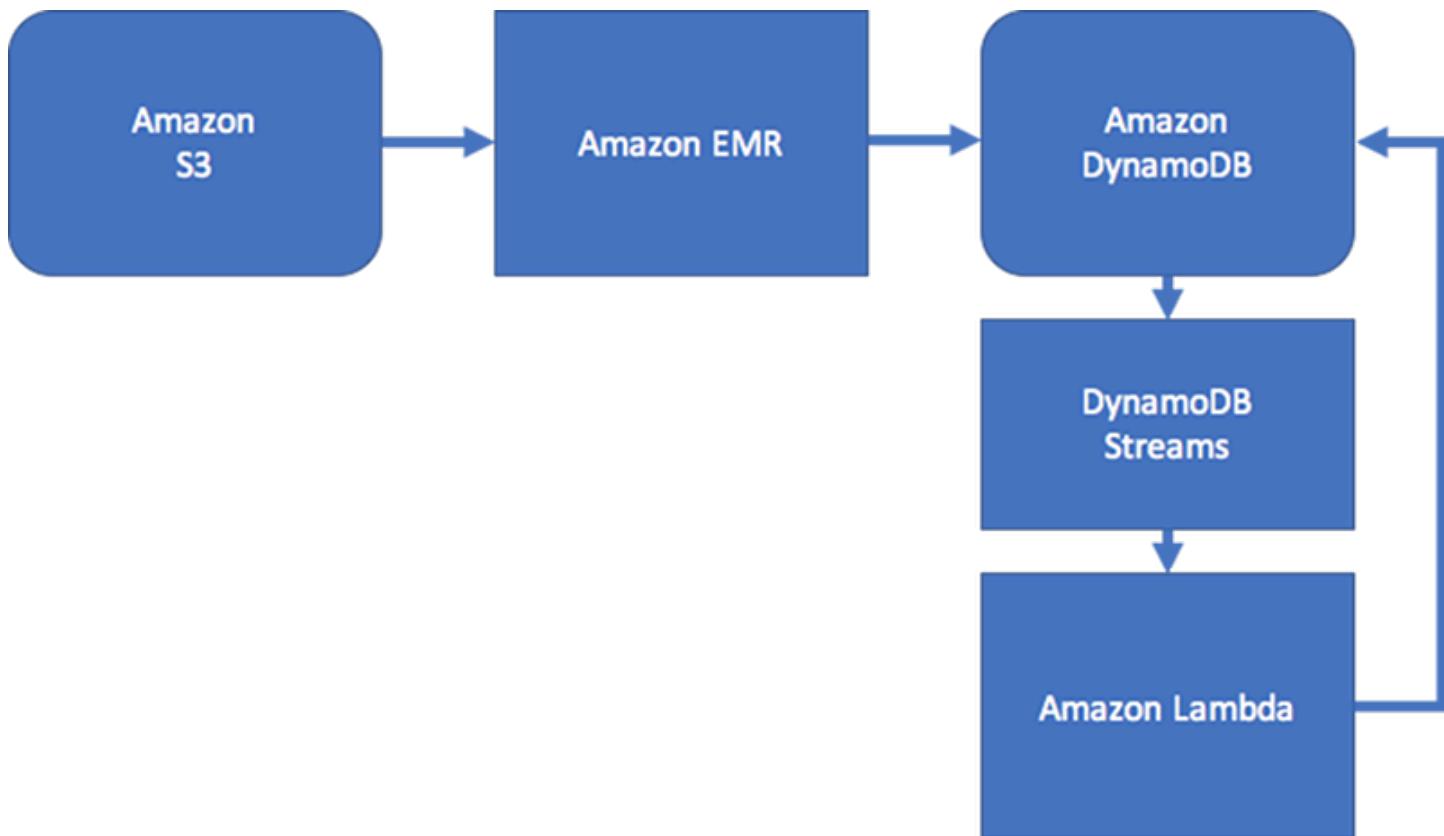
The first global secondary index is built on the Data attribute. This attribute uses global secondary index-overloading as described earlier to index several different attribute types, namely Dates, Names, Places, and Skills. Here, one global secondary index is effectively indexing four different attributes.

As you insert items into the table, you can use an intelligent sharding strategy to distribute item sets with large aggregations (birthdate, skill) across as many logical partitions on the global secondary indexes as are needed to avoid hot read/write problems.

The result of this combination of design patterns is a solid datastore for highly efficient real-time graph workflows. These workflows can provide high-performance neighbor entity state and edge aggregation queries for recommendation engines, social-networking applications, node rankings, subtree aggregations, and other common graph use cases.

If your use case isn't sensitive to real-time data consistency, you can use a scheduled Amazon EMR process to populate edges with relevant graph summary aggregations for your workflows. If your application doesn't need to know immediately when an edge is added to the graph, you can use a scheduled process to aggregate results.

To maintain some level of consistency, the design could include Amazon DynamoDB Streams and AWS Lambda to process edge updates. It could also use an Amazon EMR job to validate results on a regular interval. This approach is illustrated by the following diagram. It is commonly used in social networking applications, where the cost of a real-time query is high and the need to immediately know individual user updates is low.



IT service-management (ITSM) and security applications generally need to respond in real time to entity state changes composed of complex edge aggregations. Such applications need a system that can support real-time multiple node aggregations of second- and third-level relationships, or complex edge traversals. If your use case requires these types of real-time graph query workflows, we recommend that you consider using [Amazon Neptune](#) to manage these workflows.

Note

If you need to query highly connected datasets or execute queries that need to traverse multiple nodes (also known as multi-hop queries) with millisecond latency, you should consider using [Amazon Neptune](#). Amazon Neptune is a purpose-built, high-performance graph database engine optimized for storing billions of relationships and querying the graph with millisecond latency.

Best practices for implementing a hybrid database system

In some circumstances, migrating from one or more relational database management systems (RDBMS) to Amazon DynamoDB might not be advantageous. In these cases, it might be preferable to create a hybrid system.

If you don't want to migrate everything to DynamoDB

For example, some organizations have large investments in the code that produces a multitude of reports needed for accounting and operations. The time it takes to generate a report is not important to them. The flexibility of a relational system is well suited to this kind of task, and re-creating all those reports in a NoSQL context might be prohibitively difficult.

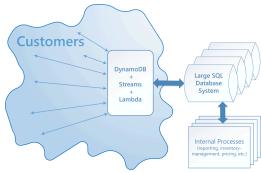
Some organizations also maintain a variety of legacy relational systems that they have acquired or inherited over decades. Migrating data from these systems might be too risky and expensive to justify the effort.

However, the same organizations may now find that their operations depend on high-traffic customer-facing websites, where millisecond response is essential. Relational systems can't scale to meet this requirement except at huge (and often unacceptable) expense.

In these situations, the answer might be to create a hybrid system, in which DynamoDB creates a materialized view of data stored in one or more relational systems and handles high-traffic requests against this view. This type of system can potentially reduce costs by eliminating server hardware, maintenance, and RDBMS licenses that were previously needed to handle customer-facing traffic.

How a hybrid system can be implemented

DynamoDB can take advantage of DynamoDB Streams and AWS Lambda to integrate seamlessly with one or more existing relational database systems:



A system that integrates DynamoDB Streams and AWS Lambda can provide several advantages:

- It can operate as a persisted cache of materialized views.
- It can be set up to fill gradually with data as that data is queried for, and as data is modified in the SQL system. This means that the entire view does not need to be pre-populated. This in turn means that provisioned throughput capacity is more likely to be used efficiently.
- It has low administrative costs and is highly available and reliable.

For this kind of integration to be implemented, essentially three kinds of interoperation must be provided.



1. **Fill the DynamoDB cache incrementally.** When an item is queried, look for it first in DynamoDB. If it is not there, look for it in the SQL system, and load it into DynamoDB.
2. **Write through a DynamoDB cache.** When a customer changes a value in DynamoDB, a Lambda function is triggered to write the new data back to the SQL system.
3. **Update DynamoDB from the SQL system.** When internal processes such as inventory management or pricing change a value in the SQL system, a stored procedure is triggered to propagate the change to the DynamoDB materialized view.

These operations are straightforward, and not all of them are needed for every scenario.

A hybrid solution can also be useful when you want to rely primarily on DynamoDB, but you also want to maintain a small relational system for one-time queries, or for operations that need special security or that are not time-critical.

Best practices for modeling relational data in DynamoDB

This section provides best practices for modeling relational data in Amazon DynamoDB. First, we introduce traditional data modeling concepts. Then, we describe the advantages of using DynamoDB over traditional relational database management systems—how it eliminates the need for JOIN operations and reduces overhead.

We then explain how to design a DynamoDB table that scales efficiently. Finally, we provide an example of how to model relational data in DynamoDB.

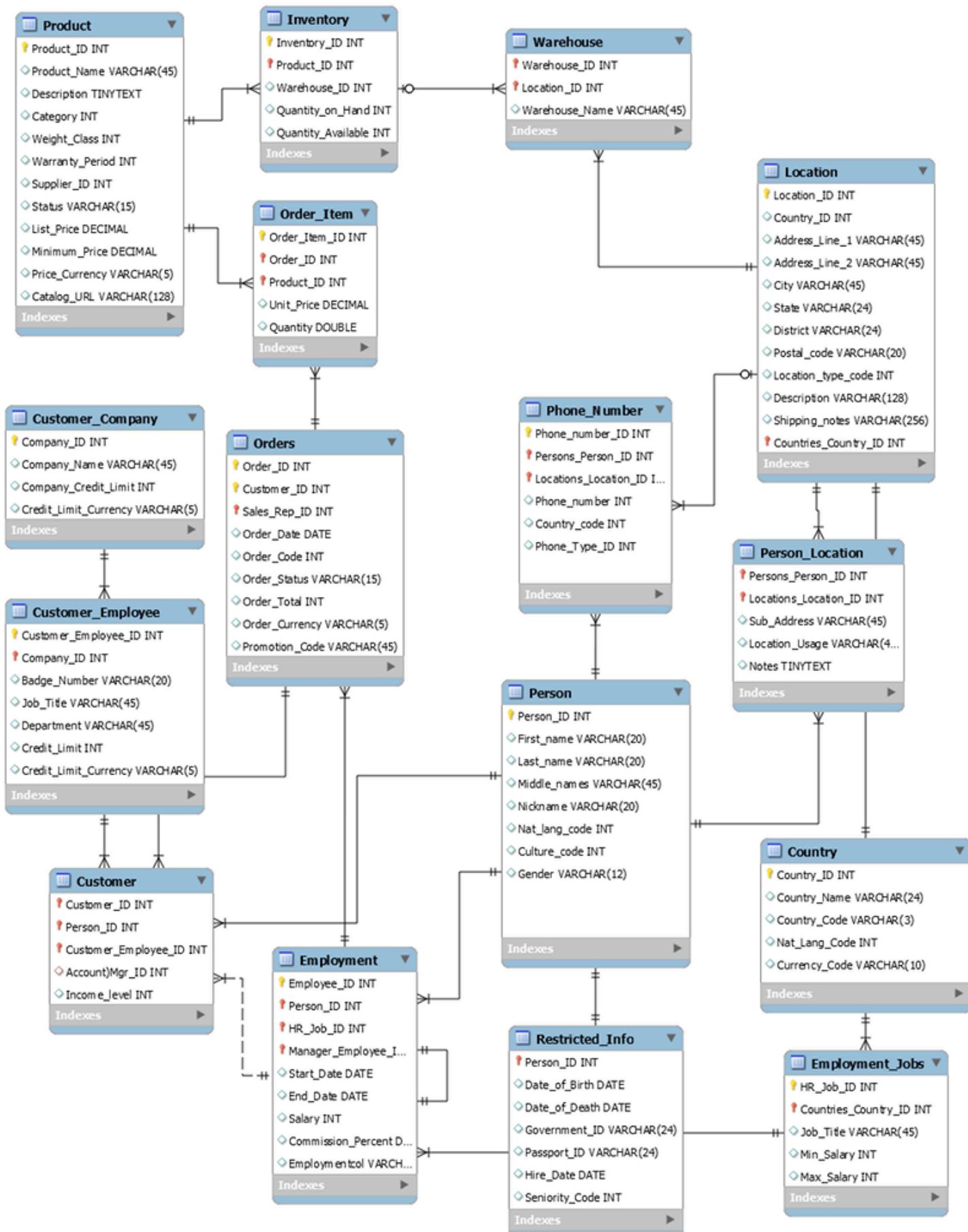
Topics

- [Traditional relational database models](#)
- [How DynamoDB eliminates the need for JOIN operations](#)
- [How DynamoDB transactions eliminate overhead to the write process](#)
- [First steps for modeling relational data in DynamoDB](#)
- [Example of modeling relational data in DynamoDB](#)

Traditional relational database models

A traditional relational database management system (RDBMS) stores data in a normalized relational structure. The objective of the relational data model is to reduce the duplication of data (through normalization) to support referential integrity and reduce data anomalies.

The following schema is an example of a relational data model for a generic order-entry application. The application supports a human resources schema that backs the operational and business support systems of a theoretical manufacturer.



As a non-relational database service, DynamoDB offers many advantages over traditional relational database management systems.

How DynamoDB eliminates the need for JOIN operations

An RDBMS uses a structure query language (SQL) to return data to the application. Because of the normalization of the data model, such queries typically require the use of the JOIN operator to combine data from one or more tables.

For example, to generate a list of purchase order items sorted by the quantity in stock at all warehouses that can ship each item, you could issue the following SQL query against the preceding schema.

```
SELECT * FROM Orders
    INNER JOIN Order_Items ON Orders.Order_ID = Order_Items.Order_ID
    INNER JOIN Products ON Products.Product_ID = Order_Items.Product_ID
    INNER JOIN Inventories ON Products.Product_ID = Inventories.Product_ID
    ORDER BY Quantity_on_Hand DESC
```

SQL queries of this kind can provide a flexible API for accessing data, but they require a significant amount of processing. Each join in the query increases the runtime complexity of the query because the data for each table must stage and then be assembled to return the result set.

Additional factors that can impact how long it takes the queries to run are the size of the tables and whether the columns being joined have indexes. The preceding query initiates complex queries across several tables and then sorts the result set.

Eliminating the need for JOINs is at the heart of NoSQL data modeling. This is why we built DynamoDB to support Amazon.com, and why DynamoDB can deliver consistent performance at any scale. Given the runtime complexity of SQL queries and JOINs, RDBMS performance is not constant at scale. This causes performance issues as customer applications grow.

While normalizing data does reduce the amount of data stored to disk, often the most constrained resources that impact performance are CPU time and network latency.

DynamoDB is built to minimize both constraints by eliminating JOINs (and encouraging denormalization of data) and optimizing the database architecture to fully answer an application query with a single request to an item. These qualities enable DynamoDB to provide single-digit, millisecond performance at any scale. This is because the runtime complexity for DynamoDB operations is constant, regardless of data size, for common access patterns.

How DynamoDB transactions eliminate overhead to the write process

Another factor that can slow down an RDBMS is the use of transactions to write to a normalized schema. As shown in the example, relational data structures used by most online transaction processing (OLTP) applications must be broken down and distributed across multiple logical tables when they are stored in an RDBMS.

Therefore, an ACID-compliant transaction framework is necessary to avoid race conditions and data integrity issues that could occur if an application tries to read an object that is in the process of being written. Such a transaction framework, when coupled with a relational schema, can add significant overhead to the write process.

The implementation of transactions in DynamoDB prohibits common scaling issues that are found with an RDBMS. DynamoDB does this by issuing a transaction as a single API call and bounding the number of items that can be accessed in that single transaction. Long-running transactions can cause operational issues by holding locks on the data either for a long time, or perpetually, because the transaction is never closed.

To prevent such issues in DynamoDB, transactions were implemented with two distinct API operations: `TransactWriteItems` and `TransactGetItems`. These API operations do not have begin and end semantics that are common in an RDBMS. Further, DynamoDB has a 100-item access limit within a transaction to similarly prevent long-running transactions. To learn more about DynamoDB transactions, see [Working with transactions](#).

For these reasons, when your business requires a low-latency response to high-traffic queries, taking advantage of a NoSQL system generally makes technical and economic sense. Amazon DynamoDB helps solve the problems that limit relational system scalability by avoiding them.

The performance of an RDBMS does not typically scale well for the following reasons:

- It uses expensive joins to reassemble required views of query results.
- It normalizes data and stores it on multiple tables that require multiple queries to write to disk.
- It generally incurs the performance costs of an ACID-compliant transaction system.

DynamoDB scales well for these reasons:

- Schema flexibility lets DynamoDB store complex hierarchical data within a single item.
- Composite key design lets it store related items close together on the same table.

- Transactions are performed in a single operation. The limit for the number of items that can be accessed is 100, to avoid long-running operations.

Queries against the data store become much simpler, often in the following form:

```
SELECT * FROM Table_X WHERE Attribute_Y = "somevalue"
```

DynamoDB does far less work to return the requested data compared to the RDBMS in the earlier example.

First steps for modeling relational data in DynamoDB

Important

NoSQL design requires a different mindset than RDBMS design. For an RDBMS, you can create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. By contrast, in Amazon DynamoDB, you shouldn't start designing your schema until you know the questions that it needs to answer. Understanding the business problems and the application use cases up front is absolutely essential.

To start designing a DynamoDB table that will scale efficiently, you must take several steps first to identify the access patterns that are required by the operations and business support systems (OSS/BSS) that it needs to support:

- For new applications, review user stories about activities and objectives. Document the various use cases you identify, and analyze the access patterns that they require.
- For existing applications, analyze query logs to find out how people are currently using the system and what the key access patterns are.

After completing this process, you should end up with a list that might look something like the following.

Most Common/Import Access Patterns in Our Organization	
1	Look up employee details by employee ID
2	Query employee details by employee name
3	Find an employee's phone number(s)
4	Find a customer's phone number(s)
5	Get orders for a given customer within a given date range
6	Show all open orders within a given date range across all customers
7	See all employees hired recently
8	Find all employees working in a given warehouse
9	Get all items on order for a given product
10	Get current inventories for a given product at all warehouses
11	Get customers by account representative
12	Get orders by account representative and date
13	Get all employees with a given job title
14	Get inventory by product and warehouse
15	Get total product inventory
16	Get account representatives ranked by order total and sales period

In a real application, your list might be much longer. But this collection represents the range of query pattern complexity that you might find in a production environment.

A common approach to DynamoDB schema design is to identify application layer entities and use denormalization and composite key aggregation to reduce query complexity.

In DynamoDB, this means using composite sort keys, overloaded global secondary indexes, partitioned tables/indexes, and other design patterns. You can use these elements to structure the data so that an application can retrieve whatever it needs for a given access pattern using a single query on a table or index. The primary pattern that you can use to model the normalized schema shown in [Relational modeling](#) is the adjacency list pattern. Other patterns used in this design can include global secondary index write sharding, global secondary index overloading, composite keys, and materialized aggregations.

Important

In general, you should maintain as few tables as possible in a DynamoDB application.

Exceptions include cases where high-volume time series data are involved, or datasets that have very different access patterns. A single table with inverted indexes can usually enable simple queries to create and retrieve the complex hierarchical data structures required by your application.

To use NoSQL Workbench for DynamoDB to help visualize your partition key design, see [Building data models with NoSQL Workbench](#).

Example of modeling relational data in DynamoDB

This example describes how to model relational data in Amazon DynamoDB. A DynamoDB table design corresponds to the relational order entry schema that is shown in [Relational modeling](#). It follows the [Adjacency list design pattern](#), which is a common way to represent relational data structures in DynamoDB.

The design pattern requires you to define a set of entity types that usually correlate to the various tables in the relational schema. Entity items are then added to the table using a compound (partition and sort) primary key. The partition key of these entity items is the attribute that uniquely identifies the item and is referred to generically on all items as PK. The sort key attribute contains an attribute value that you can use for an inverted index or global secondary index. It is generically referred to as SK.

You define the following entities, which support the relational order entry schema.

1. HR-Employee - PK: EmployeeID, SK: Employee Name
2. HR-Region - PK: RegionID, SK: Region Name
3. HR-Country - PK: CountryId, SK: Country Name
4. HR-Location - PK: LocationID, SK: Country Name
5. HR-Job - PK: JobID, SK: Job Title
6. HR-Department - PK: DepartmentID, SK: DepartmentID
7. OE-Customer - PK: CustomerID, SK: AccountRepID
8. OE-Order - PK OrderID, SK: CustomerID
9. OE-Product - PK: ProductID, SK: Product Name
10. OE-Warehouse - PK: WarehouseID, SK: Region Name

After adding these entity items to the table, you can define the relationships between them by adding edge items to the entity item partitions. The following table demonstrates this step.

In this example, the Employee, Order, and Product Entity partitions on the table have additional edge items that contain pointers to other entity items on the table. Next, define a few global secondary indexes (GSIs) to support all the access patterns defined previously. The entity items don't all use the same type of value for the primary key or the sort key attribute. All that is required is to have the primary key and sort key attributes present to be inserted on the table.

The fact that some of these entities use proper names and others use other entity IDs as sort key values allows the same global secondary index to support multiple types of queries. This technique is called *GSI overloading*. It effectively eliminates the default limit of 20 global secondary indexes for tables that contain multiple item types. This is shown in the following diagram as *GSI 1*.

GSI 2 is designed to support a fairly common application access pattern, which is to get all the items on the table that have a certain state. For a large table with an uneven distribution of items across available states, this access pattern can result in a hot key, unless the items are distributed across more than one logical partition that can be queried in parallel. This design pattern is called *write sharding*.

To accomplish this for *GSI 2*, the application adds the *GSI 2 primary key* attribute to every *Order* item. It populates that with a random number in a range of 0–N, where N can generically be calculated using the following formula, unless there is a specific reason to do otherwise.

```
ItemsPerRCU = 4KB / AvgItemSize
```

```
PartitionMaxReadRate = 3K * ItemsPerRCU
```

```
N = MaxRequiredIO / PartitionMaxReadRate
```

For example, assume that you expect the following:

- Up to 2 million orders will be in the system, growing to 3 million in 5 years.
- Up to 20 percent of these orders will be in an OPEN state at any given time.
- The average order record is around 100 bytes, with three *OrderItem* records in the order partition that are around 50 bytes each, giving you an average order entity size of 250 bytes.

For that table, the N factor calculation would look like the following.

```
ItemsPerRCU = 4KB / 250B = 16
```

```
PartitionMaxReadRate = 3K * 16 = 48K
```

```
N = (0.2 * 3M) / 48K = 13
```

In this case, you need to distribute all the orders across at least 13 logical partitions on *GSI 2* to ensure that a read of all *Order* items with an OPEN status doesn't cause a hot partition on

the physical storage layer. It is a good practice to pad this number to allow for anomalies in the dataset. So a model using $N = 15$ is probably fine. As mentioned earlier, you do this by adding the random $0-N$ value to the GSI 2 PK attribute of each Order and OrderItem record that is inserted on the table.

This breakdown assumes that the access pattern that requires gathering all OPEN invoices occurs relatively infrequently so that you can use burst capacity to fulfill the request. You can query the following global secondary index using a State and Date Range Sort Key condition to produce a subset or all Orders in a given state as needed.

In this example, the items are randomly distributed across the 15 logical partitions. This structure works because the access pattern requires a large number of items to be retrieved. Therefore, it's unlikely that any of the 15 threads will return empty result sets that could potentially represent wasted capacity. A query always uses 1 read capacity unit (RCU) or 1 write capacity unit (WCU), even if nothing is returned or no data is written.

If the access pattern requires a high velocity query on this global secondary index that returns a sparse result set, it's probably better to use a hash algorithm to distribute the items rather than a random pattern. In this case, you might select an attribute that is known when the query is run at runtime and hash that attribute into a 0–14 key space when the items are inserted. Then they can be efficiently read from the global secondary index.

Finally, you can revisit the access patterns that were defined earlier. Following is the list of access patterns and the query conditions that you will use with the new DynamoDB version of the application to accommodate them.

	Access patterns	Query conditions
1	Look up Employee Details by Employee ID	Primary Key on table, ID="HR-EMPLOYEE"
2	Query Employee Details by Employee Name	Use GSI-1, PK="Employee Name"
3	Get an employee's current job details only	Primary Key on table, PK=HR-EMPLOYEE-1, SK starts with "JH"

	Access patterns	Query conditions
4	Get Orders for a customer for a date range	Use GSI-1, PK=CUSTOMER1, SK="STATUS-DATE", for each StatusCode
5	Show all Orders in OPEN status for a date range across all customers	Use GSI-2, PK=query in parallel for the range [0..N], SK between OPEN-Date1 and OPEN-Date2
6	All Employees hired recently	Use GSI-1, PK="HR-CO NFIDENTIAL', SK > date1
7	Find all Employees in specific Warehouse	Use GSI-1, PK=WAREHOUSE1
8	Get all Orderitems for a Product including warehouse location inventories	Use GSI-1, PK=PRODUCT1
9	Get customers by Account Rep	Use GSI-1, PK=ACCOUNT-REP
10	Get orders by Account Rep and date	Use GSI-1, PK=ACCOUNT-REP, SK="STATUS-DATE", for each StatusCode
11	Get all employees with specific Job Title	Use GSI-1, PK=JOBTITLE
12	Get inventory by Product and Warehouse	Primary Key on table, PK=OE-PRODUCT1,SK=PRODUCT1
13	Get total product inventory	Primary Key on table, PK=OE-PRODUCT1,SK=PRODUCT1
14	Get Account Reps ranked by Order Total and Sales Period	Use GSI-1, PK=YYYY-Q1, scanIndexForward=False

Best practices for querying and scanning data

This section covers some best practices for using Query and Scan operations in Amazon DynamoDB.

Performance considerations for scans

In general, Scan operations are less efficient than other operations in DynamoDB. A Scan operation always scans the entire table or secondary index. It then filters out values to provide the result you want, essentially adding the extra step of removing data from the result set.

If possible, you should avoid using a Scan operation on a large table or index with a filter that removes many results. Also, as a table or index grows, the Scan operation slows. The Scan operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation. For faster response times, design your tables and indexes so that your applications can use Query instead of Scan. (For tables, you can also consider using the GetItem and BatchGetItem APIs.)

Alternatively, you can design your application to use Scan operations in a way that minimizes the impact on your request rate. This can include modeling when it might be more efficient to use a global secondary index instead of a Scan operation. Further information on this process is in the following video.

[Modeling low velocity access patterns](#)

Avoiding sudden spikes in read activity

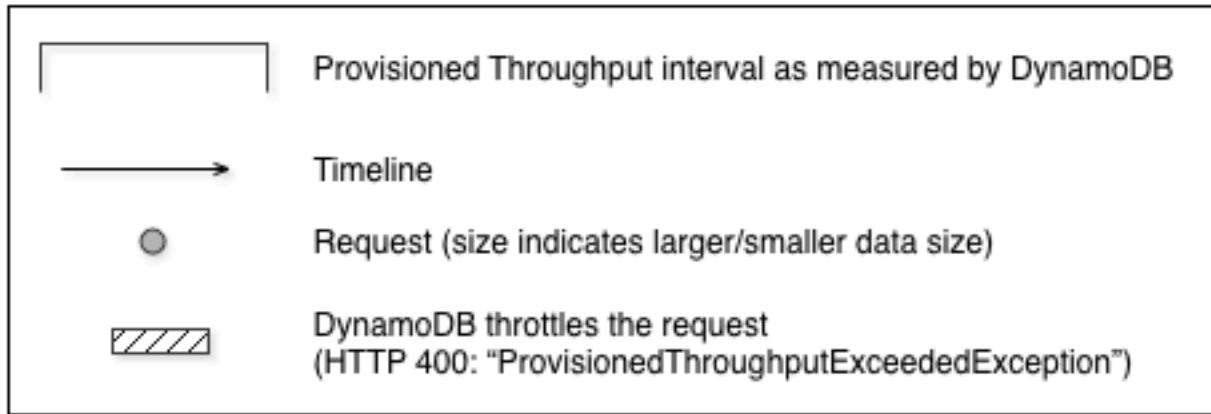
When you create a table, you set its read and write capacity unit requirements. For reads, the capacity units are expressed as the number of strongly consistent 4 KB data read requests per second. For eventually consistent reads, a read capacity unit is two 4 KB read requests per second. A Scan operation performs eventually consistent reads by default, and it can return up to 1 MB (one page) of data. Therefore, a single Scan request can consume $(1 \text{ MB page size} / 4 \text{ KB item size}) / 2$ (eventually consistent reads) = 128 read operations. If you request strongly consistent reads instead, the Scan operation would consume twice as much provisioned throughput—256 read operations.

This represents a sudden spike in usage, compared to the configured read capacity for the table. This usage of capacity units by a scan prevents other potentially more important

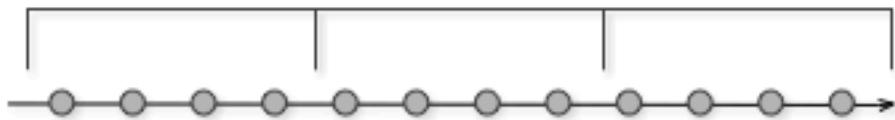
requests for the same table from using the available capacity units. As a result, you likely get a ProvisionedThroughputExceeded exception for those requests.

The problem is not just the sudden increase in capacity units that the Scan uses. The scan is also likely to consume all of its capacity units from the same partition because the scan requests read items that are next to each other on the partition. This means that the request is hitting the same partition, causing all of its capacity units to be consumed, and throttling other requests to that partition. If the request to read data is spread across multiple partitions, the operation would not throttle a specific partition.

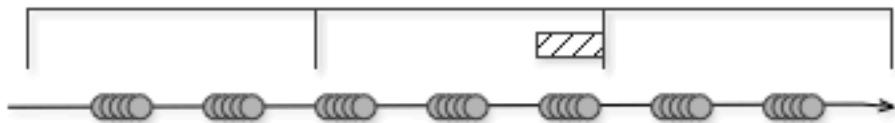
The following diagram illustrates the impact of a sudden spike of capacity unit usage by Query and Scan operations, and its impact on your other requests against the same table.



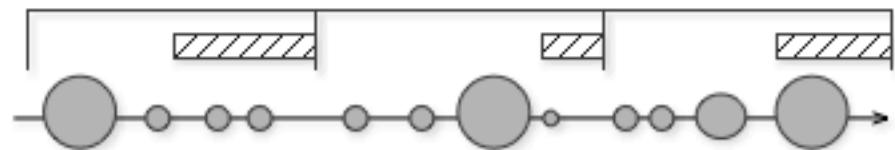
1. Good: Even distribution of requests and size



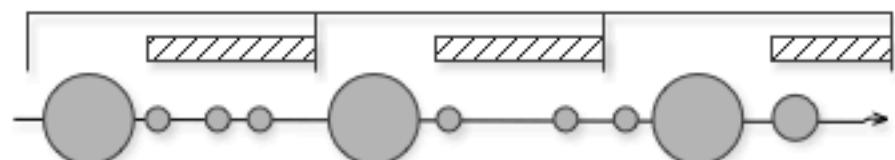
2. Not as Good: Frequent requests in bursts



3. Bad: A few random large requests



4. Bad: Large scan operations



As illustrated here, the usage spike can impact the table's provisioned throughput in several ways:

1. Good: Even distribution of requests and size

2. Not as good: Frequent requests in bursts
3. Bad: A few random large requests
4. Bad: Large scan operations

Instead of using a large Scan operation, you can use the following techniques to minimize the impact of a scan on a table's provisioned throughput.

- **Reduce page size**

Because a Scan operation reads an entire page (by default, 1 MB), you can reduce the impact of the scan operation by setting a smaller page size. The Scan operation provides a *Limit* parameter that you can use to set the page size for your request. Each Query or Scan request that has a smaller page size uses fewer read operations and creates a "pause" between each request. For example, suppose that each item is 4 KB and you set the page size to 40 items. A Query request would then consume only 20 eventually consistent read operations or 40 strongly consistent read operations. A larger number of smaller Query or Scan operations would allow your other critical requests to succeed without throttling.

- **Isolate scan operations**

DynamoDB is designed for easy scalability. As a result, an application can create tables for distinct purposes, possibly even duplicating content across several tables. You want to perform scans on a table that is not taking "mission-critical" traffic. Some applications handle this load by rotating traffic hourly between two tables—one for critical traffic, and one for bookkeeping. Other applications can do this by performing every write on two tables: a "mission-critical" table, and a "shadow" table.

Configure your application to retry any request that receives a response code that indicates you have exceeded your provisioned throughput. Or, increase the provisioned throughput for your table using the `UpdateTable` operation. If you have temporary spikes in your workload that cause your throughput to exceed, occasionally, beyond the provisioned level, retry the request with exponential backoff. For more information about implementing exponential backoff, see [Error retries and exponential backoff](#).

Taking advantage of parallel scans

Many applications can benefit from using parallel Scan operations rather than sequential scans. For example, an application that processes a large table of historical data can perform a parallel

scan much faster than a sequential one. Multiple worker threads in a background "sweeper" process could scan a table at a low priority without affecting production traffic. In each of these examples, a parallel Scan is used in such a way that it does not starve other applications of provisioned throughput resources.

Although parallel scans can be beneficial, they can place a heavy demand on provisioned throughput. With a parallel scan, your application has multiple workers that are all running Scan operations concurrently. This can quickly consume all of your table's provisioned read capacity. In that case, other applications that need to access the table might be throttled.

A parallel scan can be the right choice if the following conditions are met:

- The table size is 20 GB or larger.
- The table's provisioned read throughput is not being fully used.
- Sequential Scan operations are too slow.

Choosing TotalSegments

The best setting for TotalSegments depends on your specific data, the table's provisioned throughput settings, and your performance requirements. You might need to experiment to get it right. We recommend that you begin with a simple ratio, such as one segment per 2 GB of data. For example, for a 30 GB table, you could set TotalSegments to 15 (30 GB / 2 GB). Your application would then use 15 workers, with each worker scanning a different segment.

You can also choose a value for TotalSegments that is based on client resources. You can set TotalSegments to any number from 1 to 1000000, and DynamoDB lets you scan that number of segments. For example, if your client limits the number of threads that can run concurrently, you can gradually increase TotalSegments until you get the best Scan performance with your application.

Monitor your parallel scans to optimize your provisioned throughput use, while also making sure that your other applications aren't starved of resources. Increase the value for TotalSegments if you don't consume all of your provisioned throughput but still experience throttling in your Scan requests. Reduce the value for TotalSegments if the Scan requests consume more provisioned throughput than you want to use.

Best practices for DynamoDB table design

General design principles in Amazon DynamoDB recommend that you keep the number of tables you use to a minimum. In the majority of cases, we recommend that you consider using a single table. However if a single or small number of tables are not viable, these guidelines may be of use.

- The per account limit cannot be increased above 10,000 tables per account. If your application requires more tables, plan for distributing the tables across multiple accounts. For more information see [service, account, and table quotas in Amazon DynamoDB](#).
- Consider control plane limits for concurrent control plane operations that might impact your table management.
- Work with AWS solution architects to validate your design patterns for multi-tenant designs.

Best practices for DynamoDB global table design

Global tables build on Amazon DynamoDB's global footprint to provide you with a fully managed, multi-Region, and multi-active database that delivers fast, local, read and write performance for massively scaled, global applications. With global tables your data replicates automatically across your choice of AWS Regions. Because global tables use existing DynamoDB APIs, no changes to your application will be needed. There are no upfront costs or commitments for using global tables, and you pay only for the resources you use.

Topics

- [Prescriptive guidance for DynamoDB global table design](#)
- [Key facts about DynamoDB global table design](#)
- [Use cases](#)
- [Write modes with global tables](#)
- [Request routing with global tables](#)
- [Evacuating a Region with global tables](#)
- [Throughput capacity planning for global tables](#)
- [Preparation checklist for global tables and Frequently Asked Questions](#)

Prescriptive guidance for DynamoDB global table design

Efficient use of global tables requires careful considerations of factors like your preferred write mode, routing model, and evacuation processes. You must instrument your application across every Region and be ready to adjust your routing or perform an evacuation to maintain global health. The reward is having a globally distributed data set with low-latency reads and writes and a 99.999% service level agreement.

Key facts about DynamoDB global table design

- There are two versions of global tables: the current version [Global Tables version 2019.11.21 \(Current\)](#) (sometimes called "V2"), and [Global tables version 2017.11.29 \(Legacy\)](#) (sometimes called "V1"). This guide focuses exclusively on the current version, V2.
- Without the use of global tables, DynamoDB is a Regional service. It is highly available and intrinsically resilient to failures of infrastructure with a Region, including the failure of an entire availability zone (AZ). A single-Region DynamoDB table has a 99.99% availability [Service Level Agreement \(SLA\)](https://aws.amazon.com/dynamodb/sla/).
- With the use of global tables, DynamoDB allows a table to replicate its data between two or more Regions. A multi-Region DynamoDB table has a 99.999% availability SLA. With proper planning, global tables can help create an architecture that is resilient and resists Regional failures.
- Global tables employ an active-active replication model. From the perspective of DynamoDB, the table in each Region has equal standing to accept read and write requests. After receiving a write request, the local replica table will replicate the write to other participating Regions in the background.
- Items are replicated individually. Items updated within a single transaction may not be replicated together.
- Each table partition in the source Region replicates its writes in parallel with every other partition. The sequence of writes within the remote Region may not match the sequence of writes that happened within the source Region. For more information about table partitions, see the blog post [Scaling DynamoDB: How partitions, hot keys, and split for heat impact performance](#).
- A newly written item is usually propagated to all replica tables within a second. Nearby Regions tend to propagate faster.
- Amazon CloudWatch provides a `ReplicationLatency` metric for each Region pair. It is calculated based on looking at arriving items and comparing their arrival time with their initial

write time and computing an average. Timings are stored within CloudWatch in the source Region. Viewing the average and maximum timings can help determine the average and worst-case replication lag. There is no SLA on this latency.

- If the same item is updated at about the same time (within this `ReplicationLatency` window) in two different Regions, and the second write happens before the first write was replicated, there's a potential for write conflicts. Global tables resolves such conflicts with a *last writer wins* mechanism, based on the timestamp of the writes. The first write "loses" to the second write. These conflicts are not recorded in CloudWatch or AWS CloudTrail.
- Each item has a last write timestamp held as a private system property. The *last writer wins* approach is implemented by using a conditional write that requires the incoming item's timestamp be greater than the existing item's timestamp.
- A global table will replicate all items to all participating Regions. If you want to have different replication scopes, you can create different tables and give each of the tables different participating Regions.
- Writes will be accepted to the local Region even if the replica Region is offline or the `ReplicationLatency` grows. The local table continues to attempt replicating items to the remote table until each item succeeds.
- In the unlikely event a Region goes fully offline, when it later comes back online all pending outbound and inbound replications will be retried. No special action is required to bring the tables back in sync. The *last writer wins* mechanism ensures the data will eventually become consistent.
- You can add a new Region to a DynamoDB table at any time. DynamoDB will handle the initial sync and ongoing replication. If a Region is removed, even if it's the original Region, that will only delete the table for that Region.
- DynamoDB does not have a global endpoint. All requests are made to a regional endpoint, which then accesses the global table instance that's local to that Region.
- Calls to DynamoDB should not go cross-Region. The best practice is for the compute layer in one Region to directly access only the local DynamoDB endpoint for that Region. If problems are detected within a Region, whether those problems are in the DynamoDB layer or in the surrounding stack, then the end user traffic should be routed to a different compute layer hosted in a different Region. Thanks to global table replication, the different Region will already have a local copy of the same data for it to locally work with. In some circumstances the compute layer in one Region may pass the request onward to another Region's compute layer for processing, but this should not directly access the remote DynamoDB endpoint. For more information on this particular use case see [Compute-layer request routing](#).

Use cases

Global tables provides these common benefits:

- **Lower-latency reads.** You can place a copy of the data closer to the end user to reduce network latency during reads. The cache is kept as fresh as the `ReplicationLatency` value.
- **Lower-latency writes.** You can write to a nearby region to reduce network latency and the time taken to achieve the write. The write traffic must be carefully routed to ensure no conflicts. Techniques for routing are discussed in more detail in [Request routing with global tables](#).
- **Increased resiliency and disaster recovery.** You can evacuate a Region (move away some or all requests going to that Region) should the Region have degraded performance or a full outage, with a Recovery Point Objective (RPO) and Recovery Time Objective (RTO) measured in seconds. Using global tables also increases the [DynamoDB SLA](#) from 99.99% to 99.999%.
- **Seamless Region migration.** You can add a new Region and then delete the old Region to migrate a deployment from one Region to another, all without downtime at the data layer. For example, you can use DynamoDB global tables for an order management system achieve reliably low latency processing at a high scale while also maintaining resilience to AZ and Regional failures.

Write modes with global tables

Global tables are always active-active at the table level. However, you might want to treat them as active-passive by controlling how you route write requests. For example, you might decide to route write requests to a single Region to avoid potential write conflicts.

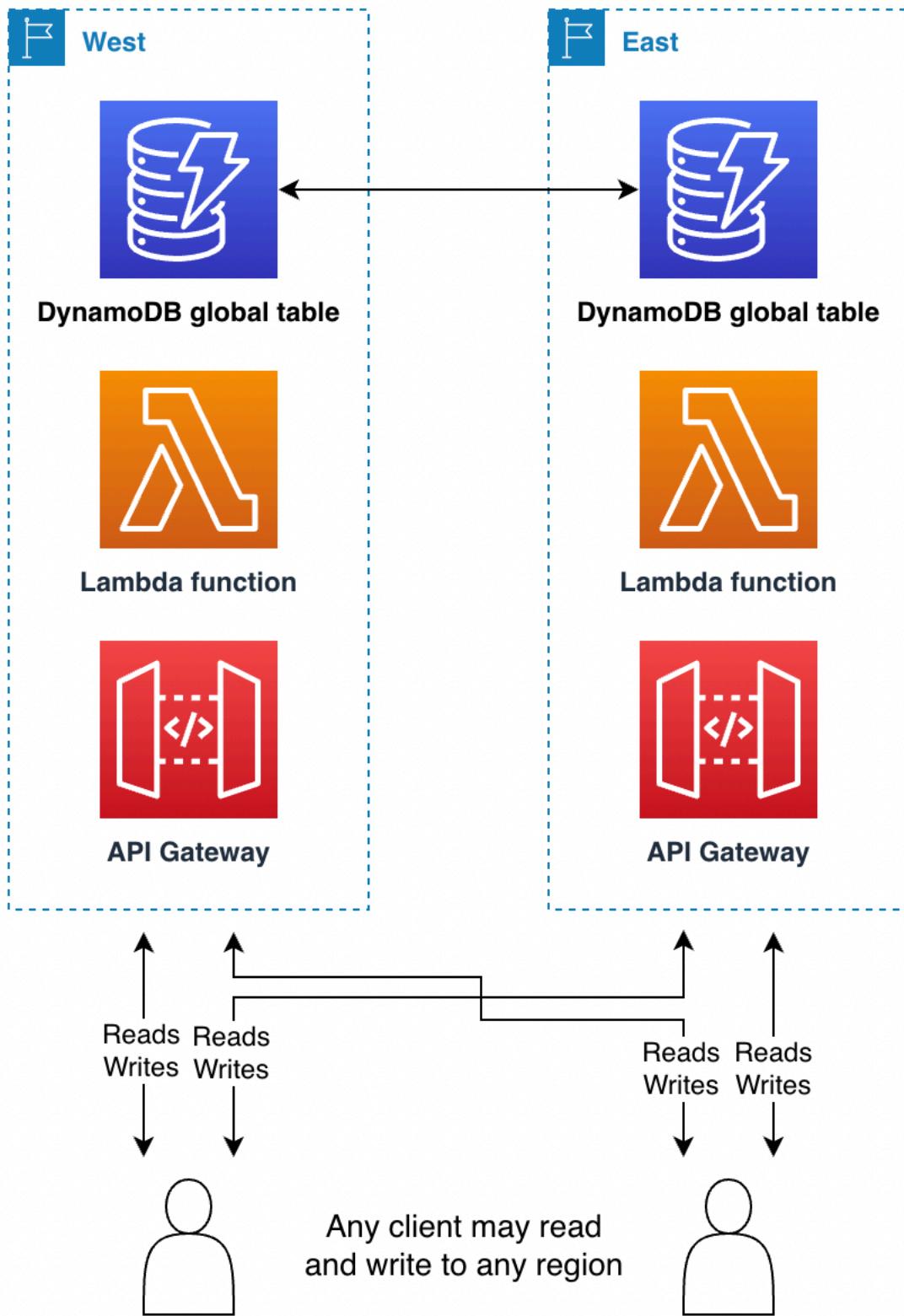
There are three main categorizations of managed write patterns:

- Write to any Region mode (no primary)
- Write to one Region mode (single primary)
- Write to your Region mode (mixed primary)

You should consider which write pattern fits your use case. This choice affects how you route requests, evacuate a Region, and handle disaster recovery. Overall best practices can differ depending on your application's write mode.

Write to any Region mode (no primary)

The *write to any Region* mode is fully active-active and doesn't impose restrictions on where a write may occur. Any Region may accept a write at any time. This is the simplest mode. It can be used only with some type of applications. It's suitable when all writers are idempotent, and therefore safely repeatable so that concurrent or repeated write operations across Regions are not in conflict. For example, when a user updates their contact data. This mode also works well for a special case of being idempotent, an append-only dataset where all writes are unique inserts under a deterministic primary key. Lastly, this mode is suitable where the risk of conflicting writes would be acceptable.



The *write to any Region* mode is the most straightforward architecture to implement. Routing is easier because any Region can be the write target at any time. Failover is easier, because any recent

writes can be replayed any number of times to any secondary Region. Where possible, you should design for this write mode.

For example, video streaming services often use global tables for tracking bookmarks, reviews, watch status flags, and so on. These deployments can use the *write to any Region* mode as long as they ensure that every write is idempotent and the next correct value for an item doesn't depend on its current value. This will be the case for user updates which assign the user's new state directly, such as setting a new latest time code, assigning a new review, or setting a new watch status. If the user's write requests are routed to different Regions, the last write operation will persist and the global state will settle according to the last assignment. Read operations in this mode will eventually become consistent, after being delayed by the latest ReplicationLatency value.

In another example, a financial services firm uses global tables as part of a system to maintain a running tally of debit card purchases for each customer, to calculate that customer's cash-back rewards. New transactions stream in from around the world and go to multiple Regions. For their current design that doesn't take advantage of global tables, they use a single RunningBalance item per customer. Customer actions update the balance with an ADD expression, which is not idempotent because the new correct value depends on the current value. This means the balance got out of sync if there were two write operations to the same balance at around the same time in different Regions.

This same firm could achieve a *write to any Region* mode through a careful redesign with DynamoDB's global tables. The new design could follow an "event streaming" model - essentially a ledger with an append-only workflow. Each customer action appends a new item to the item collection maintained for that customer. The item collection is the set of items that share a primary key, having different sort keys. Each write action that appends the cusomter action is an idempotent insert, using the customer ID as the partition key and transaction ID as the sort key. This design makes the calculation of the balance more involved, because it requires a Query to pull the items followed by some client-side math. But the advantage is that it makes all writes idempotent, which provides significant routing and failover simplifications. For more information see [Request routing with global tables](#).

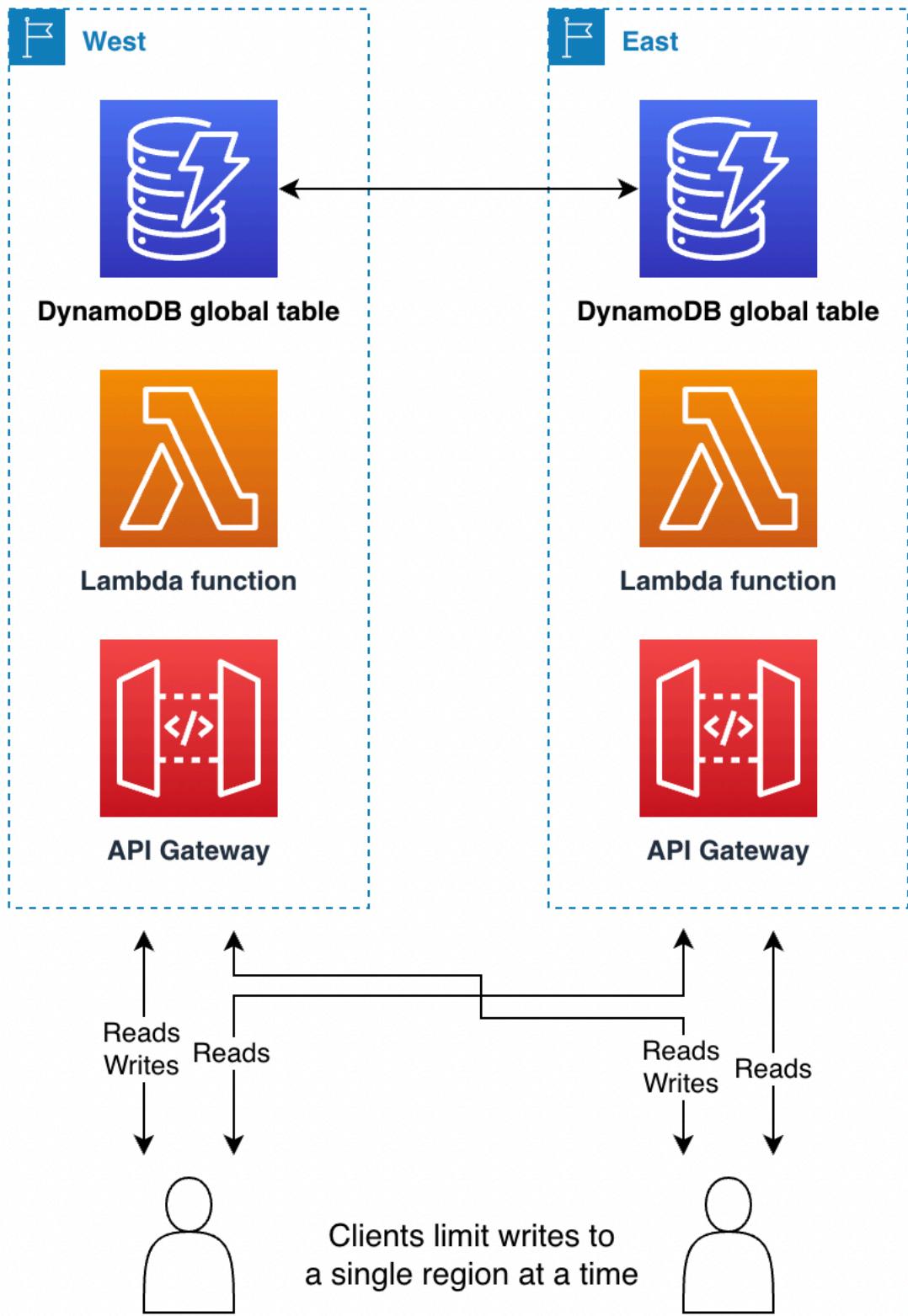
For a third example, let's say there's a customer doing online ad placement. They've decided a low risk of data loss would be acceptable to achieve the design simplifications of the *write to any Region* mode. When they serve ads, they have just a few milliseconds to retrieve enough metadata to determine what ad to show, and then to record the ad impression so the same ad won't repeated to that user. With global tables they can get both low-latency reads for end users across the world

and low-latency writes. They can record all ad impressions for a user within a single item, and represent that as a growing List. They can use one item instead of appending to an item collection, because this way they can remove older ad impressions as part of each write without paying for a delete. This write operation is NOT idempotent, so if the same end user sees ads served out of multiple Regions at approximately the same time there's a chance one ad impression write could overwrite the other. For online ad placement, the risk that a user might occasionally see a repeated ad is worth having this simpler and more efficient design.

Single primary (“Write to one Region”)

The *write to one Region* mode is active-passive and routes all table writes to a single active region. Note that DynamoDB doesn't have a notion of a single active region; the application routing outside DynamoDB manages this. The *write to one Region* mode avoids write conflicts by ensuring writes only flow to one Region at a time. This write mode is helpful when you want to use conditional expressions or transactions, because they won't work unless you know you're acting against the latest data. So using conditional expressions and transactions requires sending all write requests to the one Region with the latest data.

Eventually consistent reads can go to any replica Regions to achieve lower latencies. Strongly consistent reads must go to the single primary region.



It's sometimes necessary to change the active Region in response to a Regional failure, to help with data. [Evacuating a Region with global tables](#) is one example of this use case. Some customers will change the currently active Region on a regular schedule, such as a "follow-the-sun" deployment. This places the active Region near the geography with the most activity, giving it the lowest latency reads and writes. It also has the side benefit of calling the Region-changing code path on a daily basis, making sure it's well tested before any disaster recovery.

The passive Region(s) may keep a downscaled set of infrastructure surrounding DynamoDB that gets built up only should it become the active region. For a more in-depth discussion of pilot light and warm standby designs see [Disaster Recovery \(DR\) Architecture on AWS, Part III: Pilot Light and Warm Standby](#).

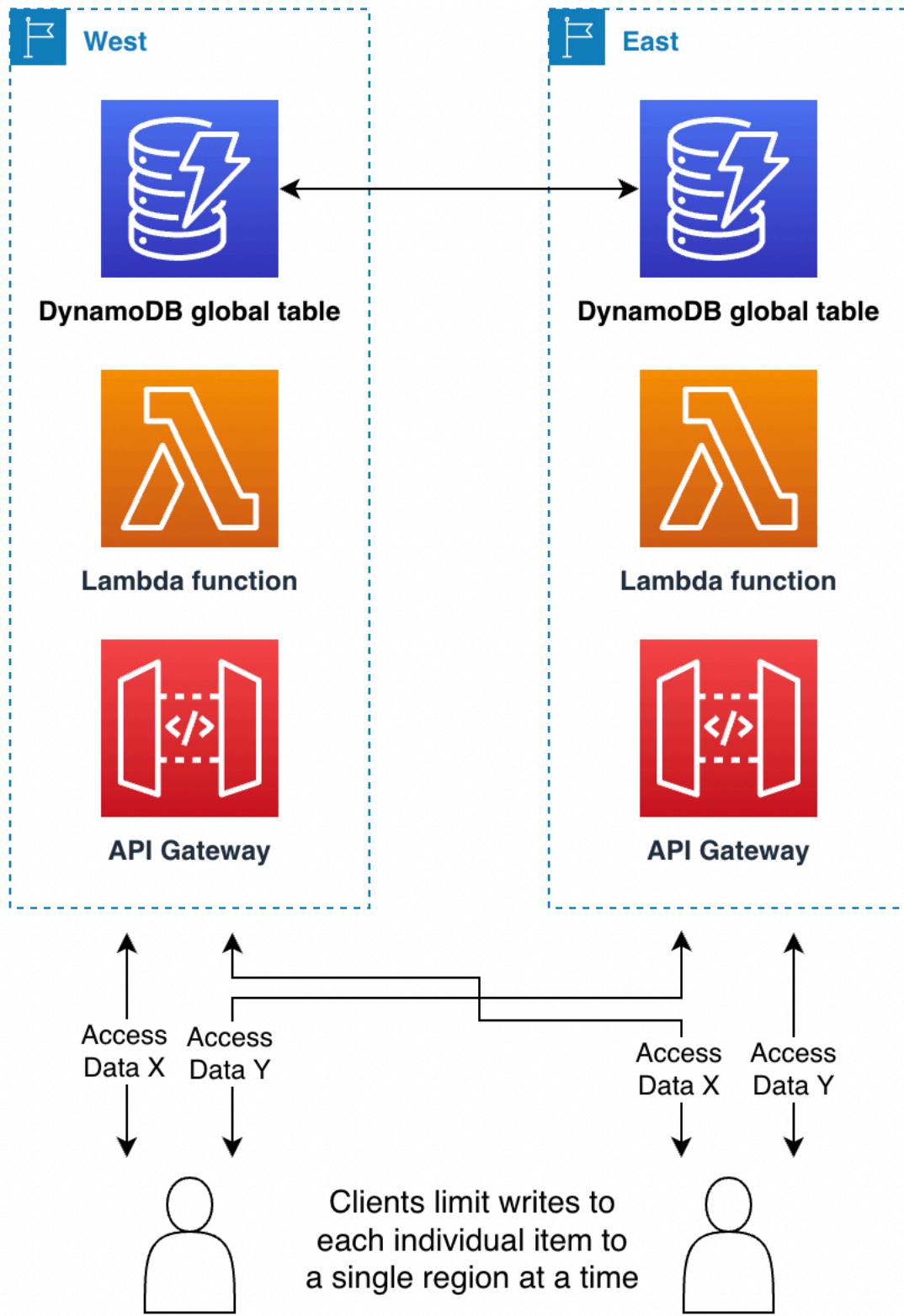
Using the *write to one Region* mode works well when leveraging global tables for low-latency globally distributed reads. For example, a large social media company has millions of users and billions of posts. Each user gets assigned to a Region at time of account creation, placed geographically near to their location. Into that non-global table goes all their data. The company uses a separate global table to hold the mapping of users to their home regions, using a *write to one Region* mode. It keeps read-only copies around the world to help locate each user's data directly with minimum added latency. Updates are rare (only when moving a user's home Region from one to another) and always go through one Region for writing, to avoid any chance of write conflicts.

As another example, consider a financial services customer who implemented a daily cash back calculation. They use *write to any Region* mode for calculating the balance but use *write to one Region* mode for tracking the actual cash back payments. If they want to reward a customer 1 penny for every \$10 spent a day, they will need to Query for all transactions from the previous day, calculate the total spent, write the cash back decision to a new table, delete the queried set of items to mark them as consumed, and replace them with a singular item storing any remainder amount that should go into the next day's calculations. This work requires transactions, and so will work better with the *write to one Region* mode. An application may mix write modes, even on the same table, as long as the workloads have no chance of overlapping.

Mixed primary ("Write to your Region")

The *write to your Region* mode assigns different data subsets to different Regions and allows write operations only to items through its home region. This mode is active-passive but assigns the active Region based on the item. Every Region is primary for its own non-overlapping data set, and writes must be guarded to ensure proper locality.

This mode is similar to *write to one Region* except that it enables lower latency writes, because the data associated with each end user can be placed in closer network proximity to that user. It also spreads the surrounding infrastructure more evenly between Regions and requires less work to build out infrastructure during a failover scenario, because all regions will have a portion of their infrastructure already active.



Determining the home region for items can be done in a variety of ways:

- **Intrinsic:** Some aspect of the data makes clear to what Region it's homed, like its partition key. For example, a customer and all data about that customer would be marked within the customer data as homed to a certain region. This technique is described in [Use Region pinning to set a home Region for items in an Amazon DynamoDB global table](#)
- **Negotiated:** The home region of each data set is negotiated in some external manner, such as with a separate global service that maintains assignments. The assignment may have a finite duration after which it's subject to renegotiation.
- **Table-oriented:** Instead of a single replicating global table, have as many global tables as there are replicating Regions. Each table's name indicates its home Region. In standard operations, all data is written to the home Region while other Regions keep a read-only copy. During a failover, another Region will temporarily adopt write duties for that table.

For example, imagine you're working for a gaming company. You need low latency reads and writes for all gamers around the world. You can home each gamer to the Region closest them. That region takes all their reads and writes, ensuring there's always strong read-after-write consistency. However, if that gamer travels or their home Region suffers an outage, a complete copy of their data will be available in alternative Regions. So the gamer can be assigned to different home Region as is useful.

As another example, imagine you're working at a video conferencing company. Each conference call's metadata gets assigned to a particular Region. Callers can use the Region that's closest to them for lowest latency. If there's a Region outage, using global tables allows quick recovery because the system can move the processing of the call to a different Region where there's already a replicated copy of the data.

Request routing with global tables

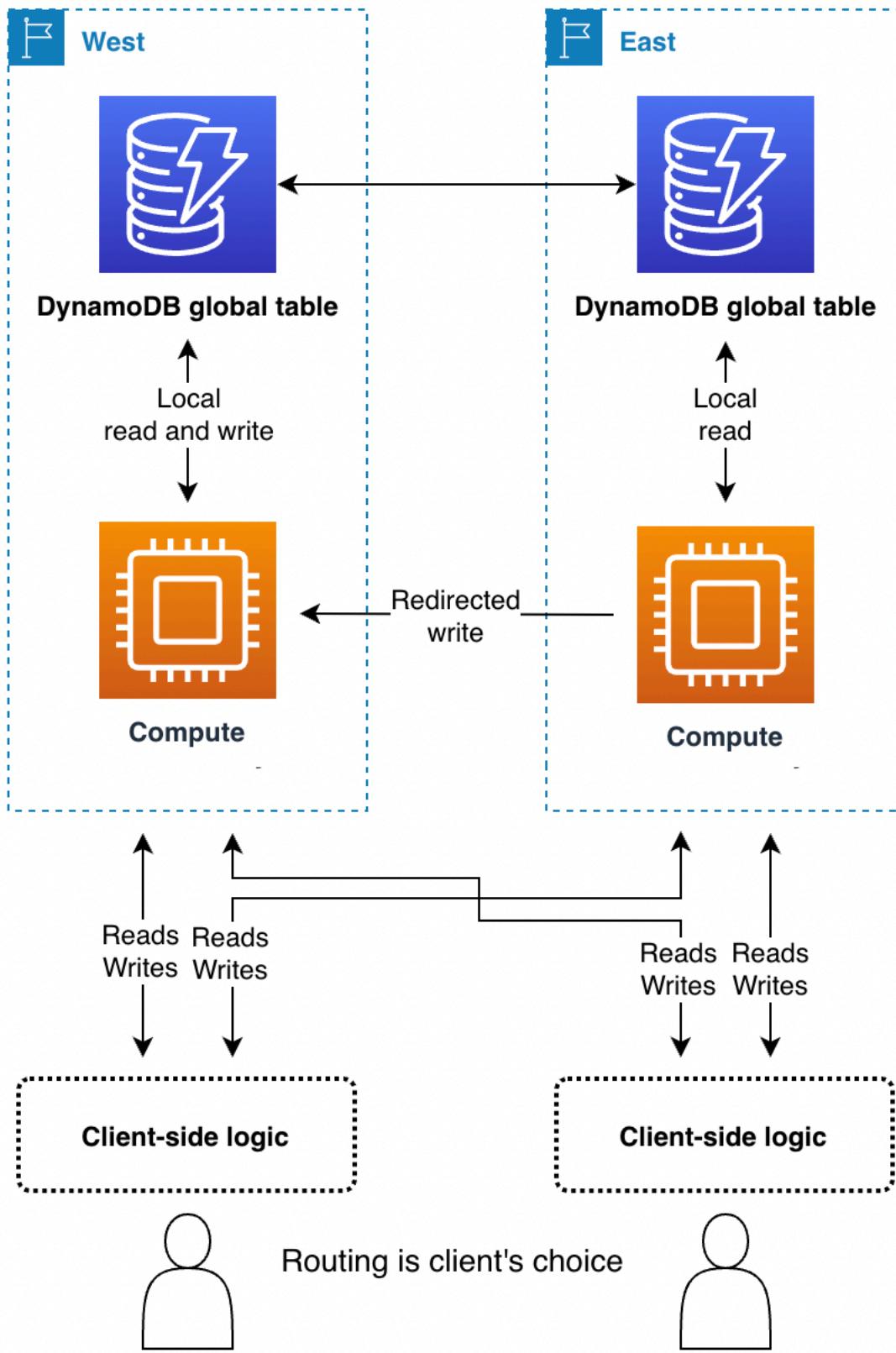
Perhaps the most complex piece of a global table deployment is managing request routing. Requests must first go from an end user to a Region that's chosen and routed in some manner. The request encounters some stack of services in that Region, including a compute layer that perhaps consists of a load balancer backed by an AWS Lambda function, container, or Amazon Elastic Compute Cloud (Amazon EC2) node, and possibly other services including another database. That compute layer communicates with DynamoDB. It should do that by using the local endpoint for that Region. The data in the global table replicates to all other participating Regions, and each Region has a similar stack of services around its DynamoDB table.

The global table provides each stack in the various Regions with a local copy of the same data. You might consider designing for a single stack in a single Region and anticipate making remote calls to a secondary Region's DynamoDB endpoint if there's an issue with the local DynamoDB table. This is not best practice. The latencies associated with going across Regions might be 100 times higher than for local access. A back-and-forth series of 5 requests might take milliseconds when performed locally but seconds when crossing the globe. It's better to route the end user to another Region for processing. To ensure resiliency you need replication across multiple Regions, with replication of the compute layer as well as the data layer.

There are numerous alternative techniques to route an end user request to a Region for processing. The optimum choice depends on your write mode and your failover considerations. This section discusses four options: client-driven, compute-layer, Route 53, and Global Accelerator.

Client-driven request routing

With client-driven request routing, an end user client such as an application, a web page with JavaScript another client will keeps track of the valid application endpoints. In this case that will be application endpoints like an Amazon API Gateway rather than literal DynamoDB endpoints. The end user client uses its own embedded logic to choose which Region to communicate with. It may choose based on random selection, lowest observed latencies, highest observed bandwidth measurements, or locally-performed health checks.



The advantage to client-driven request routing is it can be adaptive to things such as real-world public internet traffic conditions to switch Regions should it notice any degraded performance. The client must be aware of all potential endpoints, but launching a new Regional endpoint is not a frequent occurrence.

With the *write to any Region* mode, a client can unilaterally select its preferred endpoint. If its access to one Region becomes impaired, the client can route to another endpoint.

With the *write to one Region* mode, the client will need a mechanism to route its writes to the currently active region. This could be as basic as empirically testing which region is presently accepting writes (noting any write rejections and falling back to an alternate) or as complex as calling a global coordinator to query for the current application state (perhaps built on the Route 53 Application Recovery Controller (ARC) routing control which provides a 5-region quorum-driven system to maintain global state for needs such as this). The client can decide if reads can go to any Region for eventual consistency or must be routed to the active region for strong consistency. For further information see [How Route 53 works](#).

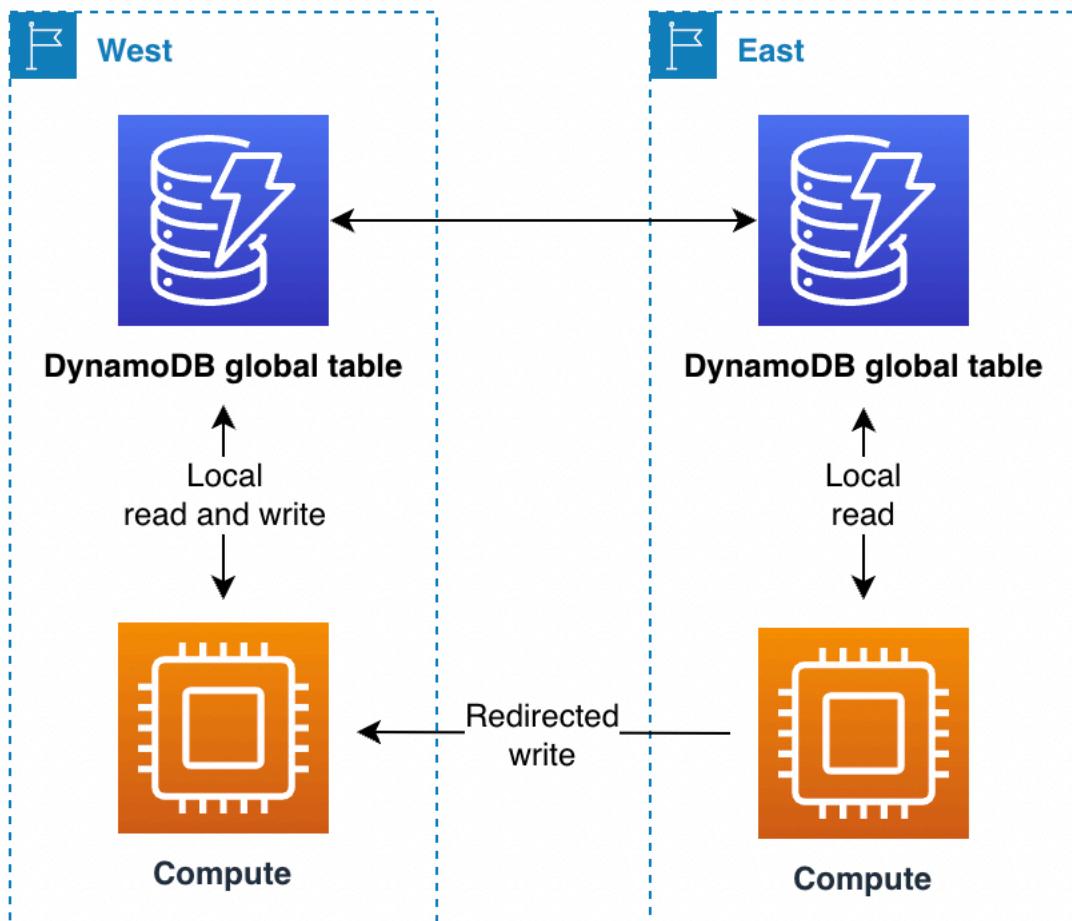
With the *write to your Region* mode, the client needs to determine the home region for the data set it's working against. For example, if the client corresponds to a user account and each user account is homed to a Region, the client can request the appropriate endpoint from a global login system.

For example, a financial services company that helps users manage their business finances via the web could use global tables with a *write to your Region* mode. Each user must login to a central service. That service returns credentials and the endpoint for the Region where those credentials will work. The credentials are valid for a short time. After that the webpage auto-negotiates a new login, which provides an opportunity to potentially redirect the user's activity to a new Region.

Compute-layer request routing

With compute-layer request routing, the code running in the compute layer decides whether it wants to process the request locally, or pass it to a copy of itself that's running in another Region. When you use the *write to one Region* mode, the compute layer may detect that it's not the active region and allow local read operations while forwarding all write operations onto another Region. This compute layer code must be aware of data topology and routing rules, and enforce them reliably based on the latest settings that specify which Regions are active for which data. The outer software stack within the Region doesn't have to be aware of how read and write requests are routed by the microservice. In a robust design, the receiving Region validates whether it is the current primary for the write operation. If it isn't, it generates an error that indicates that the

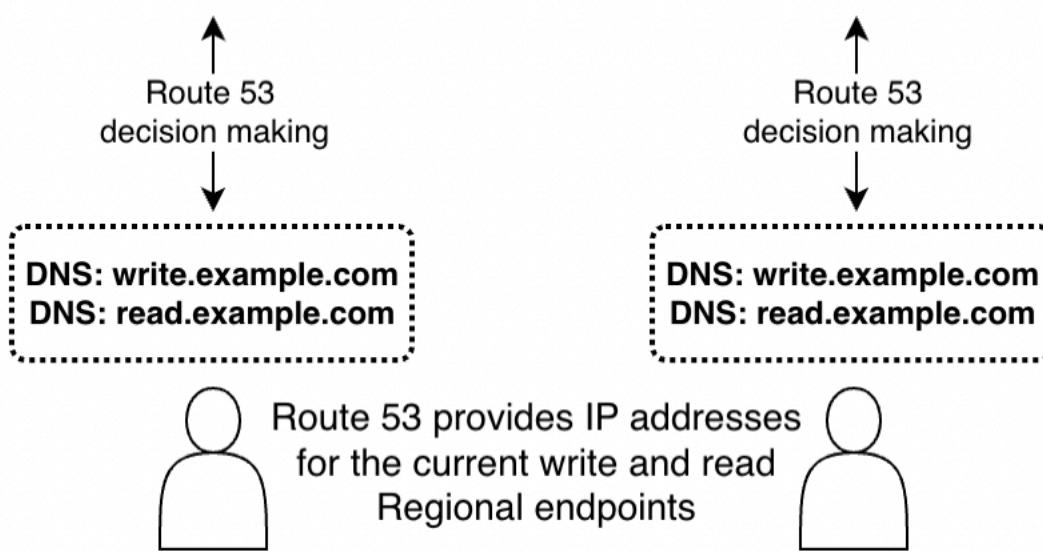
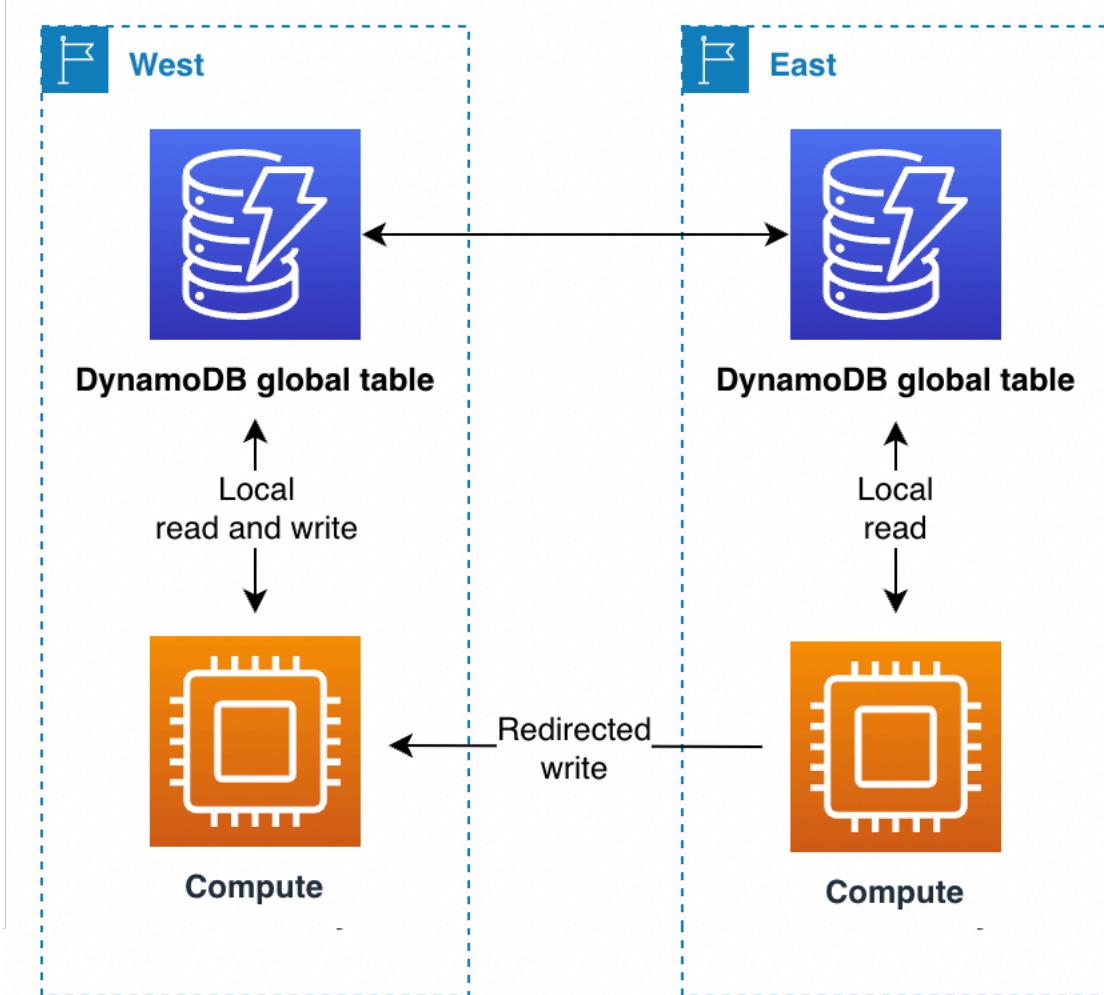
global state needs to be corrected. The receiving Region might also buffer the write operation for a while if the primary Region is in the process of changing. In all cases, the compute stack in a Region writes only to its local DynamoDB endpoint, but the compute stacks might communicate with one another.



In this scenario, let's say a financial services company uses a follow-the-sun Single Primary model. They use a system and a library for this routing process. Their overall system maintains the global state, similar to AWS's Route 53 ARC routing control. They use a global table to track which Region is the primary Region, and when the next primary switch is scheduled. All read and write operations go through the library, which coordinates with their system. The library allows read operations to be performed locally, at low latency. For write operations, the application checks if the local Region is the current primary Region. If so, the write operation completes directly. If not, the library forwards the write task to the library that's in the current primary Region. That receiving library confirms that it also considers itself the primary Region and raises an error if it isn't, which indicates a propagation delay with the global state. This approach provides a validation benefit by not writing directly to a remote DynamoDB endpoint.

Route 53 request routing

Amazon Route 53 Application Recovery Controller is a Domain Name Service (DNS) technology. With Route 53, the client requests its endpoint by looking up a well-known DNS domain name, and Route 53 returns the IP address corresponding to the regional endpoint(s) it thinks most appropriate. Route 53 has a [list of routing policies it uses to determine the appropriate region](#). Route 53 also can do [failover routing to route traffic away from regions that fail health checks](#).



- With *write to any Region* mode, or if combined with the compute-layer request routing on the backend, Route 53 can be given full access to return the Region based on any complex internal rules such as the Region in closest network proximity, or closest geographic proximity, or any other choice.
- With *write to one Region* mode, Route 53 can be configured to return the currently active region (using Route 53 ARC).

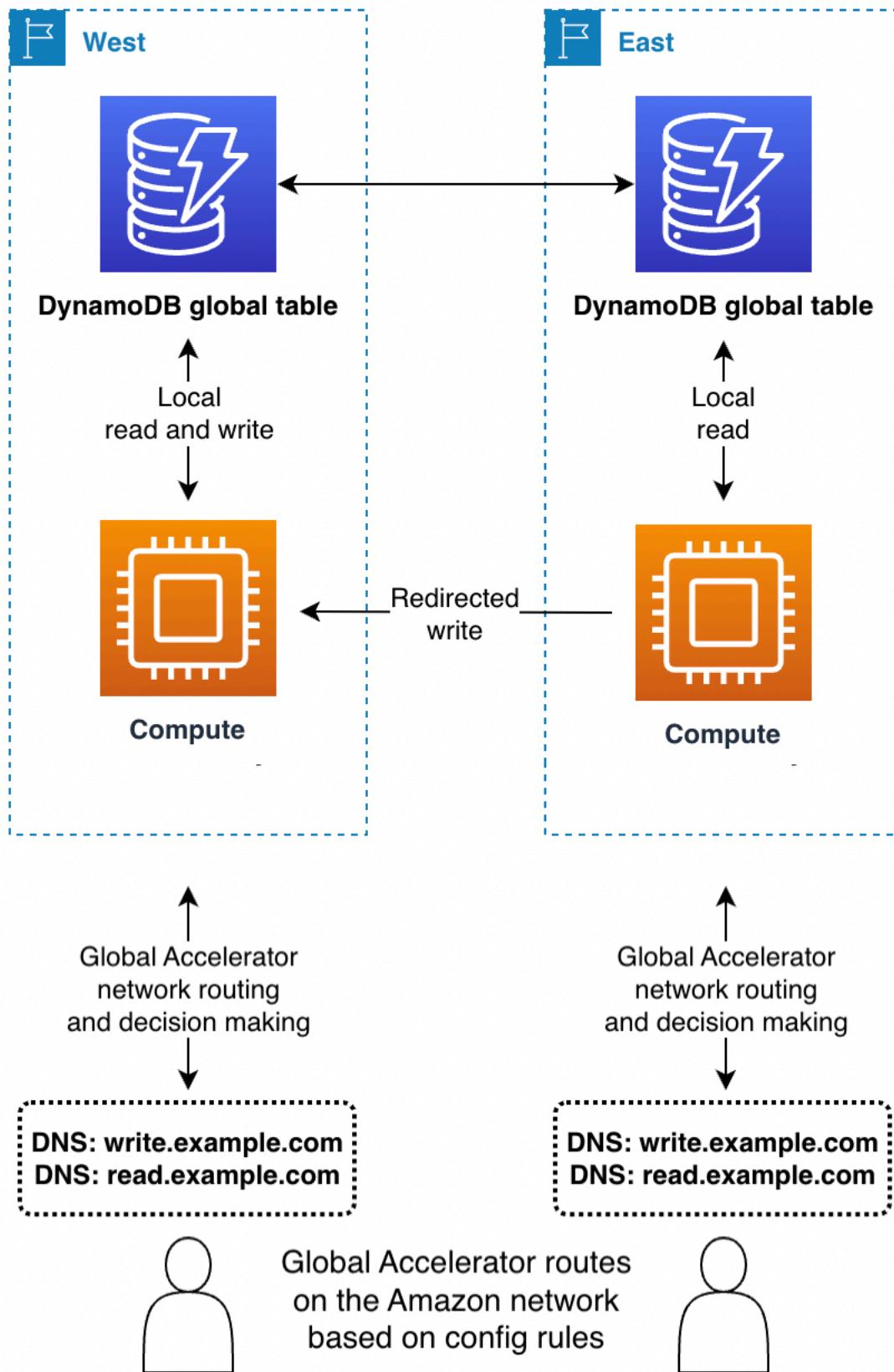
 **Note**

Clients cache the IP addresses in the response from Route 53 for a time indicated by the time to live (TTL) setting on the domain name. A longer TTL extends the recovery time objective (RTO) for all clients to recognize the new endpoint. A value of 60 seconds is typical for failover use. Not all software perfectly adheres to DNS TTL expiration.

- With *write to your Region* mode, it's best to avoid Route 53 unless you're also using compute-layer request routing.

Global Accelerator request routing

A client uses [AWS Global Accelerator](#) to looks up the well-known domain name in Route 53. However, instead of getting back an IP address that corresponds to a regional endpoint the client receives an anycast static IP address which routes to the nearest AWS edge location. Starting from that edge location, all traffic gets routed on the private AWS network and to some endpoint (such as a load balancer or API Gateway) in a Region chosen by routing rules that are maintained within Global Accelerator. Compared with routing based on Route 53 rules, Global Accelerator request routing has lower latencies because it reduces the amount of traffic on the public internet. In addition, because Global Accelerator doesn't depend on DNS TTL expiration to change routing rules it can adjust routing more quickly.



- With *write to any Region* mode, or if combined with the compute-layer request routing on the back-end, Global Accelerator works seamlessly. The client connects to the nearest edge location and need not be concerned with which Region receives the request.
- With *write to one Region* Global Accelerator routing rules must send requests to the currently active Region. You can use health checks that artificially report a failure on any Region that's not considered by your global system to be the active Region. As with DNS, it's possible to use an alternative DNS domain name for routing read requests if the requests can be from any Region.
- With *write to your Region* mode, it's best to avoid Global Accelerator unless you're also using compute-layer request routing.

Evacuating a Region with global tables

Evacuating a Region is the process of migrating read and write activity away from that Region. This is most often write activity, and sometimes read activity.

Evacuating a live Region

You might decide to evacuate a live Region for a number of reasons. The evacuation could be part of usual business activity such as if you're using a follow-the-sun write to one Region mode. The evacuation could also be due to a business decision to change the currently active Region, in response to failures in the software stack outside DynamoDB, or because you're encountering general issues such as higher than usual latencies within the Region.

With *write to any Region* mode, evacuating a live Region is straightforward. You can route traffic to the alternative Regions through any routing system, and let the write operations that have already occurred in the evacuated Region replicate as usual.

With *write to one Region* and *write to your Region* modes, you must make sure all writes to the active Region have been fully recorded, stream processed, and globally propagated before starting writes in the new active Region. This is necessary to ensure that future writes are against the latest version of the data.

Let's say that Region A is active and Region B is passive (either for the full table or for items that are homed in Region A). The typical mechanism to perform an evacuation is to pause write operations to A, wait long enough for those operations to have fully propagated to B, update the architecture stack to recognize B as active, and then resume write operations to B. There is

no metric to indicate with absolute certainty that Region A has fully replicated its data to Region B. If Region A is healthy, pausing write operations to Region A and waiting 10 times the recent maximum value of the `ReplicationLatency` metric would typically be sufficient to determine that replication is complete. If Region A is unhealthy and shows other areas of increased latencies, you would choose a larger multiple for the wait time.

Evacuating an offline Region

There's a special case to consider: what if Region A went fully offline without notice? This is extremely unlikely, but is still prudent to consider. If this happens, any write operations in Region A that were not yet propagated are held and propagated after Region A comes back online. The write operations aren't lost, but their propagation is indefinitely delayed.

How to proceed in this event is the application's decision. For business continuity, write operations might need to proceed to the new primary Region B. However, if an item in Region B receives an update while there is a pending propagation of a write operation for that item from Region A, the propagation is suppressed under the *last writer wins* model. Any update in Region B might suppress an incoming write request.

With the *write to any Region* mode, reads and writes can continue in Region B, trusting that the items in Region A will propagate to Region B eventually and recognizing the potential for missing items until Region A comes back online. When possible, you should consider replaying recent write traffic (for example, by using an upstream event source) to fill in the gap of any potentially missing write operations and let the *last writer wins* conflict resolution suppress the eventual propagation of the incoming write operation.

With the other write modes, you have to consider the degree to which work can continue with a slightly out-of-date view of the world. Some small duration of write operations, as tracked by `ReplicationLatency`, will be missing until Region A comes back online. Can business move forward? In some use cases it can, but in others it might not without additional mitigation mechanisms.

For example, imagine you need to maintain an available credit balance without interruption even after Region failure. You could split the balance into two different items, one homed in Region A and one in Region B, each starting with half the available balance. This would use the *write to your Region* mode. Transactional updates processed in each Region would write against the local copy of the balance. If Region A goes fully offline, work could still proceed with transaction processing in Region B, and write operations would be limited to the balance portion held in Region B. Splitting

the balance like this introduces complexities when the balance gets low or the credit has to be rebalanced, but it does provide one example of safe business recovery even with uncertain pending write operations.

As another example, imagine you're capturing web form data. You can use [Optimistic Concurrency Control \(OCC\)](#) to assign versions to data items and embed the latest version into the web form as a hidden field. On each submit, the write operation succeeds only if the version in the database still matches the version that the form was built against. If the versions don't match, the web form can be refreshed (or carefully merged) based on the current version in the database, and the user can proceed again. The OCC model usually protects against another client overwriting and producing a new version of the data, but it can also help during failover where a client might encounter older versions of data.

Let's imagine that you're using the timestamp as the version. Let's say that the form was first built against Region A at 12:00 but (after failover) tries to write to Region B and notices that the latest version in the database is 11:59. In this scenario, the client can either wait for the 12:00 version to propagate to Region B and then write on top of that version, or build on 11:59 and create a new 12:01 version (which, after writing, would suppress the incoming version after Region A recovers).

As a final example, a financial services company holds data about customer accounts and their financial transactions in a DynamoDB database. In the event of a complete Region A outage, they wanted to make sure that any write activity related to their accounts was either fully available in Region B, or wanted to quarantine their accounts as known partial until Region A came back online. Instead of pausing all business, they decided to pause business only to the tiny fraction of accounts that they determined had unpropagated transactions. To achieve this, they used a third Region, which we will call Region C. Before they processed any write operations in Region A, they placed a succinct summary of those pending operations (for example, a new transaction count for an account) in Region C. This summary was sufficient for Region B to determine if its view was fully up to date. This action effectively locked the account from the time of writing in Region C until Region A accepted the write operations and Region B received them. The data in Region C wasn't used except as part of a failover process, after which Region B could cross-check its data with Region C to check if any of its accounts were out of date. Those accounts would be marked as quarantined until the Region A recovery propagated the partial data to Region B.

If Region C were to fail, a new Region D could be spun up for use instead. The data in Region C was very transient, and after a few minutes Region D would have a sufficiently up-to-date record of the in-flight write operations to be fully useful. If Region B were to fail, Region A could continue accepting write requests in cooperation with Region C. This company was willing to accept higher

latency writes (to two Regions: C and then A) and was fortunate to have a data model where the state of an account could be succinctly summarized.

Throughput capacity planning for global tables

Migrating traffic from one Region to another requires careful consideration of DynamoDB table settings regarding capacity.

Some considerations on managing write capacity:

- A global table must be in on-demand mode or provisioned with auto scaling enabled.
- If provisioned with auto scaling, the write settings (minimum, maximum, and target utilization) are replicated across Regions. Although the auto scaling settings are synchronized, the actual provisioned write capacity might float independently between Regions.
- One reason you might see different provisioned write capacity is due to the TTL feature. When you enable TTL in DynamoDB, you can specify an attribute name whose value indicates the time of expiration for the item, in Unix epoch time format in seconds. After that time, DynamoDB can delete the item without incurring write costs. With global tables, you can configure TTL in any Region, and the setting is automatically replicated to other Regions that are associated with the global table. When an item is eligible for deletion through a TTL rule, that work can be done in any Region. The delete operation is performed without consuming write units on the source table, but the replica tables will get a replicated write of that delete operation and will incur replicated write unit costs.
- If you're using auto scaling, make sure that the maximum provisioned write capacity setting is sufficiently high to handle all write operations as well as all potential TTL delete operations. Auto scaling adjusts each Region according to its write consumption. On-demand tables have no maximum provisioned write capacity setting, but the *table-level maximum write throughput limit* specifies the maximum sustained write capacity the on-demand table will allow. The default limit is 40,000, but it is adjustable. We recommend that you set it high enough to handle all write operations (including TTL write operations) that the on-demand table might need. This value must be the same across all participating Regions when you set up global tables.

Some considerations on managing read capacity:

- Read capacity management settings are allowed to differ between Regions because it's assumed that different Regions might have independent read patterns. When you first add a global replica

to a table, the capacity of the source Region is propagated. After creation you can adjust the read capacity settings, which aren't transferred to the other side.

- When you use DynamoDB auto scaling, make sure that the maximum provisioned read capacity settings are sufficiently high to handle all read operations across all Regions. During standard operations the read capacity will perhaps be spread across Regions, but during failover the table should be able to automatically adapt to the increased read workload. On-demand tables have no maximum provisioned read capacity setting, but the *table-level maximum read throughput limit* specifies the maximum sustained read capacity the on-demand table will allow. The default limit is 40,000, but it is adjustable. We recommend that you set it high enough to handle all read operations that the table might need if all read operations were to route to this single Region.
- If a table in one Region doesn't usually receive read traffic but might have to absorb a large amount of read traffic after a failover, you can raise the provisioned read capacity of the table, wait for the table to finish updating, and then provision the table down again. You can either leave the table in provisioned mode or switch it to on-demand mode. This pre-warms the table for accepting a higher level of read traffic.

Route 53 ARC has [readiness checks](#) that can be useful for confirming that DynamoDB Regions have similar table settings and account quotas, whether or not you use Route 53 to route requests. These readiness checks can also help in adjusting account-level quotas to make sure they match.

Preparation checklist for global tables and Frequently Asked Questions

Use the following checklist for decisions and tasks when you deploy global tables.

- Determine how many and which Regions should participate in the global table.
- Determine your application's write mode. For more information, see [Write modes with global tables](#).
- Plan your [Request routing with global tables](#) strategy, based on your write mode.
- Define your

Evacuating a Region is the process of migrating read and write activity away from that Region. This is most often write activity, and sometimes read activity.

Evacuating a live Region

You might decide to evacuate a live Region for a number of reasons. The evacuation could be part of usual business activity such as if you're using a follow-the-sun write to one Region mode. The evacuation could also be due to a business decision to change the currently active Region, in response to failures in the software stack outside DynamoDB, or because you're encountering general issues such as higher than usual latencies within the Region.

With *write to any Region* mode, evacuating a live Region is straightforward. You can route traffic to the alternative Regions through any routing system, and let the write operations that have already occurred in the evacuated Region replicate as usual.

With *write to one Region* and *write to your Region* modes, you must make sure all writes to the active Region have been fully recorded, stream processed, and globally propagated before starting writes in the new active Region. This is necessary to ensure that future writes are against the latest version of the data.

Let's say that Region A is active and Region B is passive (either for the full table or for items that are homed in Region A). The typical mechanism to perform an evacuation is to pause write operations to A, wait long enough for those operations to have fully propagated to B, update the architecture stack to recognize B as active, and then resume write operations to B. There is no metric to indicate with absolute certainty that Region A has fully replicated its data to Region B. If Region A is healthy, pausing write operations to Region A and waiting 10 times the recent maximum value of the `ReplicationLatency` metric would typically be sufficient to determine that replication is complete. If Region A is unhealthy and shows other areas of increased latencies, you would choose a larger multiple for the wait time.

Evacuating an offline Region

There's a special case to consider: what if Region A went fully offline without notice? This is extremely unlikely, but is still prudent to consider. If this happens, any write operations in Region A that were not yet propagated are held and propagated after Region A comes back online. The write operations aren't lost, but their propagation is indefinitely delayed.

How to proceed in this event is the application's decision. For business continuity, write operations might need to proceed to the new primary Region B. However, if an item in Region B receives an update while there is a pending propagation of a write operation for that item

from Region A, the propagation is suppressed under the *last writer wins* model. Any update in Region B might suppress an incoming write request.

With the *write to any Region* mode, reads and writes can continue in Region B, trusting that the items in Region A will propagate to Region B eventually and recognizing the potential for missing items until Region A comes back online. When possible, you should consider replaying recent write traffic (for example, by using an upstream event source) to fill in the gap of any potentially missing write operations and let the *last writer wins* conflict resolution suppress the eventual propagation of the incoming write operation.

With the other write modes, you have to consider the degree to which work can continue with a slightly out-of-date view of the world. Some small duration of write operations, as tracked by `ReplicationLatency`, will be missing until Region A comes back online. Can business move forward? In some use cases it can, but in others it might not without additional mitigation mechanisms.

For example, imagine you need to maintain an available credit balance without interruption even after Region failure. You could split the balance into two different items, one homed in Region A and one in Region B, each starting with half the available balance. This would use the *write to your Region* mode. Transactional updates processed in each Region would write against the local copy of the balance. If Region A goes fully offline, work could still proceed with transaction processing in Region B, and write operations would be limited to the balance portion held in Region B. Splitting the balance like this introduces complexities when the balance gets low or the credit has to be rebalanced, but it does provide one example of safe business recovery even with uncertain pending write operations.

As another example, imagine you're capturing web form data. You can use [Optimistic Concurrency Control \(OCC\)](#) to assign versions to data items and embed the latest version into the web form as a hidden field. On each submit, the write operation succeeds only if the version in the database still matches the version that the form was built against. If the versions don't match, the web form can be refreshed (or carefully merged) based on the current version in the database, and the user can proceed again. The OCC model usually protects against another client overwriting and producing a new version of the data, but it can also help during failover where a client might encounter older versions of data.

Let's imagine that you're using the timestamp as the version. Let's say that the form was first built against Region A at 12:00 but (after failover) tries to write to Region B and notices that the latest version in the database is 11:59. In this scenario, the client can either wait for

the 12:00 version to propagate to Region B and then write on top of that version, or build on 11:59 and create a new 12:01 version (which, after writing, would suppress the incoming version after Region A recovers).

As a final example, a financial services company holds data about customer accounts and their financial transactions in a DynamoDB database. In the event of a complete Region A outage, they wanted to make sure that any write activity related to their accounts was either fully available in Region B, or wanted to quarantine their accounts as known partial until Region A came back online. Instead of pausing all business, they decided to pause business only to the tiny fraction of accounts that they determined had unpropagated transactions. To achieve this, they used a third Region, which we will call Region C. Before they processed any write operations in Region A, they placed a succinct summary of those pending operations (for example, a new transaction count for an account) in Region C. This summary was sufficient for Region B to determine if its view was fully up to date. This action effectively locked the account from the time of writing in Region C until Region A accepted the write operations and Region B received them. The data in Region C wasn't used except as part of a failover process, after which Region B could cross-check its data with Region C to check if any of its accounts were out of date. Those accounts would be marked as quarantined until the Region A recovery propagated the partial data to Region B.

If Region C were to fail, a new Region D could be spun up for use instead. The data in Region C was very transient, and after a few minutes Region D would have a sufficiently up-to-date record of the in-flight write operations to be fully useful. If Region B were to fail, Region A could continue accepting write requests in cooperation with Region C. This company was willing to accept higher latency writes (to two Regions: C and then A) and was fortunate to have a data model where the state of an account could be succinctly summarized.

evacuation plan, based on your write mode and routing strategy.

- Capture metrics on the health, latency, and errors across each Region. For a list of DynamoDB metrics, see the AWS blog post [Monitoring Amazon DynamoDB for Operational Awareness](#) for a list of metrics to observe. You should also use [synthetic canaries](#) (artificial requests designed to detect failures, named after the canary in the coal mine), as well as live observation of customer traffic. Not all issues will appear in the DynamoDB metrics.
- Set alarms for any sustained increase in ReplicationLatency. An increase might indicate an accidental misconfiguration in which the global table has different write settings in different Regions, which leads to failed replicated requests and increased latencies. It could also indicate that there is a Regional disruption. A [good example](#) is to generate an alert if the recent average

exceeds 180,000 milliseconds. You might also watch for `ReplicationLatency` dropping to 0, which indicates stalled replication. .

- Assign sufficient maximum read and write settings for each global table.
- Identify the reasons for evacuating a Region in advance. If the decision involves human judgment, document all considerations. This work should be done carefully in advance, not under stress.
- Maintain a runbook for every action that must take place when you evacuate a Region. Usually very little work is involved for the global tables, but moving the rest of the stack might be complex.

 **Note**

It's best practice to rely only on data plane operations and not control plane operations because some control plane operations may be degraded during region failures.

For more information, see the AWS blog post [Build resilient applications with Amazon DynamoDB global tables: Part 4](#).

- Test all aspects of the runbook periodically, including Region evacuations. An untested runbook is an unreliable runbook.
- Consider using Resilience Hub to evaluate the resilience of your entire application (including global tables). It provides a comprehensive view of your overall application portfolio resilience status through its dashboard.
- Consider using Route 53 ARC readiness checks to evaluate the current configuration of your application and track any deviances from best practices.
- When writing a health check for use with Route 53 or Global Accelerator, it's not sufficient to just ping that the DynamoDB endpoint is up. That doesn't cover the many failure modes such as IAM configuration errors, code deployment problems, failure in the stack outside DynamoDB, higher than average read or write latencies, and so on. It's best to perform a set of calls that exercise a full database flow.

Frequently Asked Questions (FAQ) for deploying global tables

What are some useful principles for overall usage of DynamoDB global tables?

DynamoDB global tables has very few control knobs yet still requires a number of considerations. You must determine your write mode, routing model, and evacuation processes. You must instrument your application across every Region and be ready to adjust your routing or perform an evacuation to maintain global health. The reward is having a globally distributed data set with low-latency reads and writes and a 99.999% service level agreement.

What is the pricing for global tables?

A write to a traditional DynamoDB table is priced in Write Capacity Units (WCUs, for provisioned tables) or Write Request Units (WRUs, for on-demand tables). If you write a 5 KB item it incurs a charge of 5 units. A write to a global table is priced in Replicated Write Capacity Units (rWCUs, for provisioned tables) or Replicated Write Request Units (rWRUs, for on-demand tables).

The rWCUs and rWRUs include the cost of the streaming infrastructure needed to manage the replication. As such, they are priced 50% higher than WCUs and WRUs. Cross-Region data transfer fees do apply.

Replicated Write Unit charges are incurred in every Region where the item is directly written or replicate written.

Writing to a Global Secondary Index (GSI) is considered a local write and uses regular Write Units.

There is no Reserved Capacity available for rWCUs at this time. Purchasing Reserved Capacity may still be beneficial for tables with GSIs consuming write units.

The initial bootstrap when adding a new Region to a global table is charged like a restore per GB of data restored, plus cross-Region data transfer fees.

Which Regions does global tables support?

[Global Tables version 2019.11.21 \(Current\)](#) is available in most Regions. You can see the most recent list in the Region dropdown list on the DynamoDB console when you add a replica.

How are GSIs handled with global tables?

In [Global Tables version 2019.11.21 \(Current\)](#), when you create a GSI in one Region it's automatically created in other participating Regions and automatically backfilled.

How do I stop replication of a global table?

You can delete a replica table the same way you would delete any other table. Deleting the global table stops replication to that Region and deletes the table copy kept in that Region. However,

you cannot stop replication while keeping copies of the table as independent entities, nor can you pause replication.

How do DynamoDB Streams interact with global tables?

Each global table produces an independent stream based on all its writes, wherever they started from. You can choose to consume the DynamoDB stream in one Region or in all Regions (independently). If you want to process local but not replicated write operations, you can add your own Region attribute to each item to identify the writing Region. You can then use a Lambda event filter to call the Lambda function only for write operations in the local Region. This helps with insert and update operations, but unfortunately not delete operations.

How do global tables handle transactions?

Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the Region where the write operation originally occurred. Transactions are not supported across Regions in global tables. For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a `TransactWriteItems` operation in the US East (Ohio) Region, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

How do global tables interact with the DynamoDB Accelerator cache (DAX)?

Global tables bypass DAX by updating DynamoDB directly, so DAX isn't aware that it's holding stale data. The DAX cache is refreshed only when the cache's TTL expires.

Do tags on tables propagate?

No, tags do not automatically propagate.

Should I backup tables in all Regions or just one?

The answer depends on the purpose of the backup. If you want to ensure data durability, DynamoDB already provides that safeguard. The service ensures durability. If you want to keep a snapshot for historical records (for example, to meet regulatory requirements), backing up in one Region should suffice. You can copy the backup to additional Regions by using AWS Backup. If you want to recover erroneously deleted or modified data, use [DynamoDB point-in-time recovery \(PITR\)](#) in one Region.

How do I deploy global tables using AWS CloudFormation?

CloudFormation represents a DynamoDB table and a global table as two separate resources: AWS::DynamoDB::Table and AWS::DynamoDB::GlobalTable. One approach is to create all tables that can potentially be global by using the GlobalTable construct. You can then keep them as standalone tables initially, and add Regions later if needed.

In CloudFormation, each global table is controlled by a single stack, in a single Region, regardless of the number of replicas. When you deploy your template, CloudFormation creates and updates all replicas as part of a single stack operation. You should not deploy the same [AWS::DynamoDB::GlobalTable](#) resource in multiple Regions. This will result in errors and is unsupported. If you deploy your application template in multiple Regions, you can use conditions to create the AWS::DynamoDB::GlobalTable resource in a single Region. Alternatively, you can choose to define your AWS::DynamoDB::GlobalTable resources in a stack that's separate from your application stack, and make sure that it's deployed to a single Region.

If you have a regular table and you want to convert it to a global table while keeping it managed by CloudFormation then set the deletion policy to Retain, remove the table from the stack, convert the table to a global table in the console, and then import the global table as a new resource to the stack.

Cross-account replication is not supported at this time.

Best practices for managing the control plane in DynamoDB

Note

DynamoDB is introducing a control plane throttle limit of 2,500 requests per second with the option for a retry. See below for additional details.

DynamoDB control plane operations let you manage DynamoDB tables as well as objects that are dependent on tables such as indexes. For more information about these operations, see [Control plane](#).

In some circumstances, you may need to take actions and use data returned by control plane calls as part of your business logic. For example, you might need to know the value of ProvisionedThroughput returned by `DescribeTable`. In these circumstances, follow these best practices:

- Do not excessively query the DynamoDB control plane.

- Do not mix control plane calls and data plane calls within the same code.
- Handle throttles on control plane requests and retry with a backoff.
- Invoke and track changes to a particular resource from a single client.
- Instead of retrieving data for the same table multiple times at short intervals, cache the data for processing.

Best Practices for Understanding your AWS Billing and Usage Reports

This document explains the `UsageType` billing codes for charges related to DynamoDB.

AWS provides cost and usage reports (CUR) that contain data for the services used. You can use AWS Cost and Usage Report to publish billing reports to Amazon S3 in a CSV format. When setting up the CUR you can choose to break time periods down by hour, day, or month, and you can choose if you want to break out usage by resource ID or not. For more details on generating CUR, please see [Creating Cost and Usage Reports](#)

Within the CSV export, you will find relevant attributes listed for each line. The following are examples of attributes that may be included:

- **lineitem/UsageStartDate:** The start date and time for the line item in UTC, inclusive.
- **lineitem/UsageEndDate:** The end date and time for the corresponding line item in UTC, exclusive.
- **lineitem/ProductCode:** For DynamoDB this will be "AmazonDynamoDB"
- **lineitem/UsageType:** A specific description code for the type of usage, as enumerated in this document
- **lineitem/Operation:** A name that provides context to the charge such as the operation name that incurred the charge (optional).
- **lineitem/ResourceId:** The identifier for the resource that incurred the usage. Available if the CUR includes a breakdown by resource ID.
- **lineitem/UsageAmount:** The amount of usage incurred during the specified time period.
- **lineitem/UnblendedCost:** The cost of this usage.
- **lineitem/LineItemDescription:** Textual description of the line item.

For more information about the CUR data dictionary, see [Cost and Usage Report \(CUR\) 2.0](#). Note that the exact names vary depending on context.

A UsageType is a string with a value such as ReadCapacityUnit-Hrs, USW2-ReadRequestUnits, EU-WriteCapacityUnit-Hrs, or USE1-TimedPITRStorage-ByteHrs. Each usage type begins with an optional Region prefix. If absent, that indicates the us-east-1 Region. If present, the below table maps the short billing Region code to the conventional Region code and name.

For example, the usage named USW2-ReadRequestUnits indicates read request units consumed in us-west-2.

Billing Region Code	Region Code	Region Name
AFS1	af-south-1	Africa (Cape Town)
APE1	ap-east-1	Asia Pacific (Hong Kong)
APN1	ap-northeast-1	Asia Pacific (Tokyo)
APN2	ap-northeast-2	Asia Pacific (Seoul)
APN3	ap-northeast-3	Asia Pacific (Osaka)
APS1	ap-south-1	Asia Pacific (Mumbai)
APS2	ap-south-2	Asia Pacific (Hyderabad)
APS3	ap-southeast-1	Asia Pacific (Singapore)
APS4	ap-southeast-2	Asia Pacific (Sydney)
APS5	ap-southeast-3	Asia Pacific (Jakarta)
APS6	ap-southeast-4	Asia Pacific (Melbourne)
CAN1	ca-central-1	Canada (Central)
EU	eu-central-1	Europe (Frankfurt)
EUC1	eu-central-2	Europe (Zurich)

Billing Region Code	Region Code	Region Name
EUN1	eu-north-1	Europe (Stockholm)
EUS1	eu-south-1	Europe (Milan)
EUS2	eu-south-2	Europe (Spain)
EUW1	eu-west-1	Europe (Ireland)
EUW2	eu-west-2	Europe (London)
EUW3	eu-west-3	Europe (Paris)
ILC1	il-central-1	Israel (Tel Aviv)
MEC1	me-central-1	Middle East (UAE)
MES1	me-south-1	Middle East (Bahrain)
SAE1	sa-east-1	South America (São Paulo)
USE1 (default)	us-east-1	US East (N. Virginia)
USE2	us-east-2	US East (Ohio)
UGE1	us-gov-east-1	US Government East
UGW1	us-gov-west-1	US Government West
USW1	us-west-1	US West (N. California)
USW2	us-west-2	US West (Oregon)

In the following sections, we use REG-UsageType pattern when going through the charges for DynamoDB, where REG specifies the region where usage occurred and usageType is the code for the type of charge. For example if you see a line item for USW1- ReadCapacityUnit-Hrs in your CSV file, that means the usage was incurred in US-West-1 for provisioned read capacity. In that case the listing would say REG-ReadCapacityUnit-Hrs.

Topics

- [Throughput Capacity](#)
- [Streams](#)
- [Storage](#)
- [Backup and Restore](#)
- [Data Transfer](#)
- [CloudWatch Contributor Insights](#)
- [DynamoDB Accelerator \(DAX\)](#)

Throughput Capacity

Provisioned Capacity Reads and Writes

When you create a DynamoDB table in provisioned capacity mode, you specify the read and write capacity that you expect your application to require. The usage type depends on your table class (Standard or Standard-Infrequent Access). You provision read and writes based on consumption rate per second, but the charges are priced per hour based on provisioned capacity.

UsageType	Units	Granularity	Description
REG-ReadCapacityUnit-Hrs	RCU-hours	Hour	Charges for reads in provisioned capacity mode using the Standard table class.
REG-IA-ReadCapacityUnit-Hrs	RCU-hours	Hour	Charges for reads in provisioned capacity mode using the Standard-IA table class.
REG-WriteCapacityUnit-Hrs	WCU-hours	Hour	Charges for writes in provisioned capacity mode using the Standard table class.
REG-IA-WriteCapacityUnit-Hrs	WCU-hours	Hour	Charges for writes in provisioned capacity

UsageType	Units	Granularity	Description
			mode using the Standard-IA table class.

Reserved Capacity Reads and Writes

With reserved capacity, you pay a one-time upfront fee and commit to a minimum provisioned usage level over a period of time. Reserved capacity is billed at a discounted hourly rate. Any capacity that you provision in excess of your reserved capacity is billed at standard provisioned capacity rates. Reserved capacity is available for single-region, provisioned read and write capacity units (RCU and WCU) on DynamoDB tables that use the standard table class. Both 1-year and 3-year reserved capacity are billed using the same SKUs.

UsageType	Units	Granularity	Description
REG-HeavyUsage:dynamodb.read	RCU-hours	Up-front then monthly	Charges for reserved capacity reads: a one-time up-front charge and a monthly charge at the start of each month covering all the discounted committed RCU-hours during the month. Will have matching zero-cost REG-ReadCapacityUnit-Hrs line items.
REG-HeavyUsage:dynamodb.write	WCUs	Up-front then monthly	Charges for reserved capacity writes: a one-time up-front charge and a monthly charge at the start of each month covering

UsageType	Units	Granularity	Description
			all the discounted committed WCU-hours during the month. Will have matching zero-cost REG-WriteCapacityUnit-Hrs line items.

On-Demand Capacity Reads and Writes

When you create a DynamoDB table in on-demand capacity mode, you pay only for the reads and writes your application performs. The prices for read and write requests depend on your table class.

UsageType	Units	Granularity	Description
REG-ReadRequestUnits	RRUs	Unit	Charges for reads in on-demand capacity mode with Standard table class.
REG-IA-ReadRequest Units	RRUs	Unit	Charges for reads in on-demand capacity mode with Standard-IA table class.
REG-WriteRequestUnits	WRUs	Unit	Charges for writes in on-demand capacity mode with Standard table class.
REG-IA-WriteRequestUnits	WRUs	Unit	Charges for writes in on-demand capacity mode with Standard-IA table class.

Global Tables Reads and Writes

DynamoDB charges for global tables usage based on the resources used on each replica table. For provisioned global tables, write requests for global tables are measured in replicated WCUs (rWCUs) instead of standard WCUs and writes to global secondary indexes in global tables are measured in WCUs. For on-demand global tables, write requests are measured in replicated WRUs (rWRUs) instead of standard WRUs. The number of rWCUs or rWRUs consumed for replication depends on the version of global tables you are using. The pricing depends on your table class.

Writes to global secondary indexes (GSIs) are billed using standard write units (WCUs and WRUs). Read requests and data storage are billed identically to single-region tables.

If you add a table replica to create or extend a global table in new Regions, DynamoDB charges for a table restore in the added Regions per gigabyte of data restored. Restored Data is charged as REG-RestoreContentSize-Bytes. Please refer to [Using On-Demand backup and restore for DynamoDB](#) for details. Cross-Region replication and adding replicas to tables that contain data also incur charges for data transfer out.

When you select on-demand capacity mode for your DynamoDB global tables, you pay only for the resources your application uses on each replica table.

UsageType	Units	Granularity	Description
REG-ReplWriteCapacityUnit-Hrs	rWCU-hours	Hour	Global table, provisioned, Standard table class.
REG-IA-ReplWriteCapacityUnit-Hrs	rWCU-hours	Hour	Global table, provisioned, Standard-IA table class.
REG-ReplWriteRequestUnits	rWRU	Unit	Global table, on-demand, Standard table class.
REG-IA-ReplWriteRequestUnits	rWRU	Unit	Global table, on-demand, Standard- IA table class

Streams

DynamoDB has two streaming technologies, DynamoDB Streams and Kinesis. Each have separate pricing.

DynamoDB Streams charges for reading data in read request units. Each GetRecords API call is billed as a streams read request. You are not charged for GetRecords API calls invoked by AWS Lambda as part of DynamoDB triggers or by DynamoDB global tables as part of replication.

UsageType	Units	Granularity	Description
REG-Streams-RequestsCount	Count	Unit	Read request units for DynamoDB Streams.

Amazon Kinesis Data Streams charges in change data capture units. DynamoDB charges one change data capture unit for each write (up to 1 KB). For items larger than 1 KB, additional change data capture units are required. You pay only for the writes your application performs without having to manage throughput capacity on the table.

UsageType	Units	Granularity	Description
REG-ChangeDataCaptureUnits-Kinesis	CDC Units	Unit	Change data capture units for Kinesis Data Streams.

Storage

DynamoDB measures the size of your billable data by adding the raw byte size of your data plus a per-item storage overhead that depends on the features you have enabled.

Note

Storage usage values in the CUR will be higher compared with the storage values when using `DescribeTable`, because `DescribeTable` does not include the per-item storage overhead.

Storage is calculated hourly but priced monthly as calculated from an average of the hourly charges.

Although the storage UsageType uses ByteHrs as a suffix, storage usage in the CUR is measured in GB and priced by GB-month.

UsageType	Units	Granularity	Description
REG-TimedStorage-B yteHrs	GB	Month	Amount of storage used by your DynamoDB tables and indexes, for tables with the Standard table class.
REG-IA-TimedStorage- ByteHrs	GB	Month	Amount of storage used by your DynamoDB tables and indexes, for tables with the Standard-IA table class.

Backup and Restore

DynamoDB offers two types of backups: Point In Time Recovery (PITR) backups and on-demand backups. Users can also restore from those backups into DynamoDB tables. The charges below refers to both backups and restores.

Backup storage charges are incurred on the first of the month with adjustments made throughout the month as backups are added or removed. See the [Understanding Amazon DynamoDB On-demand Backups and Billing](#) blog for more information

UsageType	Units	Granularity	Description
REG-TimedBackupStorage-ByteHrs	GB	Month	The storage consumed by on-

UsageType	Units	Granularity	Description
			demand backups of your DynamoDB tables and Local Secondary Indexes.
TimedPITRStorage-ByteHrs	GB	Month	The storage used by point-in-time recovery (PITR) backups. DynamoDB monitors the size of your PITR-enabled tables continuously throughout the month to determine your backup charges and bills for storage as long as PITR is enabled.
REG-RestoreDataSize-Bytes	GB	Size	The total size of data restored (including table data, local secondary indexes and global secondary indexes) measured in GB from DynamoDB backups.

AWS Backup

AWS Backup is a fully managed backup service that makes it easy to centralize and automate the backup of data across AWS services in the cloud as well as on premises. AWS Backup is charged for storage (warm or cold storage), restoration activities, and cross-Region data transfer. The following UsageType charges appear under the "AWSBackup" ProductCode rather than "AmazonDynamoDB".

UsageType	Units	Granularity	Description
REG-WarmStorage-ByteHrs-DynamoDB	GB	Month	The storage used by DynamoDB backups managed by AWS Backup throughout the month, measured in GB-Month.
REG-CrossRegion-WarmBytes-DynamoDB	GB	Size	The data transferred to a different AWS Region either within the same account or to a different AWS account. Cross-Region transfers charges occur when copying backups from one Region to another Region. The charge is always billed to the account where the data is transferred from.
REG-Restore-WarmBytes-DynamoDB	GB	Size	The total size of the data restored from warm storage, measured in GB.
REG-ColdStorage-ByteHrs-DynamoDB	GB	Month	The cold storage used by DynamoDB backups managed by AWS Backup throughout the month, measured in GB-Month.

UsageType	Units	Granularity	Description
REG-Restore-ColdBytes-DynamoDB	GB	Month	The total size of the data restored from cold storage, measured in GB.

Export and Import

You can export data from DynamoDB to Amazon S3 or import data from Amazon S3 to a new DynamoDB table.

Although the UsageType uses Bytes as a suffix, export and import usage in the CUR is measured and priced in GB.

UsageType	Units	Granularity	Description
REG-ExportDataSize-Bytes	GB	Size	The charge for exporting data to S3. DynamoDB charges for data you export based on the size of the DynamoDB base table (table data and local secondary indexes) at the specified point in time when the export was created.
REG-ImportDataSize-Bytes	GB	Size	The charge for importing data from S3. The size is calculated based on the uncompressed object size of the data within Amazon

UsageType	Units	Granularity	Description
			S3. There are no extra charges for importing to tables with GSIs.
REG-IncrementalExportDataSize-Bytes	GB	Size	The charge for size of the data processed from the continuous backup to produce incremental exports.

Data Transfer

Data transfer activity may appear associated with the DynamoDB service. DynamoDB does not charge for inbound data transfer, and it does not charge for data transferred between DynamoDB and other AWS services within the same AWS Region (in other words, \$0.00 per GB). Data transferred across AWS Regions (such as between DynamoDB in the US East [N. Virginia] Region and Amazon EC2 in the EU [Ireland] Region) is charged on both sides of the transfer.

UsageType	Units	Granularity	Description
REG-DataTransfer-In-Bytes	GB	Units	Data transferred in to DynamoDB from the internet.
REG-DataTransfer-Out-Bytes	GB	Units	Data transferred out from DynamoDB to the internet.

CloudWatch Contributor Insights

CloudWatch Contributor Insights for DynamoDB is a diagnostic tool for identifying the most frequently accessed and throttled keys in your DynamoDB table. The following UsageType charges appear under the "AmazonCloudWatch" ProductCode rather than "AmazonDynamoDB".

UsageType	Units	Granularity	Description
REG-CW:Contributor EventsManaged	Events processed	Units	The amount of DynamoDB events processed. For example for a table with CloudWatch Contributor Insights enabled, anytime an item is read or written, it's counted as one event. If the table has a sort key, it results in charges for two events.
REG-CW:Contributor RulesManaged	Rule count	Month	DynamoDB creates rules to identify most accessed items and most throttled keys when you enable Cloud Watch Contributor Insights. This charge is incurred for the rules added for each entity (tables and GSIs) configured for logging CloudWatch contributor insights.

DynamoDB Accelerator (DAX)

DynamoDB Accelerator (DAX) is billed by the hour based on the instance type selected for the service. The charges below refers to the DynamoDB Accelerator instances provisioned.

The following UsageType charges appear under the “AmazonDAX” ProductCode rather than “AmazonDynamoDB”.

UsageType	Units	Granularity	Description
REG-NodeUsage:dax-<INSTANCETYPE>	Node-hour	Hour	The hourly usage of a particular instance type. Pricing is per node-hour consumed, from the time a node is launched until it is terminated. Each partial node-hour consumed will be billed as a full hour. DAX charges for each node in a DAX cluster. If you have a cluster with multiple nodes, you would see multiple line items in your billing report.

The instance type will be a value such as one from the following table. For details about node types, refer to [Nodes](#).

<INSTANCETYPE>		
r3.2xlarge	r4.8xlarge	r5.8xlarge
r3.4xlarge	r4.large	r5.large
r3.8xlarge	r4.xlarge	r5.xlarge
r3.2xlarge	r5.12xlarge	t2.medium
r3.4xlarge	r4.large	r5.large

<INSTANCETYPE>		
r3.xlarge	r5.16xlarge	t2.small
r4.16xlarge	r5.24xlarge	t3.medium
r4.2xlarge	r5.2xlarge	t3.small
r4.4xlarge	r5.4xlarge	

Using DynamoDB with other AWS services

Amazon DynamoDB is integrated with other AWS services, letting you automate repeating tasks or build applications that span multiple services.

Topics

- [Configuring AWS credentials in your files using Amazon Cognito](#)
- [Loading data from DynamoDB into Amazon Redshift](#)
- [Processing DynamoDB data with Apache Hive on Amazon EMR](#)
- [Integrating with Amazon S3](#)
- [DynamoDB zero-ETL integration with Amazon OpenSearch Service](#)
- [Best practices for integrating with DynamoDB](#)

Configuring AWS credentials in your files using Amazon Cognito

The recommended way to obtain AWS credentials for your web and mobile applications is to use Amazon Cognito. Amazon Cognito helps you avoid hardcoding your AWS credentials on your files. It uses AWS Identity and Access Management (IAM) roles to generate temporary credentials for your application's authenticated and unauthenticated users.

For example, to configure your JavaScript files to use an Amazon Cognito unauthenticated role to access the Amazon DynamoDB web service, do the following.

To configure credentials to integrate with Amazon Cognito

1. Create an Amazon Cognito identity pool that allows unauthenticated identities.

```
aws cognito-identity create-identity-pool \
    --identity-pool-name DynamoPool \
    --allow-unauthenticated-identities \
    --output json
{
    "IdentityPoolId": "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
    "AllowUnauthenticatedIdentities": true,
    "IdentityPoolName": "DynamoPool"
```

{

2. Copy the following policy into a file named `myCognitoPolicy.json`. Replace the identity pool ID (`us-west-2:12345678-1ab2-123a-1234-a12345ab12`) with your own IdentityPoolId obtained in the previous step.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Federated": "cognito-identity.amazonaws.com"  
            },  
            "Action": "sts:AssumeRoleWithWebIdentity",  
            "Condition": {  
                "StringEquals": {  
                    "cognito-identity.amazonaws.com:aud": "us-west-2:12345678-1ab2-123a-1234-a12345ab12"  
                },  
                "ForAnyValue:StringLike": {  
                    "cognito-identity.amazonaws.com:amr": "unauthenticated"  
                }  
            }  
        }  
    ]  
}
```

3. Create an IAM role that assumes the previous policy. In this way, Amazon Cognito becomes a trusted entity that can assume the `Cognito_DynamoPoolUnauth` role.

```
aws iam create-role --role-name Cognito_DynamoPoolUnauth \  
--assume-role-policy-document file://PathToFile/myCognitoPolicy.json --output json
```

4. Grant the `Cognito_DynamoPoolUnauth` role full access to DynamoDB by attaching a managed policy (`AmazonDynamoDBFullAccess`).

```
aws iam attach-role-policy --policy-arn arn:aws:iam::aws:policy/  
AmazonDynamoDBFullAccess \  
--role-name Cognito_DynamoPoolUnauth
```

Note

Alternatively, you can grant fine-grained access to DynamoDB. For more information, see [Using IAM policy conditions for fine-grained access control](#).

5. Obtain and copy the IAM role Amazon Resource Name (ARN).

```
aws iam get-role --role-name Cognito_DynamoPoolUnauth --output json
```

6. Add the Cognito_DynamoPoolUnauth role to the DynamoPool identity pool. The format to specify is KeyName=string, where KeyName is unauthenticated and the string is the role ARN obtained in the previous step.

```
aws cognito-identity set-identity-pool-roles \  
--identity-pool-id "us-west-2:12345678-1ab2-123a-1234-a12345ab12" \  
--roles unauthenticated=arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth --  
output json
```

7. Specify the Amazon Cognito credentials in your files. Modify the IdentityPoolId and RoleArn accordingly.

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({  
  IdentityPoolId: "us-west-2:12345678-1ab2-123a-1234-a12345ab12",  
  RoleArn: "arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth"  
});
```

You can now run your JavaScript programs against the DynamoDB web service using Amazon Cognito credentials. For more information, see [Setting credentials in a web browser](#) in the *AWS SDK for JavaScript Getting Started Guide*.

Loading data from DynamoDB into Amazon Redshift

Amazon Redshift complements Amazon DynamoDB with advanced business intelligence capabilities and a powerful SQL-based interface. When you copy data from a DynamoDB table into Amazon Redshift, you can perform complex data analysis queries on that data, including joins with other tables in your Amazon Redshift cluster.

In terms of provisioned throughput, a copy operation from a DynamoDB table counts against that table's read capacity. After the data is copied, your SQL queries in Amazon Redshift do not affect DynamoDB in any way. This is because your queries act upon a copy of the data from DynamoDB, rather than upon DynamoDB itself.

Before you can load data from a DynamoDB table, you must first create an Amazon Redshift table to serve as the destination for the data. Keep in mind that you are copying data from a NoSQL environment into a SQL environment, and that there are certain rules in one environment that do not apply in the other. Here are some of the differences to consider:

- DynamoDB table names can contain up to 255 characters, including '.' (dot) and '-' (dash) characters, and are case-sensitive. Amazon Redshift table names are limited to 127 characters, cannot contain dots or dashes and are not case-sensitive. In addition, table names cannot conflict with any Amazon Redshift reserved words.
- DynamoDB does not support the SQL concept of NULL. You need to specify how Amazon Redshift interprets empty or blank attribute values in DynamoDB, treating them either as NULLs or as empty fields.
- DynamoDB data types do not correspond directly with those of Amazon Redshift. You need to ensure that each column in the Amazon Redshift table is of the correct data type and size to accommodate the data from DynamoDB.

Here is an example COPY command from Amazon Redshift SQL:

```
copy favoritemovies from 'dynamodb://my-favorite-movies-table'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-
Access-Key>'
readratio 50;
```

In this example, the source table in DynamoDB is `my-favorite-movies-table`. The target table in Amazon Redshift is `favoritemovies`. The `readratio 50` clause regulates the percentage of provisioned throughput that is consumed; in this case, the COPY command will use no more than 50 percent of the read capacity units provisioned for `my-favorite-movies-table`. We highly recommend setting this ratio to a value less than the average unused provisioned throughput.

For detailed instructions on loading data from DynamoDB into Amazon Redshift, refer to the following sections in the [Amazon Redshift Database Developer Guide](#):

- [Loading data from a DynamoDB table](#)

- [The COPY command](#)
- [COPY examples](#)

Processing DynamoDB data with Apache Hive on Amazon EMR

Amazon DynamoDB is integrated with Apache Hive, a data warehousing application that runs on Amazon EMR. Hive can read and write data in DynamoDB tables, allowing you to:

- Query live DynamoDB data using a SQL-like language (HiveQL).
- Copy data from a DynamoDB table to an Amazon S3 bucket, and vice-versa.
- Copy data from a DynamoDB table into Hadoop Distributed File System (HDFS), and vice-versa.
- Perform join operations on DynamoDB tables.

Topics

- [Overview](#)
- [Tutorial: Working with Amazon DynamoDB and Apache Hive](#)
- [Creating an external table in Hive](#)
- [Processing HiveQL statements](#)
- [Querying data in DynamoDB](#)
- [Copying data to and from Amazon DynamoDB](#)
- [Performance tuning](#)

Overview

Amazon EMR is a service that makes it easy to quickly and cost-effectively process vast amounts of data. To use Amazon EMR, you launch a managed cluster of Amazon EC2 instances running the Hadoop open source framework. *Hadoop* is a distributed application that implements the MapReduce algorithm, where a task is mapped to multiple nodes in the cluster. Each node processes its designated work, in parallel with the other nodes. Finally, the outputs are reduced on a single node, yielding the final result.

You can choose to launch your Amazon EMR cluster so that it is persistent or transient:

- A *persistent* cluster runs until you shut it down. Persistent clusters are ideal for data analysis, data warehousing, or any other interactive use.

- A *transient* cluster runs long enough to process a job flow, and then shuts down automatically. Transient clusters are ideal for periodic processing tasks, such as running scripts.

For information about Amazon EMR architecture and administration, see the [Amazon EMR Management Guide](#).

When you launch an Amazon EMR cluster, you specify the initial number and type of Amazon EC2 instances. You also specify other distributed applications (in addition to Hadoop itself) that you want to run on the cluster. These applications include Hue, Mahout, Pig, Spark, and more.

For information about applications for Amazon EMR, see the [Amazon EMR Release Guide](#).

Depending on the cluster configuration, you might have one or more of the following node types:

- Leader node — Manages the cluster, coordinating the distribution of the MapReduce executable and subsets of the raw data, to the core and task instance groups. It also tracks the status of each task performed and monitors the health of the instance groups. There is only one leader node in a cluster.
- Core nodes — Runs MapReduce tasks and stores data using the Hadoop Distributed File System (HDFS).
- Task nodes (optional) — Runs MapReduce tasks.

Tutorial: Working with Amazon DynamoDB and Apache Hive

In this tutorial, you will launch an Amazon EMR cluster, and then use Apache Hive to process data stored in a DynamoDB table.

Hive is a data warehouse application for Hadoop that allows you to process and analyze data from multiple sources. Hive provides a SQL-like language, *HiveQL*, that lets you work with data stored locally in the Amazon EMR cluster or in an external data source (such as Amazon DynamoDB).

For more information, see to the [Hive Tutorial](#).

Topics

- [Before you begin](#)
- [Step 1: Create an Amazon EC2 key pair](#)
- [Step 2: Launch an Amazon EMR cluster](#)

- [Step 3: Connect to the Leader node](#)
- [Step 4: Load data into HDFS](#)
- [Step 5: Copy data to DynamoDB](#)
- [Step 6: Query the data in the DynamoDB table](#)
- [Step 7: \(Optional\) clean up](#)

Before you begin

For this tutorial, you will need the following:

- An AWS account. If you do not have one, see [Signing up for AWS](#).
- An SSH client (Secure Shell). You use the SSH client to connect to the leader node of the Amazon EMR cluster and run interactive commands. SSH clients are available by default on most Linux, Unix, and Mac OS X installations. Windows users can download and install the [PuTTY](#) client, which has SSH support.

Next step

[Step 1: Create an Amazon EC2 key pair](#)

In this step, you will create the Amazon EC2 key pair you need to connect to an Amazon EMR leader node and run Hive commands.

1. Sign in to the AWS Management Console and open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Choose a region (for example, US West (Oregon)). This should be the same region in which your DynamoDB table is located.
3. In the navigation pane, choose **Key Pairs**.
4. Choose **Create Key Pair**.
5. In **Key pair name**, type a name for your key pair (for example, mykeypair), and then choose **Create**.
6. Download the private key file. The file name will end with . pem (such as mykeypair.pem). Keep this private key file in a safe place. You will need it to access any Amazon EMR cluster that you launch with this key pair.

⚠ Important

If you lose the key pair, you cannot connect to the leader node of your Amazon EMR cluster.

For more information about key pairs, see [Amazon EC2 Key Pairs](#) in the *Amazon EC2 User Guide for Linux Instances*.

Next step[Step 2: Launch an Amazon EMR cluster](#)**Step 2: Launch an Amazon EMR cluster**

In this step, you will configure and launch an Amazon EMR cluster. Hive and a storage handler for DynamoDB will already be installed on the cluster.

1. Open the Amazon EMR console at <https://console.aws.amazon.com/emr>.
2. Choose **Create Cluster**.
3. On the **Create Cluster - Quick Options** page, do the following:
 - a. In **Cluster name**, type a name for your cluster (for example: My EMR cluster).
 - b. In **EC2 key pair**, choose the key pair you created earlier.

Leave the other settings at their defaults.

4. Choose **Create cluster**.

It will take several minutes to launch your cluster. You can use the **Cluster Details** page in the Amazon EMR console to monitor its progress.

When the status changes to **Waiting**, the cluster is ready for use.

Cluster log files and Amazon S3

An Amazon EMR cluster generates log files that contain information about the cluster status and debugging information. The default settings for **Create Cluster - Quick Options** include setting up Amazon EMR logging.

If one does not already exist, the AWS Management Console creates an Amazon S3 bucket.

The bucket name is `aws-logs-account-id-region`, where `account-id` is your AWS account number and `region` is the region in which you launched the cluster (for example, `aws-logs-123456789012-us-west-2`).

Note

You can use the Amazon S3 console to view the log files. For more information, see [View Log Files](#) in the *Amazon EMR Management Guide*.

You can use this bucket for purposes in addition to logging. For example, you can use the bucket as a location for storing a Hive script or as a destination when exporting data from Amazon DynamoDB to Amazon S3.

Next step

[Step 3: Connect to the Leader node](#)

Step 3: Connect to the Leader node

When the status of your Amazon EMR cluster changes to **Waiting**, you will be able to connect to the leader node using SSH and perform command line operations.

1. In the Amazon EMR console, choose your cluster's name to view its status.
2. On the **Cluster Details** page, find the **Leader public DNS** field. This is the public DNS name for the leader node of your Amazon EMR cluster.
3. To the right of the DNS name, choose the **SSH** link.
4. Follow the instructions in **Connect to the Leader Node Using SSH**.

Depending on your operating system, choose the **Windows** tab or the **Mac/Linux** tab, and follow the instructions for connecting to the leader node.

After you connect to the leader node using either SSH or PuTTY, you should see a command prompt similar to the following:

```
[hadoop@ip-192-0-2-0 ~]$
```

Next step

[Step 4: Load data into HDFS](#)

Step 4: Load data into HDFS

In this step, you will copy a data file into Hadoop Distributed File System (HDFS), and then create an external Hive table that maps to the data file.

Download the sample data

1. Download the sample data archive (`features.zip`):

```
wget https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/features.zip
```

2. Extract the `features.txt` file from the archive:

```
unzip features.zip
```

3. View the first few lines of the `features.txt` file:

```
head features.txt
```

The result should look similar to this:

```
1535908|Big Run|Stream|WV|38.6370428|-80.8595469|794  
875609|Constable Hook|Cape|NJ|40.657881|-74.0990309|7  
1217998|Gooseberry Island|Island|RI|41.4534361|-71.3253284|10  
26603|Boone Moore Spring|Spring|AZ|34.0895692|-111.410065|3681  
1506738|Missouri Flat|Flat|WA|46.7634987|-117.0346113|2605  
1181348|Minnow Run|Stream|PA|40.0820178|-79.3800349|1558  
1288759|Hunting Creek|Stream|TN|36.343969|-83.8029682|1024  
533060|Big Charles Bayou|Bay|LA|29.6046517|-91.9828654|0  
829689|Greenwood Creek|Stream|NE|41.596086|-103.0499296|3671  
541692|Button Willow Island|Island|LA|31.9579389|-93.0648847|98
```

The features.txt file contains a subset of data from the United States Board on Geographic Names (http://geonames.usgs.gov/domestic/download_data.htm). The fields in each line represent the following:

- Feature ID (unique identifier)
- Name
- Class (lake; forest; stream; and so on)
- State
- Latitude (degrees)
- Longitude (degrees)
- Height (in feet)

4. At the command prompt, enter the following command:

```
hive
```

The command prompt changes to this: hive>

5. Enter the following HiveQL statement to create a native Hive table:

```
CREATE TABLE hive_features
  (feature_id          BIGINT,
   feature_name        STRING ,
   feature_class       STRING ,
   state_alpha         STRING,
   prim_lat_dec       DOUBLE ,
   prim_long_dec      DOUBLE ,
   elev_in_ft          BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
LINES TERMINATED BY '\n';
```

6. Enter the following HiveQL statement to load the table with data:

```
LOAD DATA
LOCAL
INPATH './features.txt'
OVERWRITE
INTO TABLE hive_features;
```

7. You now have a native Hive table populated with data from the `features.txt` file. To verify, enter the following HiveQL statement:

```
SELECT state_alpha, COUNT(*)  
FROM hive_features  
GROUP BY state_alpha;
```

The output should show a list of states and the number of geographic features in each.

Next step

[Step 5: Copy data to DynamoDB](#)

Step 5: Copy data to DynamoDB

In this step, you will copy data from the Hive table (`hive_features`) to a new table in DynamoDB.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. On the **Create DynamoDB table** page, do the following:
 - a. In **Table**, type **Features**.
 - b. For **Primary key**, in the **Partition key** field, type **Id**. Set the data type to **Number**.

Clear **Use Default Settings**. For **Provisioned Capacity**, type the following:

- **Read Capacity Units**—10
- **Write Capacity Units**—10

Choose **Create**.

4. At the Hive prompt, enter the following HiveQL statement:

```
CREATE EXTERNAL TABLE ddb_features  
(feature_id    BIGINT,  
 feature_name   STRING,  
 feature_class  STRING,  
 state_alpha    STRING,  
 prim_lat_dec  DOUBLE,
```

```
    prim_long_dec DOUBLE,  
    elev_in_ft     BIGINT)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES(  
    "dynamodb.table.name" = "Features",  
  
    "dynamodb.column.mapping"="feature_id:Id,feature_name:Name,feature_class:Class,state_alpha:  
);
```

You have now established a mapping between Hive and the Features table in DynamoDB.

- Enter the following HiveQL statement to import data to DynamoDB:

```
INSERT OVERWRITE TABLE ddb_features  
SELECT  
    feature_id,  
    feature_name,  
    feature_class,  
    state_alpha,  
    prim_lat_dec,  
    prim_long_dec,  
    elev_in_ft  
FROM hive_features;
```

Hive will submit a MapReduce job, which will be processed by your Amazon EMR cluster. It will take several minutes to complete the job.

- Verify that the data has been loaded into DynamoDB:

- In the DynamoDB console navigation pane, choose **Tables**.
- Choose the Features table, and then choose the **Items** tab to view the data.

Next step

[Step 6: Query the data in the DynamoDB table](#)

In this step, you will use HiveQL to query the Features table in DynamoDB. Try the following Hive queries:

- All of the feature types (feature_class) in alphabetical order:

```
SELECT DISTINCT feature_class  
FROM ddb_features  
ORDER BY feature_class;
```

2. All of the lakes that begin with the letter "M":

```
SELECT feature_name, state_alpha  
FROM ddb_features  
WHERE feature_class = 'Lake'  
AND feature_name LIKE 'M%'  
ORDER BY feature_name;
```

3. States with at least three features higher than a mile (5,280 feet):

```
SELECT state_alpha, feature_class, COUNT(*)  
FROM ddb_features  
WHERE elev_in_ft > 5280  
GROUP by state_alpha, feature_class  
HAVING COUNT(*) >= 3  
ORDER BY state_alpha, feature_class;
```

Next step

Step 7: (Optional) clean up

Step 7: (Optional) clean up

Now that you have completed the tutorial, you can continue reading this section to learn more about working with DynamoDB data in Amazon EMR. You might decide to keep your Amazon EMR cluster up and running while you do this.

If you don't need the cluster anymore, you should terminate it and remove any associated resources. This will help you avoid being charged for resources you don't need.

1. Terminate the Amazon EMR cluster:
 - a. Open the Amazon EMR console at <https://console.aws.amazon.com/emr>.
 - b. Choose the Amazon EMR cluster, choose **Terminate**, and then confirm.
2. Delete the Features table in DynamoDB:

- a. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
 - b. In the navigation pane, choose **Tables**.
 - c. Choose the Features table. From the **Actions** menu, choose **Delete Table**.
3. Delete the Amazon S3 bucket containing the Amazon EMR log files:
- a. Open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
 - b. From the list of buckets, choose aws-logs- *accountID-region*, where *accountID* is your AWS account number and *region* is the region in which you launched the cluster.
 - c. From the **Action** menu, choose **Delete**.

Creating an external table in Hive

In [Tutorial: Working with Amazon DynamoDB and Apache Hive](#), you created an external Hive table that mapped to a DynamoDB table. When you issued HiveQL statements against the external table, the read and write operations were passed through to the DynamoDB table.

You can think of an external table as a pointer to a data source that is managed and stored elsewhere. In this case, the underlying data source is a DynamoDB table. (The table must already exist. You cannot create, update, or delete a DynamoDB table from within Hive.) You use the `CREATE EXTERNAL TABLE` statement to create the external table. After that, you can use HiveQL to work with data in DynamoDB, as if that data were stored locally within Hive.

Note

You can use `INSERT` statements to insert data into an external table and `SELECT` statements to select data from it. However, you cannot use `UPDATE` or `DELETE` statements to manipulate data in the table.

If you no longer need the external table, you can remove it using the `DROP TABLE` statement. In this case, `DROP TABLE` only removes the external table in Hive. It does not affect the underlying DynamoDB table or any of its data.

Topics

- [CREATE EXTERNAL TABLE syntax](#)
- [Data type mappings](#)

CREATE EXTERNAL TABLE syntax

The following shows the HiveQL syntax for creating an external Hive table that maps to a DynamoDB table:

```
CREATE EXTERNAL TABLE hive_table
  (hive_column1_name hive_column1_datatype, hive_column2_name hive_column2_datatype...)
  STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
  TBLPROPERTIES (
    "dynamodb.table.name" = "dynamodb_table",
    "dynamodb.column.mapping" =
    "hive_column1_name:dynamodb_attribute1_name,hive_column2_name:dynamodb_attribute2_name..."
  );
```

Line 1 is the start of the CREATE EXTERNAL TABLE statement, where you provide the name of the Hive table (*hive_table*) you want to create.

Line 2 specifies the columns and data types for *hive_table*. You need to define columns and data types that correspond to the attributes in the DynamoDB table.

Line 3 is the STORED BY clause, where you specify a class that handles data management between the Hive and the DynamoDB table. For DynamoDB, STORED BY should be set to 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'.

Line 4 is the start of the TBLPROPERTIES clause, where you define the following parameters for DynamoDBStorageHandler:

- dynamodb.table.name—the name of the DynamoDB table.
- dynamodb.column.mapping—pairs of column names in the Hive table and their corresponding attributes in the DynamoDB table. Each pair is of the form *hive_column_name*:*dynamodb_attribute_name*, and the pairs are separated by commas.

Note the following:

- The name of the Hive table name does not have to be the same as the DynamoDB table name.
- The Hive table column names do not have to be the same as those in the DynamoDB table.
- The table specified by dynamodb.table.name must exist in DynamoDB.

- For dynamodb.column.mapping:

- You must map the key schema attributes for the DynamoDB table. This includes the partition key and the sort key (if present).
- You do not have to map the non-key attributes of the DynamoDB table. However, you will not see any data from those attributes when you query the Hive table.
- If the data types of a Hive table column and a DynamoDB attribute are incompatible, you will see NULL in these columns when you query the Hive table.

 **Note**

The CREATE EXTERNAL TABLE statement does not perform any validation on the TBLPROPERTIES clause. The values you provide for dynamodb.table.name and dynamodb.column.mapping are only evaluated by the DynamoDBStorageHandler class when you attempt to access the table.

Data type mappings

The following table shows DynamoDB data types and compatible Hive data types:

DynamoDB Data Type	Hive Data Type
String	STRING
Number	BIGINT or DOUBLE
Binary	BINARY
String Set	ARRAY<STRING>
Number Set	ARRAY<BIGINT> or ARRAY<DOUBLE>
Binary Set	ARRAY<BINARY>

Note

The following DynamoDB data types are not supported by the `DynamoDBStorageHandler` class, so they cannot be used with `dynamodb.column.mapping`:

- Map
- List
- Boolean
- Null

However, if you need to work with these data types, you can create a single entity called `item` that represents the entire DynamoDB item as a map of strings for both keys and values in the map. For more information, see [Copying data without a column mapping](#)

If you want to map a DynamoDB attribute of type Number, you must choose an appropriate Hive type:

- The Hive BIGINT type is for 8-byte signed integers. It is the same as the `long` data type in Java.
- The Hive DOUBLE type is for 8-bit double precision floating point numbers. It is the same as the `double` type in Java.

If you have numeric data stored in DynamoDB that has a higher precision than the Hive data type you choose, then accessing the DynamoDB data could cause a loss of precision.

If you export data of type Binary from DynamoDB to (Amazon S3) or HDFS, the data is stored as a Base64-encoded string. If you import data from Amazon S3 or HDFS into the DynamoDB Binary type, you must ensure the data is encoded as a Base64 string.

Processing HiveQL statements

Hive is an application that runs on Hadoop, which is a batch-oriented framework for running MapReduce jobs. When you issue a HiveQL statement, Hive determines whether it can return the results immediately or whether it must submit a MapReduce job.

For example, consider the *ddb_features* table (from [Tutorial: Working with Amazon DynamoDB and Apache Hive](#)). The following Hive query prints state abbreviations and the number of summits in each:

```
SELECT state_alpha, count(*)  
FROM ddb_features  
WHERE feature_class = 'Summit'  
GROUP BY state_alpha;
```

Hive does not return the results immediately. Instead, it submits a MapReduce job, which is processed by the Hadoop framework. Hive will wait until the job is complete before it shows the results from the query:

```
AK 2  
AL 2  
AR 2  
AZ 3  
CA 7  
CO 2  
CT 2  
ID 1  
KS 1  
ME 2  
MI 1  
MT 3  
NC 1  
NE 1  
NM 1  
NY 2  
OR 5  
PA 1  
TN 1  
TX 1  
UT 4  
VA 1  
VT 2  
WA 2  
WY 3  
  
Time taken: 8.753 seconds, Fetched: 25 row(s)
```

Monitoring and canceling jobs

When Hive launches a Hadoop job, it prints output from that job. The job completion status is updated as the job progresses. In some cases, the status might not be updated for a long time. (This can happen when you are querying a large DynamoDB table that has a low provisioned read capacity setting.)

If you need to cancel the job before it is complete, you can type **Ctrl+C** at any time.

Querying data in DynamoDB

The following examples show some ways that you can use HiveQL to query data stored in DynamoDB.

These examples refer to the *ddb_features* table in the tutorial ([Step 5: Copy data to DynamoDB](#)).

Topics

- [Using aggregate functions](#)
- [Using the GROUP BY and HAVING clauses](#)
- [Joining two DynamoDB tables](#)
- [Joining tables from different sources](#)

Using aggregate functions

HiveQL provides built-in functions for summarizing data values. For example, you can use the MAX function to find the largest value for a selected column. The following example returns the elevation of the highest feature in the state of Colorado.

```
SELECT MAX(elev_in_ft)
FROM ddb_features
WHERE state_alpha = 'CO';
```

Using the GROUP BY and HAVING clauses

You can use the GROUP BY clause to collect data across multiple records. This is often used with an aggregate function such as SUM, COUNT, MIN, or MAX. You can also use the HAVING clause to discard any results that do not meet certain criteria.

The following example returns a list of the highest elevations from states that have more than five features in the *ddb_features* table.

```
SELECT state_alpha, max(elev_in_ft)
FROM ddb_features
GROUP BY state_alpha
HAVING count(*) >= 5;
```

Joining two DynamoDB tables

The following example maps another Hive table (*east_coast_states*) to a table in DynamoDB. The SELECT statement is a join across these two tables. The join is computed on the cluster and returned. The join does not take place in DynamoDB.

Consider a DynamoDB table named *EastCoastStates* that contains the following data:

StateName	StateAbbrev
Maine	ME
New Hampshire	NH
Massachusetts	MA
Rhode Island	RI
Connecticut	CT
New York	NY
New Jersey	NJ
Delaware	DE
Maryland	MD
Virginia	VA
North Carolina	NC
South Carolina	SC
Georgia	GA
Florida	FL

Let's assume the table is available as a Hive external table named *east_coast_states*:

```
CREATE EXTERNAL TABLE ddb_east_coast_states (state_name STRING, state_alpha STRING)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "EastCoastStates",
"dynamodb.column.mapping" = "state_name:StateName,state_alpha:StateAbbrev");
```

The following join returns the states on the East Coast of the United States that have at least three features:

```
SELECT ecs.state_name, f.feature_class, COUNT(*)  
FROM ddb_east_coast_states ecs  
JOIN ddb_features f on ecs.state_alpha = f.state_alpha  
GROUP BY ecs.state_name, f.feature_class  
HAVING COUNT(*) >= 3;
```

Joining tables from different sources

In the following example, `s3_east_coast_states` is a Hive table associated with a CSV file stored in Amazon S3. The `ddb_features` table is associated with data in DynamoDB. The following example joins these two tables, returning the geographic features from states whose names begin with "New."

```
create external table s3_east_coast_states (state_name STRING, state_alpha STRING)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
LOCATION 's3://<bucketname>/path/subpath/';
```

```
SELECT ecs.state_name, f.feature_name, f.feature_class  
FROM s3_east_coast_states ecs  
JOIN ddb_features f  
ON ecs.state_alpha = f.state_alpha  
WHERE ecs.state_name LIKE 'New%';
```

Copying data to and from Amazon DynamoDB

In the [Tutorial: Working with Amazon DynamoDB and Apache Hive](#), you copied data from a native Hive table into an external DynamoDB table, and then queried the external DynamoDB table. The table is external because it exists outside of Hive. Even if you drop the Hive table that maps to it, the table in DynamoDB is not affected.

Hive is an excellent solution for copying data among DynamoDB tables, Amazon S3 buckets, native Hive tables, and Hadoop Distributed File System (HDFS). This section provides examples of these operations.

Topics

- [Copying data between DynamoDB and a native Hive table](#)
- [Copying data between DynamoDB and Amazon S3](#)
- [Copying data between DynamoDB and HDFS](#)

- [Using data compression](#)
- [Reading non-printable UTF-8 character data](#)

Copying data between DynamoDB and a native Hive table

If you have data in a DynamoDB table, you can copy the data to a native Hive table. This will give you a snapshot of the data, as of the time you copied it.

You might decide to do this if you need to perform many HiveQL queries, but do not want to consume provisioned throughput capacity from DynamoDB. Because the data in the native Hive table is a copy of the data from DynamoDB, and not "live" data, your queries should not expect that the data is up-to-date.

 **Note**

The examples in this section are written with the assumption you followed the steps in [Tutorial: Working with Amazon DynamoDB and Apache Hive](#) and have an external table in DynamoDB named *ddb_features*.

Example From DynamoDB to native Hive table

You can create a native Hive table and populate it with data from *ddb_features*, like this:

```
CREATE TABLE features_snapshot AS  
SELECT * FROM ddb_features;
```

You can then refresh the data at any time:

```
INSERT OVERWRITE TABLE features_snapshot  
SELECT * FROM ddb_features;
```

In these examples, the subquery `SELECT * FROM ddb_features` will retrieve all of the data from *ddb_features*. If you only want to copy a subset of the data, you can use a `WHERE` clause in the subquery.

The following example creates a native Hive table, containing only some of the attributes for lakes and summits:

```
CREATE TABLE lakes_and_summits AS
SELECT feature_name, feature_class, state_alpha
FROM ddb_features
WHERE feature_class IN ('Lake', 'Summit');
```

Example From native Hive table to DynamoDB

Use the following HiveQL statement to copy the data from the native Hive table to *ddb_features*:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM features_snapshot;
```

Copying data between DynamoDB and Amazon S3

If you have data in a DynamoDB table, you can use Hive to copy the data to an Amazon S3 bucket.

You might do this if you want to create an archive of data in your DynamoDB table. For example, suppose you have a test environment where you need to work with a baseline set of test data in DynamoDB. You can copy the baseline data to an Amazon S3 bucket, and then run your tests. Afterward, you can reset the test environment by restoring the baseline data from the Amazon S3 bucket to DynamoDB.

If you worked through [Tutorial: Working with Amazon DynamoDB and Apache Hive](#), then you already have an Amazon S3 bucket that contains your Amazon EMR logs. You can use this bucket for the examples in this section, if you know the root path for the bucket:

1. Open the Amazon EMR console at <https://console.aws.amazon.com/emr>.
2. For **Name**, choose your cluster.
3. The URI is listed in **Log URI** under **Configuration Details**.
4. Make a note of the root path of the bucket. The naming convention is:

`s3://aws-logs-accountID-region`

where *accountID* is your AWS account ID and *region* is the AWS region for the bucket.

Note

For these examples, we will use a subpath within the bucket, as in this example:

```
s3://aws-logs-123456789012-us-west-2/hive-test
```

The following procedures are written with the assumption you followed the steps in the tutorial and have an external table in DynamoDB named *ddb_features*.

Topics

- [Copying data using the Hive default format](#)
- [Copying data with a user-specified format](#)
- [Copying data without a column mapping](#)
- [Viewing the data in Amazon S3](#)

Copying data using the Hive default format

Example From DynamoDB to Amazon S3

Use an `INSERT OVERWRITE` statement to write directly to Amazon S3.

```
INSERT OVERWRITE DIRECTORY 's3://aws-logs-123456789012-us-west-2/hive-test'  
SELECT * FROM ddb_features;
```

The data file in Amazon S3 looks like this:

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135  
1178153^AJones Run^ASTream^APA^A41.2120086^A-79.25920781260  
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133  
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900  
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

Each field is separated by an SOH character (start of heading, 0x01). In the file, SOH appears as `^A`.

Example From Amazon S3 to DynamoDB

1. Create an external table pointing to the unformatted data in Amazon S3.

```
CREATE EXTERNAL TABLE s3_features_unformatted  
(feature_id      BIGINT,  
 feature_name    STRING ,  
 feature_class   STRING ,
```

```
    state_alpha      STRING,  
    prim_lat_dec   DOUBLE ,  
    prim_long_dec  DOUBLE ,  
    elev_in_ft     BIGINT)  
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. Copy the data to DynamoDB.

```
INSERT OVERWRITE TABLE ddb_features  
SELECT * FROM s3_features_unformatted;
```

Copying data with a user-specified format

If you want to specify your own field separator character, you can create an external table that maps to the Amazon S3 bucket. You might use this technique for creating data files with comma-separated values (CSV).

Example From DynamoDB to Amazon S3

1. Create a Hive external table that maps to Amazon S3. When you do this, ensure that the data types are consistent with those of the DynamoDB external table.

```
CREATE EXTERNAL TABLE s3_features_csv  
(feature_id      BIGINT,  
 feature_name    STRING,  
 feature_class   STRING,  
 state_alpha     STRING,  
 prim_lat_dec   DOUBLE,  
 prim_long_dec  DOUBLE,  
 elev_in_ft     BIGINT)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. Copy the data from DynamoDB.

```
INSERT OVERWRITE TABLE s3_features_csv  
SELECT * FROM ddb_features;
```

The data file in Amazon S3 looks like this:

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135  
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260  
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133  
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900  
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example From Amazon S3 to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from Amazon S3:

```
INSERT OVERWRITE TABLE ddb_features  
SELECT * FROM s3_features_csv;
```

Copying data without a column mapping

You can copy data from DynamoDB in a raw format and write it to Amazon S3 without specifying any data types or column mapping. You can use this method to create an archive of DynamoDB data and store it in Amazon S3.

Example From DynamoDB to Amazon S3

1. Create an external table associated with your DynamoDB table. (There is no dynamodb.column.mapping in this HiveQL statement.)

```
CREATE EXTERNAL TABLE ddb_features_no_mapping  
  (item MAP<STRING, STRING>)  
 STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
 TBLPROPERTIES ("dynamodb.table.name" = "Features");
```

2. Create another external table associated with your Amazon S3 bucket.

```
CREATE EXTERNAL TABLE s3_features_no_mapping  
  (item MAP<STRING, STRING>)  
 ROW FORMAT DELIMITED  
 FIELDS TERMINATED BY '\t'  
 LINES TERMINATED BY '\n'  
 LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

3. Copy the data from DynamoDB to Amazon S3.

```
INSERT OVERWRITE TABLE s3_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

The data file in Amazon S3 looks like this:

```
Name^C{"s":"Soldiers Farewell
Hill"}^BState^C{"s":"NM"}^BCClass^C{"s":"Summit"}^BElevation^C{"n":6135"}^BLatitude^C{"n":32.25
Name^C{"s":"Jones
Run"}^BState^C{"s":"PA"}^BCClass^C{"s":"Stream"}^BElevation^C{"n":1260"}^BLatitude^C{"n":41.25
Name^C{"s":"Sentinel
Dome"}^BState^C{"s":"CA"}^BCClass^C{"s":"Summit"}^BElevation^C{"n":8133"}^BLatitude^C{"n":37.25
Name^C{"s":"NeverSweet
Gulch"}^BState^C{"s":"CA"}^BCClass^C{"s":"Valley"}^BElevation^C{"n":2900"}^BLatitude^C{"n":41.25
Name^C{"s":"Chacaloochee
Bay"}^BState^C{"s":"AL"}^BCClass^C{"s":"Bay"}^BElevation^C{"n":0"}^BLatitude^C{"n":30.6979676}
```

Each field begins with an STX character (start of text, 0x02) and ends with an ETX character (end of text, 0x03). In the file, STX appears as ^B and ETX appears as ^C.

Example From Amazon S3 to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from Amazon S3:

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM s3_features_no_mapping;
```

Viewing the data in Amazon S3

If you use SSH to connect to the leader node, you can use the AWS Command Line Interface (AWS CLI) to access the data that Hive wrote to Amazon S3.

The following steps are written with the assumption you have copied data from DynamoDB to Amazon S3 using one of the procedures in this section.

1. If you are currently at the Hive command prompt, exit to the Linux command prompt.

```
hive> exit;
```

2. List the contents of the hive-test directory in your Amazon S3 bucket. (This is where Hive copied the data from DynamoDB.)

```
aws s3 ls s3://aws-logs-123456789012-us-west-2/hive-test/
```

The response should look similar to this:

```
2016-11-01 23:19:54 81983 000000_0
```

The file name (`000000_0`) is system-generated.

3. (Optional) You can copy the data file from Amazon S3 to the local file system on the leader node. After you do this, you can use standard Linux command line utilities to work with the data in the file.

```
aws s3 cp s3://aws-logs-123456789012-us-west-2/hive-test/000000_0 .
```

The response should look similar to this:

```
download: s3://aws-logs-123456789012-us-west-2/hive-test/000000_0  
to ./000000_0
```

 **Note**

The local file system on the leader node has limited capacity. Do not use this command with files that are larger than the available space in the local file system.

Copying data between DynamoDB and HDFS

If you have data in a DynamoDB table, you can use Hive to copy the data to the Hadoop Distributed File System (HDFS).

You might do this if you are running a MapReduce job that requires data from DynamoDB. If you copy the data from DynamoDB into HDFS, Hadoop can process it, using all of the available nodes in the Amazon EMR cluster in parallel. When the MapReduce job is complete, you can then write the results from HDFS to DDB.

In the following examples, Hive will read from and write to the following HDFS directory: `/user/hadoop/hive-test`

Note

The examples in this section are written with the assumption you followed the steps in [Tutorial: Working with Amazon DynamoDB and Apache Hive](#) and you have an external table in DynamoDB named *ddb_features*.

Topics

- [Copying data using the Hive default format](#)
- [Copying data with a user-specified format](#)
- [Copying data without a column mapping](#)
- [Accessing the data in HDFS](#)

Copying data using the Hive default format**Example From DynamoDB to HDFS**

Use an `INSERT OVERWRITE` statement to write directly to HDFS.

```
INSERT OVERWRITE DIRECTORY 'hdfs://user/hadoop/hive-test'  
SELECT * FROM ddb_features;
```

The data file in HDFS looks like this:

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135  
1178153^AJones Run^ASTream^APA^A41.2120086^A-79.25920781260  
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133  
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900  
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

Each field is separated by an SOH character (start of heading, 0x01). In the file, SOH appears as `^A`.

Example From HDFS to DynamoDB

1. Create an external table that maps to the unformatted data in HDFS.

```
CREATE EXTERNAL TABLE hdfs_features_unformatted
```

```
(feature_id      BIGINT,
 feature_name    STRING ,
 feature_class   STRING ,
 state_alpha     STRING,
 prim_lat_dec   DOUBLE ,
 prim_long_dec  DOUBLE ,
 elev_in_ft      BIGINT)
LOCATION 'hdfs://user/hadoop/hive-test';
```

2. Copy the data to DynamoDB.

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM hdfs_features_unformatted;
```

Copying data with a user-specified format

If you want to use a different field separator character, you can create an external table that maps to the HDFS directory. You might use this technique for creating data files with comma-separated values (CSV).

Example From DynamoDB to HDFS

1. Create a Hive external table that maps to HDFS. When you do this, ensure that the data types are consistent with those of the DynamoDB external table.

```
CREATE EXTERNAL TABLE hdfs_features_csv
(feature_id      BIGINT,
 feature_name    STRING ,
 feature_class   STRING ,
 state_alpha     STRING,
 prim_lat_dec   DOUBLE ,
 prim_long_dec  DOUBLE ,
 elev_in_ft      BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 'hdfs://user/hadoop/hive-test';
```

2. Copy the data from DynamoDB.

```
INSERT OVERWRITE TABLE hdfs_features_csv
SELECT * FROM ddb_features;
```

The data file in HDFS looks like this:

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135  
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260  
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133  
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900  
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example From HDFS to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from HDFS:

```
INSERT OVERWRITE TABLE ddb_features  
SELECT * FROM hdfs_features_csv;
```

Copying data without a column mapping

You can copy data from DynamoDB in a raw format and write it to HDFS without specifying any data types or column mapping. You can use this method to create an archive of DynamoDB data and store it in HDFS.

Note

If your DynamoDB table contains attributes of type Map, List, Boolean or Null, then this is the only way you can use Hive to copy data from DynamoDB to HDFS.

Example From DynamoDB to HDFS

1. Create an external table associated with your DynamoDB table. (There is no dynamodb.column.mapping in this HiveQL statement.)

```
CREATE EXTERNAL TABLE ddb_features_no_mapping  
  (item MAP<STRING, STRING>)  
 STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
 TBLPROPERTIES ("dynamodb.table.name" = "Features");
```

2. Create another external table associated with your HDFS directory.

```
CREATE EXTERNAL TABLE hdfs_features_no_mapping
  (item MAP<STRING, STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
LOCATION 'hdfs://user/hadoop/hive-test';
```

3. Copy the data from DynamoDB to HDFS.

```
INSERT OVERWRITE TABLE hdfs_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

The data file in HDFS looks like this:

```
Name^C{"s":"Soldiers Farewell
Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":6135"}^BLatitude^C{"n":32.25}^BLongitude^C{"n":-106.35}
Name^C{"s":"Jones
Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":1260"}^BLatitude^C{"n":41.25}^BLongitude^C{"n":-75.75}
Name^C{"s":"Sentinel
Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":8133"}^BLatitude^C{"n":37.83}^BLongitude^C{"n":-122.45}
Name^C{"s":"Neversweet
Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":2900"}^BLatitude^C{"n":41.25}^BLongitude^C{"n":-121.75}
Name^C{"s":"Chacaloochee
Bay"}^BState^C{"s":"AL"}^BClass^C{"s":"Bay"}^BElevation^C{"n":0"}^BLatitude^C{"n":30.69796767}^BLongitude^C{"n":-86.75}
```

Each field begins with an STX character (start of text, 0x02) and ends with an ETX character (end of text, 0x03). In the file, STX appears as ^B and ETX appears as ^C.

Example From HDFS to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from HDFS:

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM hdfs_features_no_mapping;
```

Accessing the data in HDFS

HDFS is a distributed file system, accessible to all of the nodes in the Amazon EMR cluster. If you use SSH to connect to the leader node, you can use command line tools to access the data that Hive wrote to HDFS.

HDFS is not the same thing as the local file system on the leader node. You cannot work with files and directories in HDFS using standard Linux commands (such as `cat`, `cp`, `mv`, or `rm`). Instead, you perform these tasks using the `hadoop fs` command.

The following steps are written with the assumption you have copied data from DynamoDB to HDFS using one of the procedures in this section.

1. If you are currently at the Hive command prompt, exit to the Linux command prompt.

```
hive> exit;
```

2. List the contents of the `/user/hadoop/hive-test` directory in HDFS. (This is where Hive copied the data from DynamoDB.)

```
hadoop fs -ls /user/hadoop/hive-test
```

The response should look similar to this:

```
Found 1 items  
-rw-r--r-- 1 hadoop hadoop 29504 2016-06-08 23:40 /user/hadoop/hive-test/000000_0
```

The file name (`000000_0`) is system-generated.

3. View the contents of the file:

```
hadoop fs -cat /user/hadoop/hive-test/000000_0
```

Note

In this example, the file is relatively small (approximately 29 KB). Be careful when you use this command with files that are very large or contain non-printable characters.

4. (Optional) You can copy the data file from HDFS to the local file system on the leader node. After you do this, you can use standard Linux command line utilities to work with the data in the file.

```
hadoop fs -get /user/hadoop/hive-test/000000_0
```

This command will not overwrite the file.

Note

The local file system on the leader node has limited capacity. Do not use this command with files that are larger than the available space in the local file system.

Using data compression

When you use Hive to copy data among different data sources, you can request on-the-fly data compression. Hive provides several compression codecs. You can choose one during your Hive session. When you do this, the data is compressed in the specified format.

The following example compresses data using the Lempel-Ziv-Oberhumer (LZO) algorithm.

```
SET hive.exec.compress.output=true;
SET io.seqfile.compression.type=BLOCK;
SET mapred.output.compression.codec = com.hadoop.compression.lzo.LzopCodec;

CREATE EXTERNAL TABLE lzo_compression_table (line STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE lzo_compression_table SELECT *
FROM hiveTableName;
```

The resulting file in Amazon S3 will have a system-generated name with .lzo at the end (for example, 8d436957-57ba-4af7-840c-96c2fc7bb6f5-000000.lzo).

The available compression codecs are:

- org.apache.hadoop.io.compress.GzipCodec
- org.apache.hadoop.io.compress.DefaultCodec
- com.hadoop.compression.lzo.LzoCodec
- com.hadoop.compression.lzo.LzopCodec
- org.apache.hadoop.io.compress.BZip2Codec
- org.apache.hadoop.io.compress.SnappyCodec

Reading non-printable UTF-8 character data

To read and write non-printable UTF-8 character data, you can use the STORED AS SEQUENCEFILE clause when you create a Hive table. A SequenceFile is a Hadoop binary file format. You need to use Hadoop to read this file. The following example shows how to export data from DynamoDB into Amazon S3. You can use this functionality to handle non-printable UTF-8 encoded characters.

```
CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col array<string>)
STORED AS SEQUENCEFILE
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

Performance tuning

When you create a Hive external table that maps to a DynamoDB table, you do not consume any read or write capacity from DynamoDB. However, read and write activity on the Hive table (such as INSERT or SELECT) translates directly into read and write operations on the underlying DynamoDB table.

Apache Hive on Amazon EMR implements its own logic for balancing the I/O load on the DynamoDB table and seeks to minimize the possibility of exceeding the table's provisioned throughput. At the end of each Hive query, Amazon EMR returns runtime metrics, including the number of times your provisioned throughput was exceeded. You can use this information, together with CloudWatch metrics on your DynamoDB table, to improve performance in subsequent requests.

The Amazon EMR console provides basic monitoring tools for your cluster. For more information, see [View and Monitor a Cluster](#) in the *Amazon EMR Management Guide*.

You can also monitor your cluster and Hadoop jobs using web-based tools, such as Hue, Ganglia, and the Hadoop web interface. For more information, see [View Web Interfaces Hosted on Amazon EMR Clusters](#) in the *Amazon EMR Management Guide*.

This section describes steps you can take to performance-tune Hive operations on external DynamoDB tables.

Topics

- [DynamoDB provisioned throughput](#)
- [Adjusting the mappers](#)
- [Additional topics](#)

DynamoDB provisioned throughput

When you issue HiveQL statements against the external DynamoDB table, the `DynamoDBStorageHandler` class makes the appropriate low-level DynamoDB API requests, which consume provisioned throughput. If there is not enough read or write capacity on the DynamoDB table, the request will be throttled, resulting in slow HiveQL performance. For this reason, you should ensure that the table has enough throughput capacity.

For example, suppose that you have provisioned 100 read capacity units for your DynamoDB table. This will let you read 409,600 bytes per second (100×4 KB read capacity unit size). Now suppose that the table contains 20 GB of data (21,474,836,480 bytes) and you want to use the SELECT statement to select all of the data using HiveQL. You can estimate how long the query will take to run like this:

$$21,474,836,480 / 409,600 = 52,429 \text{ seconds} = 14.56 \text{ hours}$$

In this scenario, the DynamoDB table is a bottleneck. It won't help to add more Amazon EMR nodes, because the Hive throughput is constrained to only 409,600 bytes per second. The only way to decrease the time required for the SELECT statement is to increase the provisioned read capacity of the DynamoDB table.

You can perform a similar calculation to estimate how long it would take to bulk-load data into a Hive external table mapped to a DynamoDB table. Determine the total number of write capacity units needed per item (less than 1KB = 1, 1-2KB = 2, etc), and multiply that by the number of items to load. This will give you the number of write capacity units required. Divide that number by the number of write capacity units that are allocated per second. This will yield the number of seconds it will take to load the table.

You should regularly monitor the CloudWatch metrics for your table. For a quick overview in the DynamoDB console, choose your table and then choose the **Metrics** tab. From here, you can view read and write capacity units consumed and read and write requests that have been throttled.

Read capacity

Amazon EMR manages the request load against your DynamoDB table, according to the table's provisioned throughput settings. However, if you notice a large number of ProvisionedThroughputExceeded messages in the job output, you can adjust the default read rate. To do this, you can modify the dynamodb.throughput.read.percent configuration variable. You can use the SET command to set this variable at the Hive command prompt:

```
SET dynamodb.throughput.read.percent=1.0;
```

This variable persists for the current Hive session only. If you exit Hive and return to it later, dynamodb.throughput.read.percent will return to its default value.

The value of dynamodb.throughput.read.percent can be between 0.1 and 1.5, inclusively. 0.5 represents the default read rate, meaning that Hive will attempt to consume half of the read capacity of the table. If you increase the value above 0.5, Hive will increase the request rate; decreasing the value below 0.5 decreases the read request rate. (The actual read rate will vary, depending on factors such as whether there is a uniform key distribution in the DynamoDB table.)

If you notice that Hive is frequently depleting the provisioned read capacity of the table, or if your read requests are being throttled too much, try reducing dynamodb.throughput.read.percent below 0.5. If you have sufficient read capacity in the table and want more responsive HiveQL operations, you can set the value above 0.5.

Write capacity

Amazon EMR manages the request load against your DynamoDB table, according to the table's provisioned throughput settings. However, if you notice a large number of ProvisionedThroughputExceeded messages in the job output, you can adjust the default write rate. To do this, you can modify the dynamodb.throughput.write.percent configuration variable. You can use the SET command to set this variable at the Hive command prompt:

```
SET dynamodb.throughput.write.percent=1.0;
```

This variable persists for the current Hive session only. If you exit Hive and return to it later, dynamodb.throughput.write.percent will return to its default value.

The value of dynamodb.throughput.write.percent can be between 0.1 and 1.5, inclusively. 0.5 represents the default write rate, meaning that Hive will attempt to consume half of the write capacity of the table. If you increase the value above 0.5, Hive will increase the request rate;

decreasing the value below `0.5` decreases the write request rate. (The actual write rate will vary, depending on factors such as whether there is a uniform key distribution in the DynamoDB table.)

If you notice that Hive is frequently depleting the provisioned write capacity of the table, or if your write requests are being throttled too much, try reducing `dynamodb.throughput.write.percent` below `0.5`. If you have sufficient capacity in the table and want more responsive HiveQL operations, you can set the value above `0.5`.

When you write data to DynamoDB using Hive, ensure that the number of write capacity units is greater than the number of mappers in the cluster. For example, consider an Amazon EMR cluster consisting of 10 `m1.xlarge` nodes. The `m1.xlarge` node type provides 8 mapper tasks, so the cluster would have a total of 80 mappers (10×8). If your DynamoDB table has fewer than 80 write capacity units, then a Hive write operation could consume all of the write throughput for that table.

To determine the number of mappers for Amazon EMR node types, see [Task Configuration](#) in the *Amazon EMR Developer Guide*.

For more information on mappers, see [Adjusting the mappers](#).

Adjusting the mappers

When Hive launches a Hadoop job, the job is processed by one or more mapper tasks. Assuming that your DynamoDB table has sufficient throughput capacity, you can modify the number of mappers in the cluster, potentially improving performance.

Note

The number of mapper tasks used in a Hadoop job are influenced by *input splits*, where Hadoop subdivides the data into logical blocks. If Hadoop does not perform enough input splits, then your write operations might not be able to consume all the write throughput available in the DynamoDB table.

Increasing the number of mappers

Each mapper in an Amazon EMR has a maximum read rate of 1 MiB per second. The number of mappers in a cluster depends on the size of the nodes in your cluster. (For information about node sizes and the number of mappers per node, see [Task Configuration](#) in the *Amazon EMR Developer Guide*.)

If your DynamoDB table has ample throughput capacity for reads, you can try increasing the number of mappers by doing one of the following:

- Increase the size of the nodes in your cluster. For example, if your cluster is using *m1.large* nodes (three mappers per node), you can try upgrading to *m1.xlarge* nodes (eight mappers per node).
- Increase the number of nodes in your cluster. For example, if you have three-node cluster of *m1.xlarge* nodes, you have a total of 24 mappers available. If you were to double the size of the cluster, with the same type of node, you would have 48 mappers.

You can use the AWS Management Console to manage the size or the number of nodes in your cluster. (You might need to restart the cluster for these changes to take effect.)

Another way to increase the number of mappers is to modify the `mapred.tasktracker.map.tasks.maximum` Hadoop configuration parameter. (This is a Hadoop parameter, not a Hive parameter. You cannot modify it interactively from the command prompt.). If you increase the value of `mapred.tasktracker.map.tasks.maximum`, you can increase the number of mappers without increasing the size or number of nodes. However, it is possible for the cluster nodes to run out of memory if you set the value too high.

You set the value for `mapred.tasktracker.map.tasks.maximum` as a bootstrap action when you first launch your Amazon EMR cluster. For more information, see [\(Optional\) Create Bootstrap Actions to Install Additional Software](#) in the *Amazon EMR Management Guide*.

Decreasing the number of mappers

If you use the `SELECT` statement to select data from an external Hive table that maps to DynamoDB, the Hadoop job can use as many tasks as necessary, up to the maximum number of mappers in the cluster. In this scenario, it is possible that a long-running Hive query can consume all of the provisioned read capacity of the DynamoDB table, negatively impacting other users.

You can use the `dynamodb.max.map.tasks` parameter to set an upper limit for map tasks:

```
SET dynamodb.max.map.tasks=1
```

This value must be equal to or greater than 1. When Hive processes your query, the resulting Hadoop job will use no more than `dynamodb.max.map.tasks` when reading from the DynamoDB table.

Additional topics

The following are some more ways to tune applications that use Hive to access DynamoDB.

Retry duration

By default, Hive will rerun a Hadoop job if it has not returned any results from DynamoDB within two minutes. You can adjust this interval by modifying the `dynamodb.retry.duration` parameter:

```
SET dynamodb.retry.duration=2;
```

The value must be a nonzero integer, representing the number of minutes in the retry interval. The default for `dynamodb.retry.duration` is 2 (minutes).

Parallel data requests

Multiple data requests, either from more than one user or more than one application to a single table can drain read provisioned throughput and slow performance.

Process duration

Data consistency in DynamoDB depends on the order of read and write operations on each node. While a Hive query is in progress, another application might load new data into the DynamoDB table or modify or delete existing data. In this case, the results of the Hive query might not reflect changes made to the data while the query was running.

Request time

Scheduling Hive queries that access a DynamoDB table when there is lower demand on the DynamoDB table improves performance. For example, if most of your application's users live in San Francisco, you might choose to export daily data at 4:00 A.M. PST when the majority of users are asleep and not updating records in your DynamoDB database.

Integrating with Amazon S3

Amazon DynamoDB import and export capabilities provide a simple and efficient way to move data between Amazon S3 and DynamoDB tables without writing any code.

DynamoDB import and export features help you move, transform, and copy DynamoDB table accounts. You can import from your S3 sources, and you can export your DynamoDB table data to

Amazon S3 and use AWS services such as Athena, Amazon SageMaker, and AWS Lake Formation to analyze your data and extract actionable insights. You can also import data directly into new DynamoDB tables to build new applications with single-digit millisecond performance at scale, facilitate data sharing between tables and accounts, and simplify your disaster recovery and business continuity plans.

Topics

- [DynamoDB data import from Amazon S3: how it works](#)
- [DynamoDB data export to Amazon S3: how it works](#)

DynamoDB data import from Amazon S3: how it works

To import data into DynamoDB, your data must be in an Amazon S3 bucket in CSV, DynamoDB JSON, or Amazon Ion format. Data can be compressed in ZSTD or GZIP format, or can be directly imported in uncompressed form. Source data can either be a single Amazon S3 object or multiple Amazon S3 objects that use the same prefix.

Your data will be imported into a new DynamoDB table, which will be created when you initiate the import request. You can create this table with secondary indexes, then query and update your data across all primary and secondary indexes as soon as the import is complete. You can also add a global table replica after the import is complete.

Note

During the Amazon S3 import process, DynamoDB creates a new target table that will be imported into. Import into existing tables is not currently supported by this feature.

Import from Amazon S3 does not consume write capacity on the new table, so you do not need to provision any extra capacity for importing data into DynamoDB. Data import pricing is based on the uncompressed size of the source data in Amazon S3, that is processed as a result of the import. Items that are processed but fail to load into the table due to formatting or other inconsistencies in the source data are also billed as part of the import process. See [Amazon DynamoDB pricing](#) for details.

You can import data from an Amazon S3 bucket owned by a different account if you have the correct permissions to read from that specific bucket. The new table may also be in a different

Region from the source Amazon S3 bucket. For more information, see [Amazon Simple Storage Service setup and permissions](#).

Import times are directly related to your data's characteristics in Amazon S3. This includes data size, data format, compression scheme, uniformity of data distribution, number of Amazon S3 objects, and other related variables. In particular, data sets with uniformly distributed keys will be faster to import than skewed data sets. For example, if your secondary index's key is using the month of the year for partitioning, and all your data is from the month of December, then importing this data may take significantly longer.

The attributes associated with keys are expected to be unique on the base table. If any keys are not unique, the import will overwrite the associated items until only the last overwrite remains. For example, if the primary key is the month and multiple items are set to the month of September, each new item will overwrite the previously written items and only one item with the primary key of "month" set to September will remain. In such cases, the number of items processed in the import table description will not match the number of items in the target table.

AWS CloudTrail logs all console and API actions for table import. For more information, see [Logging DynamoDB operations by using AWS CloudTrail](#).

The following video is an introduction to importing directly from Amazon S3 into DynamoDB.

[Import from Amazon S3](#)

Topics

- [Requesting a table import in DynamoDB](#)
- [Amazon S3 import formats for DynamoDB](#)
- [Import format quotas and validation](#)
- [Best practices for importing from Amazon S3 into DynamoDB](#)

Requesting a table import in DynamoDB

DynamoDB import allows you to import data from an Amazon S3 bucket to a new DynamoDB table. You can request a table import using the [DynamoDB console](#), the [CLI](#), [CloudFormation](#) or the [DynamoDB API](#).

If you want to use the AWS CLI, you must configure it first. For more information, see [Accessing DynamoDB](#).

 **Note**

- The Import Table feature interacts with multiple different AWS Services such as Amazon S3 and CloudWatch. Before you begin an import, make sure that the user or role that invokes the import APIs has permissions to all services and resources the feature depends on.
 - Do not modify the Amazon S3 objects while the import is in progress, as this can cause the operation to fail or be cancelled.

For more information on errors and troubleshooting, see [Import format quotas and validation](#)

Topics

- [Setting up IAM permissions](#)
 - [Requesting an import using the AWS Management Console](#)
 - [Getting details about past imports in the AWS Management Console](#)
 - [Requesting an import using the AWS CLI](#)
 - [Getting details about past imports in the AWS CLI](#)

Setting up IAM permissions

You can import data from any Amazon S3 bucket you have permission to read from. The source bucket does not need to be in the same Region or have the same owner as the source table. Your AWS Identity and Access Management (IAM) must include the relevant actions on the source Amazon S3 bucket, and required CloudWatch permissions for providing debugging information. An example policy is shown below.

```
        "dynamodb:DescribeImport",
        "dynamodb>ListImports"
    ],
    "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table*"
},
{
    "Sid": "AllowS3Access",
    "Effect": "Allow",
    "Action": [
        "s3:GetObject",
        "s3>ListBucket"
    ],
    "Resource": [
        "arn:aws:s3:::your-bucket/*",
        "arn:aws:s3:::your-bucket"
    ]
},
{
    "Sid": "AllowCloudwatchAccess",
    "Effect": "Allow",
    "Action": [
        "logs>CreateLogGroup",
        "logs>CreateLogStream",
        "logs>DescribeLogGroups",
        "logs>DescribeLogStreams",
        "logs>PutLogEvents",
        "logs>PutRetentionPolicy"
    ],
    "Resource": "arn:aws:logs:us-east-1:111122223333:log-group:/aws-dynamodb/*"
}
]
}
```

Amazon S3 permissions

When starting an import on an Amazon S3 bucket source that is owned by another account, ensure that the role or user has access to the Amazon S3 objects. You can check that by executing an Amazon S3 GetObject command and using the credentials. When using the API, the Amazon S3 bucket owner parameter defaults to the current user's account ID. For cross account imports, ensure that this parameter is correctly populated with the bucket owner's account ID. The following code is an example Amazon S3 bucket policy in the source account.

{

```
"Version": "2012-10-17",
"Statement": [
    {"Sid": "ExampleStatement",
        "Effect": "Allow",
        "Principal": {"AWS": "arn:aws:iam::123456789012:user/Dave"},
        "Action": [
            "s3:GetObject",
            "s3>ListBucket"
        ],
        "Resource": "arn:aws:s3:::awsexamplebucket1/*"
    }
]
```

AWS Key Management Service

When creating the new table for import, if you select an encryption at rest key that is not owned by DynamoDB then you must provide the AWS KMS permissions required to operate a DynamoDB table encrypted with customer managed keys. For more information see [Authorizing use of your AWS KMS key](#). If the Amazon S3 objects are encrypted with server side encryption KMS (SSE-KMS), ensure that the role or user initiating the import has access to decrypt using the AWS KMS key. This feature does not support customer-provided encryption keys (SSE-C) encrypted Amazon S3 objects.

CloudWatch permissions

The role or user that is initiating the import will need create and manage permissions for the log group and log streams associated with the import.

Requesting an import using the AWS Management Console

The following example demonstrates how to use the DynamoDB console to import existing data to a new table named MusicCollection.

To request a table import

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Import from S3**.
3. On the page that appears, select **Import from S3**.

4. Choose **Import from S3**.
5. Enter the Amazon S3 source URL.

If the source bucket is owned by your account, you can find it by using the **Browse S3** button. Alternately, you can enter the URL of the bucket in the `s3://bucket/prefix` format. The prefix is an Amazon S3 key prefix that is either the Amazon S3 object name that you want to import, or the key prefix shared by all the Amazon S3 objects that you want to import.

6. Specify if you are the **S3 bucket owner**. If the source bucket is owned by a different account, select **A different AWS account**. Then enter the account ID of the bucket owner.
7. Under **Import file compression**, select either **No compression**, **GZIP** or **ZSTD** as appropriate.
8. Select the appropriate Import file format. The options are **DynamoDB JSON**, **Amazon Ion** or **CSV**. If you select **CSV**, you will have two additional options: **CSV header** and **CSV delimiter character**.

For **CSV header**, choose if the header will either be taken from the first line of the file or be customized. If you select **Customize your headers**, you can specify the header values you want to import with. CSV Headers specified by this method are case-sensitive and are expected to contain the keys of the target table.

For **CSV delimiter character**, you set the character which will separate items. Comma is selected by default. If you select **Custom delimiter character**, the delimiter must match the regex pattern: `[, ; : | \t]`.

9. Select the **Next** button and select the options for the new table that will be created to store your data.

 **Note**

Primary Key and Sort Key must match the attributes in the file, or the import will fail. The attributes are case sensitive.

10. Select **Next** again to review your import options, then click **Import** to begin the import task. You will first see your new table listed in the "Tables" with the status "Creating". At this time the table is not accessible.
11. Once the import completes, the status will show as "Active" and you can start using the table.

Getting details about past imports in the AWS Management Console

You can find information about import tasks you've run in the past by clicking **Import from S3** in the navigation sidebar, then selecting the **Imports** tab. The import panel contains a list of all imports you've created in the past 90 days. Selecting the ARN of a task listed in the Imports tab will retrieve information about that import, including any advanced configuration settings you chose.

Requesting an import using the AWS CLI

The following example imports CSV formatted data from an S3 bucket called `bucket` with a prefix of `prefix` to a new table called `target-table`.

```
aws dynamodb import-table --s3-bucket-source S3Bucket=bucket,S3KeyPrefix=prefix \
    --input-format CSV --table-creation-parameters '{"TableName":"target-
table","KeySchema": \
    [{"AttributeName":"hk","KeyType":"HASH"}],"AttributeDefinitions": \
    [{"AttributeName":"hk","AttributeType":"S"}],"BillingMode":"PAY_PER_REQUEST"}' \
    --input-format-options '{"Csv": {"HeaderList": ["hk", "title", "artist", \
    "year_of_release"], "Delimiter": ";"}}'
```

Note

If you choose to encrypt your import using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination Amazon S3 bucket.

Getting details about past imports in the AWS CLI

You can find information about import tasks you've run in the past by using the `list-imports` command. This command returns a list of all imports you've created in the past 90 days. Note that although import task metadata expires after 90 days and jobs older than that are no longer found on this list, DynamoDB does not delete any of the objects in your Amazon S3 bucket or the table created during import.

```
aws dynamodb list-imports
```

To retrieve detailed information about a specific import task, including any advanced configuration settings, use the `describe-import` command.

```
aws dynamodb describe-import \
--import-arn arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/exp
```

Amazon S3 import formats for DynamoDB

DynamoDB can import data in three formats: CSV, DynamoDB JSON, and Amazon Ion.

Topics

- [CSV](#)
- [DynamoDB Json](#)
- [Amazon Ion](#)

CSV

A file in CSV format consists of multiple items delimited by newlines. By default, DynamoDB interprets the first line of an import file as the header and expects columns to be delimited by commas. You can also define headers that will be applied, as long as they match the number of columns in the file. If you define headers explicitly, the first line of the file will be imported as values.

Note

When importing from CSV files, all columns other than the hash range and keys of your base table and secondary indexes are imported as DynamoDB strings.

Escaping double quotes

Any double quotes characters that exist in the CSV file must be escaped. If they are not escaped, such as in this following example, the import will fail:

```
id,value
"123",Women's Full Lenth Dress
```

This same import will succeed if the quotes are escaped with two sets of double quotes:

```
id,value
""""123""""",Women's Full Lenth Dress
```

Once the text has been properly escaped and imported, it will appear as it did in the original CSV file:

```
id,value
"123",Women's Full Lenth Dress
```

DynamoDB Json

A file in DynamoDB JSON format can consist of multiple Item objects. Each individual object is in DynamoDB's standard marshalled JSON format, and newlines are used as item delimiters. As an added feature, exports from point in time are supported as an import source by default.

Note

New lines are used as item delimiters for a file in DynamoDB JSON format and shouldn't be used within an item object.

```
{
  "Item": {
    "Authors": {
      "SS": ["Author1", "Author2"]
    },
    "Dimensions": {
      "S": "8.5 x 11.0 x 1.5"
    },
    "ISBN": {
      "S": "333-3333333333"
    },
    "Id": {
      "N": "103"
    },
    "InPublication": {
      "BOOL": false
    },
    "PageCount": {
      "N": "600"
    },
    "Price": {
      "N": "2000"
    },
    "ProductCategory": {
```

```
        "S": "Book"
    },
    "Title": {
        "S": "Book 103 Title"
    }
}
```

Note

New lines are used as item delimiters for a file in DynamoDB JSON format and shouldn't be used within an item object.

```
{
    "Item": {
        "Authors": {
            "SS": ["Author1", "Author2"]
        },
        "Dimensions": {
            "S": "8.5 x 11.0 x 1.5"
        },
        "ISBN": {
            "S": "333-333333333"
        },
        "Id": {
            "N": "103"
        },
        "InPublication": {
            "BOOL": false
        },
        "PageCount": {
            "N": "600"
        },
        "Price": {
            "N": "2000"
        },
        "ProductCategory": {
            "S": "Book"
        },
        "Title": {
            "S": "Book 103 Title"
        }
    }
}
```

```
        }
    }
} {
    "Item": {
        "Authors": {
            "SS": ["Author1", "Author2"]
        },
        "Dimensions": {
            "S": "8.5 x 11.0 x 1.5"
        },
        "ISBN": {
            "S": "444-444444444"
        },
        "Id": {
            "N": "104"
        },
        "InPublication": {
            "BOOL": false
        },
        "PageCount": {
            "N": "600"
        },
        "Price": {
            "N": "2000"
        },
        "ProductCategory": {
            "S": "Book"
        },
        "Title": {
            "S": "Book 104 Title"
        }
    }
} {
    "Item": {
        "Authors": {
            "SS": ["Author1", "Author2"]
        },
        "Dimensions": {
            "S": "8.5 x 11.0 x 1.5"
        },
        "ISBN": {
            "S": "555-5555555555"
        },
        "Id": {
```

```
        "N": "105"
    },
    "InPublication": {
        "BOOL": false
    },
    "PageCount": {
        "N": "600"
    },
    "Price": {
        "N": "2000"
    },
    "ProductCategory": {
        "S": "Book"
    },
    "Title": {
        "S": "Book 105 Title"
    }
}
}
```

Amazon Ion

[Amazon Ion](#) is a richly-typed, self-describing, hierarchical data serialization format built to address rapid development, decoupling, and efficiency challenges faced every day while engineering large-scale, service-oriented architectures.

When you import data in Ion format, the Ion datatypes are mapped to DynamoDB datatypes in the new DynamoDB table.

	Ion to DynamoDB datatype conversion	B
1	Ion Data Type	DynamoDB Representation
2	string	String (s)
3	bool	Boolean (BOOL)
4	decimal	Number (N)
5	blob	Binary (B)

	Ion to DynamoDB datatype conversion	B
6	list (with type annotation \$dynamodb_SS, \$dynamodb_NS, or \$dynamodb_BS)	Set (SS, NS, BS)
7	list	List
8	struct	Map

Items in an Ion file are delimited by newlines. Each line begins with an Ion version marker, followed by an item in Ion format.

 **Note**

In the following example, an item from an Ion-formatted file has been formatted on multiple lines for the sake of readability.

```
$ion_1_0 {
    Item: {
        Authors:$dynamodb_SS::["Author1", "Author2"],
        Dimensions:"8.5 x 11.0 x 1.5",
        ISBN:"333-333333333",
        Id:103.,
        InPublication:false,
        PageCount:6d2,
        Price:2d3,
        ProductCategory:"Book",
        Title:"Book 103 Title"
    }
}
```

Import format quotas and validation

Import quotas

DynamoDB Import from Amazon S3 can support up to 50 concurrent import jobs with a total import source object size of 15TB at a time in us-east-1, us-west-2, and eu-west-1 regions. In all other regions, up to 50 concurrent import tasks with a total size of 1TB is supported. Each import job can take up to 50,000 Amazon S3 objects in all regions. These default quotas are applied to every account. If you feel you need to revise these quotas, please contact your account team, and this will be considered on a case-by-case basis. For more details on DynamoDB limits, see [Service Quotas](#).

Validation errors

During the import process, DynamoDB may encounter errors while parsing your data. For each error, DynamoDB emits a CloudWatch log and keeps a count of the total number of errors encountered. If the Amazon S3 object itself is malformed or if its contents cannot form a DynamoDB item, then we may skip processing the remaining portion of the object.

Note

If the Amazon S3 data source has multiple items that share the same key, the items will overwrite until one remains. This can appear as if 1 item was imported and the others were ignored. The duplicate items will be overwritten in random order, are not counted as errors, and are not emitted to CloudWatch logs.

Once the import is complete you can see the total count of items imported, total count of errors, and total count of items processed. For further troubleshooting you can also check the total size of items imported and total size of data processed.

There are three categories of import errors: API validation errors, data validation errors, and configuration errors.

API validation errors

API validation errors are item-level errors from the sync API. Common causes are permissions issues, missing required parameters and parameter validation failures. Details on why the API call failed are contained in the exceptions thrown by the `ImportTable` request.

Data validation errors

Data validation errors can occur at either the item level or file level. During import, items are validated based on DynamoDB rules before importing into the target table. When an item fails validation and is not imported, the import job skips over that item and continues on with the next item. At the end of job, the import status is set to FAILED with a FailureCode, ItemValidationError and the FailureMessage "Some of the items failed validation checks and were not imported. Please check CloudWatch error logs for more details."

Common causes for data validation errors include objects being unparsable, objects being in the incorrect format (input specifies DYNAMODB_JSON but the object is not in DYNAMODB_JSON), and schema mismatch with specified source table keys.

Configuration errors

Configuration errors are typically workflow errors due to permission validation. The Import workflow checks some permissions after accepting the request. If there are issues calling any of the required dependencies like Amazon S3 or CloudWatch the process marks the import status as FAILED. The failureCode and failureMessage point to the reason for failure. Where applicable, the failure message also contains the request id that you can use to investigate the reason for failure in CloudTrail.

Common configuration errors include having the wrong URL for the Amazon S3 bucket, and not having permission to access the Amazon S3 bucket, CloudWatch Logs, and AWS KMS keys used to decrypt the Amazon S3 object. For more information see [Using and data keys](#).

Validating source Amazon S3 objects

In order to validate source S3 objects, take the following steps.

1. Validate the data format and compression type

- Make sure that all matching Amazon S3 objects under the specified prefix have the same format (DYNAMODB_JSON, DYNAMODB_ION, CSV)
- Make sure that all matching Amazon S3 objects under the specified prefix are compressed the same way (GZIP, ZSTD, NONE)

Note

The Amazon S3 objects do not need to have the corresponding extension (.csv / .json / .ion / .gz / .zstd etc) as the input format specified in ImportTable call takes precedence.

2. Validate that the import data conforms to the desired table schema

- Make sure that each item in the source data has the primary key. A sort key is optional for imports.
- Make sure that the attribute type associated with the primary key and any sort key matches the attribute type in the Table and the GSI schema, as specified in table creation parameters

Troubleshooting

CloudWatch logs

For Import jobs that fail, detailed error messages are posted to CloudWatch logs. To access these logs, first retrieve the ImportArn from the output and describe-import using this command:

```
aws dynamodb describe-import --import-arn arn:aws:dynamodb:us-east-1:ACCOUNT:table/  
target-table/import/01658528578619-c4d4e311  
}
```

Example output:

```
aws dynamodb describe-import --import-arn "arn:aws:dynamodb:us-  
east-1:531234567890:table/target-table/import/01658528578619-c4d4e311"  
{  
    "ImportTableDescription": {  
        "ImportArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table/  
import/01658528578619-c4d4e311",  
        "ImportStatus": "FAILED",  
        "TableArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table",  
        "TableId": "7b7ecc22-302f-4039-8ea9-8e7c3eb2bcb8",  
        "ClientToken": "30f8891c-e478-47f4-af4a-67a5c3b595e3",  
        "S3BucketSource": {  
            "S3BucketOwner": "ACCOUNT",  
            "S3Bucket": "my-import-source",
```

```
        "S3KeyPrefix": "import-test"
    },
    "ErrorCount": 1,
    "CloudWatchLogGroupArn": "arn:aws:logs:us-east-1:ACCOUNT:log-group:/aws-dynamodb/imports:*",
    "InputFormat": "CSV",
    "InputCompressionType": "NONE",
    "TableCreationParameters": {
        "TableName": "target-table",
        "AttributeDefinitions": [
            {
                "AttributeName": "pk",
                "AttributeType": "S"
            }
        ],
        "KeySchema": [
            {
                "AttributeName": "pk",
                "KeyType": "HASH"
            }
        ],
        "BillingMode": "PAY_PER_REQUEST"
    },
    "StartTime": 1658528578.619,
    "EndTime": 1658528750.628,
    "ProcessedSizeBytes": 70,
    "ProcessedItemCount": 1,
    "ImportedItemCount": 0,
    "FailureCode": "ItemValidationException",
    "FailureMessage": "Some of the items failed validation checks and were not imported. Please check CloudWatch error logs for more details."
}
}
```

Retrieve the log group and the import id from the above response and use it to retrieve the error logs. The import ID is the last path element of the ImportArn field. The log group name is /aws-dynamodb/imports. The error log stream name is import-id/error. For this example, it would be 01658528578619-c4d4e311/error.

Missing the key pk in the item

If the source S3 object does not contain the primary key that was provided as a parameter, the import will fail. For example, when you define the primary key for the import as column name "pk".

```
aws dynamodb import-table --s3-bucket-source S3Bucket=my-import-source,S3KeyPrefix=import-test.csv \
    --input-format CSV --table-creation-parameters '{"TableName":"target-table","KeySchema": \
        [{"AttributeName":"pk","KeyType":"HASH"}],"AttributeDefinitions": \
        [{"AttributeName":"pk","AttributeType":"S"}],"BillingMode":"PAY_PER_REQUEST"}'
```

The column "pk" is missing from the the source object `import-test.csv` which has the following contents:

```
title,artist,year_of_release
The Dark Side of the Moon,Pink Floyd,1973
```

This import will fail due to item validation error because of the missing primary key in the data source.

Example CloudWatch error log:

```
aws logs get-log-events --log-group-name /aws-dynamodb/imports --log-stream-name
01658528578619-c4d4e311/error
{
  "events": [
    {
      "timestamp": 1658528745319,
      "message": "{\"itemS3Pointer\":{\"bucket\":\"my-import-source\",\"key\":\\\"import-test.csv\\\",\\\"itemIndex\\\":0},\\\"importArn\\\":\\\"arn:aws:dynamodb:us-east-1:531234567890:table/target-table/import/01658528578619-c4d4e311\\\",\\\"errorMessages\\\":\\\"One or more parameter values were invalid: Missing the key pk in the item\\\"]}",
      "ingestionTime": 1658528745414
    }
  ],
  "nextForwardToken": "f/36986426953797707963335499204463414460239026137054642176/s",
  "nextBackwardToken": "b/36986426953797707963335499204463414460239026137054642176/s"
}
```

This error log indicates that "One or more parameter values were invalid: Missing the key pk in the item". Since this import job failed, the table "target-table" now exists and is empty because no items were imported. The first item was processed and the object failed Item Validation.

To fix the issue, first delete "target-table" if it is no longer needed. Then either use a primary key column name that exists in the source object, or update the source data to:

```
pk,title,artist,year_of_release
Albums::Rock::Classic::1973::AlbumId::ALB25,The Dark Side of the Moon,Pink Floyd,1973
```

Target table exists

When you start an import job and receive a response as follows:

```
An error occurred (ResourceInUseException) when calling the ImportTable operation:
Table already exists: target-table
```

To fix this error, you will need to choose a table name that doesn't already exist and retry the import.

The specified bucket does not exist

If the source bucket does not exist, the import will fail and log the error message details in CloudWatch.

Example describe import:

```
aws dynamodb --endpoint-url $ENDPOINT describe-import --import-arn "arn:aws:dynamodb:us-east-1:531234567890:table/target-table/import/01658530687105-e6035287"
{
  "ImportTableDescription": {
    "ImportArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table/import/01658530687105-e6035287",
    "ImportStatus": "FAILED",
    "TableArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table",
    "TableId": "e1215a82-b8d1-45a8-b2e2-14b9dd8eb99c",
    "ClientToken": "3048e16a-069b-47a6-9dfb-9c259fd2fb6f",
    "S3BucketSource": {
      "S3BucketOwner": "531234567890",
      "S3Bucket": "BUCKET_DOES_NOT_EXIST",
      "S3KeyPrefix": "import-test"
    },
    "ErrorCount": 0,
    "CloudWatchLogGroupArn": "arn:aws:logs:us-east-1:ACCOUNT:log-group:/aws-dynamodb/imports:*",
    "InputFormat": "CSV",
    "InputCompressionType": "NONE",
    "TableCreationParameters": {
      "TableName": "target-table",
    }
  }
}
```

```
"AttributeDefinitions": [
    {
        "AttributeName": "pk",
        "AttributeType": "S"
    }
],
"KeySchema": [
    {
        "AttributeName": "pk",
        "KeyType": "HASH"
    }
],
"BillingMode": "PAY_PER_REQUEST"
},
"StartTime": 1658530687.105,
"EndTime": 1658530701.873,
"ProcessedSizeBytes": 0,
"ProcessedItemCount": 0,
"ImportedItemCount": 0,
"FailureCode": "S3NoSuchBucket",
"FailureMessage": "The specified bucket does not exist (Service: Amazon S3; Status Code: 404; Error Code: NoSuchBucket; Request ID: Q4W6QYYFDWY6WAKH; S3 Extended Request ID: ObqSlLeIMJpQqHLRX2C5Sy7n+8g6iGPwy7ixg7eEeTuEkg/+chU/JF+RbliWytMlkUlUcuCLTrI=; Proxy: null)"
}
}
```

The `FailureCode` is `S3NoSuchBucket`, with `FailureMessage` containing details such as request id and the service that threw the error. Since the error was caught before the data was imported into the table, a new DynamoDB table is not created. In some cases, when these errors are encountered after the data import has started, the table with partially imported data is retained.

To fix this error, make sure that the source Amazon S3 bucket exists and then restart the import process.

Best practices for importing from Amazon S3 into DynamoDB

The following are the best practices for importing data from Amazon S3 into DynamoDB.

Stay under the limit of 50,000 S3 objects

Each import job supports a maximum of 50,000 S3 objects. If your dataset contains more than 50,000 objects, consider consolidating them into larger objects.

Avoid excessively large S3 objects

S3 objects are imported in parallel. Having numerous mid-sized S3 objects allows for parallel execution without excessive overhead. For items under 1 KB, consider placing 4,000,000 items into each S3 object. If you have a larger average item size, place proportionally fewer items into each S3 object.

Randomize sorted data

If an S3 object holds data in sorted order, it can create a *rolling hot partition*. This is a situation where one partition receives all the activity, and then the next partition after that, and so on. Data in sorted order is defined as items in sequence in the S3 object that will be written to the same target partition during the import. One common situation where data is in sorted order is a CSV file where items are sorted by partition key so that repeated items share the same partition key.

To avoid a rolling hot partition, we recommend that you randomize the order in these cases. This can improve performance by spreading the write operations. For more information, see [Distributing write activity efficiently during data upload](#).

Compress data to keep the total S3 object size below the Regional limit

In the [import from S3 process](#), there is a limit on the sum total size of the S3 object data to be imported. The limit is 15 TB in the us-east-1, us-west-2, and eu-west-1 Regions, and 1 TB in all other Regions. The limit is based on the raw S3 object sizes.

Compression allows more raw data to fit within the limit. If compression alone isn't sufficient to fit the import within the limit, you can also contact [AWS Premium Support](#) for a quota increase.

Be aware of how item size impacts performance

If your average item size is very small (below 200 bytes), the import process might take a little longer than for larger item sizes.

Consider importing without any Global Secondary Indexes

The duration of an import task may depend on the presence of one or multiple global secondary indexes (GSIs). If you plan to establish indexes with partition keys that have low cardinality, you may see a faster import if you defer index creation until after the import task is finished (rather than including them in the import job).

Note

Creating a GSI during the import does not incur write charges (creating a GSI after the import would).

DynamoDB data export to Amazon S3: how it works

DynamoDB export to S3 is a fully managed solution for exporting your DynamoDB data to an Amazon S3 bucket at scale. Using DynamoDB export to S3, you can export data from an Amazon DynamoDB table from any time within your [point-in-time recovery \(PITR\)](#) window to an Amazon S3 bucket. You need to enable PITR on your table to use the export functionality. This feature enables you to perform analytics and complex queries on your data using other AWS services such as Athena, AWS Glue, Amazon SageMaker, Amazon EMR, and AWS Lake Formation.

DynamoDB export to S3 allows you to export both full and incremental data from your DynamoDB table. Exports do not consume any [read capacity units \(RCUs\)](#) and have no impact on table performance and availability. The export file formats supported are DynamoDB JSON and Amazon Ion formats. You can also export data to an S3 bucket owned by another AWS account and to a different AWS region. Your data is always encrypted end-to-end.

DynamoDB full exports are charged based on the size of the DynamoDB table (table data and local secondary indexes) at the point in time for which the export is done. DynamoDB incremental exports are charged based on the size of data processed from your continuous backups for the time period being exported. Additional charges apply for storing exported data in Amazon S3 and for PUT requests made against your Amazon S3 bucket. For more information about these charges, see [Amazon DynamoDB pricing](#) and [Amazon S3 pricing](#).

For specifics on service quotas, see [Table export to Amazon S3](#).

Topics

- [Requesting a table export in DynamoDB](#)
- [DynamoDB table export output format](#)

Requesting a table export in DynamoDB

DynamoDB table exports allow you to export table data to an Amazon S3 bucket, enabling you to perform analytics and complex queries on your data using other AWS services such as Athena, AWS

Glue, Amazon SageMaker, Amazon EMR, and AWS Lake Formation. You can request a table export using the AWS Management Console, the AWS CLI, or the DynamoDB API.

DynamoDB supports both full export and incremental export:

- With **full exports**, you can export a full snapshot of your table from any point in time within the point-in-time recovery (PITR) window to your Amazon S3 bucket.
- With **incremental exports**, you can export data from your DynamoDB table that was changed, updated, or deleted between a specified time period, within your PITR window, to your Amazon S3 bucket.

Topics

- [Prerequisites](#)
- [Requesting an export using the AWS Management Console](#)
- [Getting details about past exports in the AWS Management Console](#)
- [Requesting an export using the AWS CLI](#)
- [Getting details about past exports in the AWS CLI](#)
- [Requesting an export using the AWS SDK](#)
- [Getting details about past exports using the AWS SDK](#)

Prerequisites

Enable PITR

To use the export to S3 feature, you will need to enable PITR on your table. For details on how to enable PITR, see [Point-in-time recovery](#). If you request an export for a table that does not have PITR enabled, your request will fail with an exception message: "An error occurred (PointInTimeRecoveryUnavailableException) when calling the ExportTableToPointInTime operation: Point in time recovery is not enabled for table 'my-dynamodb-table'".

Set up S3 permissions

You can export your table data to any Amazon S3 bucket you have permission to write to. The destination bucket does not need to be in the same AWS Region or have the same owner as the source table owner. Your AWS Identity and Access Management (IAM) policy needs to allow you to be able to perform S3 actions (`s3:AbortMultipartUpload`,

`s3:PutObject`, and `s3:PutObjectAcl`) and the DynamoDB export action (`dynamodb:ExportTableToPointInTime`). Here is an example of a sample policy that will grant your user permissions to perform exports to an S3 bucket.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowDynamoDBExportAction",  
            "Effect": "Allow",  
            "Action": "dynamodb:ExportTableToPointInTime",  
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table"  
        },  
        {  
            "Sid": "AllowWriteToDestinationBucket",  
            "Effect": "Allow",  
            "Action": [  
                "s3:AbortMultipartUpload",  
                "s3:PutObject",  
                "s3:PutObjectAcl"  
            ],  
            "Resource": "arn:aws:s3:::your-bucket/*"  
        }  
    ]  
}
```

If you need to write to an S3 bucket that is in another account or you don't have permissions to write to, the S3 bucket owner will need to add a bucket policy to allow you to export from DynamoDB to that bucket. Here is an example policy on the target S3 bucket.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ExampleStatement",  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "arn:aws:iam::123456789012:user/Dave"  
            },  
            "Action": [  
                "s3:AbortMultipartUpload",  
                "s3:PutObject",  
                "s3:PutObjectAcl"  
            ]  
        }  
    ]  
}
```

```
        "s3:PutObjectAcl"
    ],
    "Resource": "arn:aws:s3:::awsexamplebucket1/*"
}
]
```

Revoking these permissions while an export is in progress will result in partial files.

 **Note**

If the table or bucket you're exporting to is encrypted with customer managed keys, that KMS key's policies must give DynamoDB permission to use it. This permission is given through the IAM User/Role that triggers the export job. For more information on encryption including best practices, see [How DynamoDB uses AWS KMS](#) and [Using a custom KMS key](#).

Requesting an export using the AWS Management Console

The following example demonstrates how to use the DynamoDB console to export an existing table named `MusicCollection`.

 **Note**

This procedure assumes that you have enabled point-in-time recovery. To enable it for the `MusicCollection` table, on the table's **Overview** tab, in the **Table details** section, choose **Enable for Point-in-time recovery**.

To request a table export

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Exports to S3**.
3. Select the **Export to S3** button.
4. Choose a source table and destination S3 bucket. If the destination bucket is owned by your account, you can use the **Browse S3** button to find it. Otherwise, enter the URL of the bucket

using the `s3://bucketname/prefix` format. The **prefix** is an optional folder to help keep your destination bucket organized.

5. Choose **Full export** or **Incremental export**. A **full export** outputs the full table snapshot of your table as it was at the point in time you specify. An **incremental export** outputs the changes made to your table during the specified export period. Your output is compacted such that it will only contain the final state of the item from the export period. The item will only appear once in the export even if it has multiple updates within the same export period.

Full export

1. Select the point in time you want to export the full table snapshot from. This can be any point in time within the PITR window. Alternatively, you can select **Current time** to export the latest snapshot.

Export settings

Full export
Export the table data in its current state, or from any specific point up to 35 days ago.

Incremental export
Export any table data that's changed within a specific time period.

Export from a specific point in time | [Info](#)

Current time

Export from an earlier point in time
Your earliest export point is the same as the earliest restore point for your table.

2023/09/01 [Calendar icon](#) 12:00:00 (UTC+01:00)

For date, use YYYY/MM/DD format. For time, use 24-hour format.

2. For **Exported file format**, choose between **DynamoDB JSON** and **Amazon Ion**. By default, your table will be exported in DynamoDB JSON format from the latest restorable time in the point in time recovery window and encrypted using an Amazon S3 key (SSE-S3). You can change these export settings if necessary.

 **Note**

If you choose to encrypt your export using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination S3 bucket.

Exported file format | [Info](#)

DynamoDB JSON

Amazon Ion

Open-source text format, which is a superset of JSON.

Incremental export

1. Select the **Export period** you want to export the incremental data for. Pick a start time within the PITR window. The export period duration must be at least 15 minutes and be no longer than 24 hours. The export period's start time is inclusive and the end time is exclusive.

Export settings

Full export

Export the table data in its current state, or from any specific point up to 35 days ago.

Incremental export

Export any table data that's changed within a specific time period.

Export period

Specify when the incremental export starts and ends. Your earliest export point is the same as the earliest restore point for your table.

 2023-09-01T12:00:00+01:00 — 2023-09-02T12:00:00+01:00

2. Choose between **Absolute mode** or **Relative mode**.

- a. **Absolute mode** will export incremental data for the time period you specify.

Relative modeAbsolute mode

<August 2023September 2023>

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
	1	2	3	4	5	6					1	2	3
7	8	9	10	11	12	13	4	5	6	7	8	9	10
14	15	16	17	18	19	20	11	12	13	14	15	16	17
21	22	23	24	25	26	27	18	19	20	21	22	23	24
28	29	30	31				25	26	27	28	29	30	

Start date	Start time	End date	End time
2023/09/01	12:00:00	2023/09/02	12:00:00

The export period must be between 15 minutes and 24 hours. For date, use YYYY/MM/DD. For time, use 24 hr format.

ClearCancelApply

- b. **Relative mode** will export incremental data for an export period that is relative to your export job submission time.

Relative mode**Absolute mode**

Choose a range

- Last 1 hour
- Last 6 hours
- Last 12 hours
- Last 24 hours
- Custom range

Set a custom range in the past

[Clear](#)[Cancel](#)[Apply](#)

3. For **Exported file format**, choose between **DynamoDB JSON** and **Amazon Ion**. By default, your table will be exported in DynamoDB JSON format from the latest restorable time in the point in time recovery window and encrypted using an Amazon S3 key (SSE-S3). You can change these export settings if necessary.

Note

If you choose to encrypt your export using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination S3 bucket.

Exported file format | [Info](#) **DynamoDB JSON** **Amazon Ion**

Open-source text format, which is a superset of JSON.

4. For **Export view type**, select either **New and old images** or **New images only**. New image provides the latest state of the item. Old image provides the state of the item right before the specified “start date and time”. The default setting is **New and old images**. For more information on new images and old images, see [Incremental export output](#).

Export view type

- New and old images**
- New images only**

6. Click the **Export** button to begin the export.

Exported data is not transactionally consistent. This means that your transaction operations can be torn between two export outputs. You may see a subset of items modified by a transaction operation reflected in the export, while another subset of modifications in the same transaction is not reflected in the same export request. However, exports are eventually consistent. If a transaction is torn during an export, you are guaranteed to have the remaining transaction in your next contiguous export, with no duplicates. The time periods used for exports are based on an internal system clock and can vary by one minute of your application's local clock.

Getting details about past exports in the AWS Management Console

You can find information about export tasks you've run in the past by clicking the **Exports to S3** section in the navigation sidebar. This section contains a list of all exports you've created in the past 90 days. Selecting the ARN of a task listed in the Exports tab will retrieve information about that export, including any advanced configuration settings you chose. Note that although export task metadata expires after 90 days and jobs older than that are no longer found in this list, the objects in your S3 bucket remain as long as their bucket policies allow. DynamoDB never deletes any of the objects it creates in your S3 bucket during an export.

Requesting an export using the AWS CLI

The following example shows how to use the AWS CLI to export an existing table named `MusicCollection` to an S3 bucket called `ddb-export-musiccollection`.

Note

This procedure assumes that you have enabled point-in-time recovery. To enable it for the MusicCollection table, run the following command.

```
aws dynamodb update-continuous-backups \
--table-name MusicCollection \
--point-in-time-recovery-specification PointInTimeRecoveryEnabled=True
```

Full export

The following command exports the MusicCollection to an S3 bucket called ddb-export-musiccollection-9012345678 with a prefix of 2020-Nov. Table data will be exported in DynamoDB JSON format from a specific time within the point in time recovery window and encrypted using an Amazon S3 key (SSE-S3).

Note

If requesting a cross-account table export, make sure to include the `--s3-owner` option.

```
aws dynamodb export-table-to-point-in-time \
--table-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
--s3-bucket ddb-export-musiccollection-9012345678 \
--s3-prefix 2020-Nov \
--export-format DYNAMODB_JSON \
--export-time 1604632434 \
--s3-owner 9012345678 \
--s3-sse-algorithm AES256
```

Incremental export

The following command performs an incremental export by providing a new `--export-type` and `--incremental-export-specification`. Substitute your own values for anything in *italics*. Times are specified as seconds since epoch.

```
aws dynamodb export-table-to-point-in-time \
```

```
--table-arn arn:aws:dynamodb:REGION:ACCOUNT:table/TABLENAME \
--s3-bucket BUCKET --s3-prefix PREFIX \
--incremental-export-specification
ExportFromTime=1693569600,ExportToTime=1693656000,ExportViewType=NEW_AND_OLD_IMAGES
\
--export-type INCREMENTAL_EXPORT
```

Note

If you choose to encrypt your export using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination S3 bucket.

Getting details about past exports in the AWS CLI

You can find information about export requests you've run in the past by using the `list-exports` command. This command returns a list of all exports you've created in the past 90 days. Note that although export task metadata expires after 90 days and jobs older than that are no longer returned by the `list-exports` command, the objects in your S3 bucket remain as long as their bucket policies allow. DynamoDB never deletes any of the objects it creates in your S3 bucket during an export.

Exports have a status of PENDING until they either succeed or fail. If they succeed, the status will change to COMPLETED. If they fail, the status will change to FAILED with a `failure_message` and `failure_reason`.

In the following example, we use the optional `table-arn` parameter to list only exports of a specific table.

```
aws dynamodb list-exports \
--table-arn arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog
```

To retrieve detailed information about a specific export task, including any advanced configuration settings, use the `describe-export` command.

```
aws dynamodb describe-export \
--export-arn arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/
export/01234567890123-a1b2c3d4
```

Requesting an export using the AWS SDK

Use these code snippets to request a table export using the AWS SDK of your choice.

Python

Full export

```
import boto3
from datetime import datetime

# remove endpoint_url for real use
client = boto3.client('dynamodb')

# https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/
# dynamodb/client/export_table_to_point_in_time.html
client.export_table_to_point_in_time(
    TableName='arn:aws:dynamodb:us-east-1:0123456789:table/TABLE',
    ExportTime=datetime(2023, 9, 20, 12, 0, 0),
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
    ExportFormat='DYNAMODB_JSON'
)
```

Incremental export

```
import boto3
from datetime import datetime

client = boto3.client('dynamodb')

client.export_table_to_point_in_time(
    TableName='arn:aws:dynamodb:us-east-1:0123456789:table/TABLE',
    IncrementalExportSpecification={
        'ExportFromTime': datetime(2023, 9, 20, 12, 0, 0),
        'ExportToTime': datetime(2023, 9, 20, 13, 0, 0),
        'ExportViewType': 'NEW_AND_OLD_IMAGES'
    },
    ExportType='INCREMENTAL_EXPORT',
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
```

```
    ExportFormat='DYNAMODB_JSON'  
)
```

Getting details about past exports using the AWS SDK

Use these code snippets to get details about past table exports using the AWS SDK of your choice.

Python

Full export

```
import boto3  
  
client = boto3.client('dynamodb')  
  
# https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb/client/list\_exports.html  
  
print(  
    client.list_exports(  
        TableArn='arn:aws:dynamodb:us-east-1:0123456789:table/TABLE',  
    )  
)
```

Incremental export

```
import boto3  
  
client = boto3.client('dynamodb')  
  
# https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb/client/describe\_export.html  
  
print(  
    client.describe_export(  
        ExportArn='arn:aws:dynamodb:us-east-1:0123456789:table/TABLE/  
export/01695353076000-06e2188f',  
        )['ExportDescription'])
```

DynamoDB table export output format

A DynamoDB table export includes manifest files in addition to the files containing your table data. These files are all saved in the Amazon S3 bucket that you specify in your [export request](#). The following sections describe the format and contents of each output object.

Full export output

Manifest files

DynamoDB creates manifest files, along with their checksum files, in the specified S3 bucket for each export request.

```
export-prefix/AWSDynoDB/ExportId/manifest-summary.json  
export-prefix/AWSDynoDB/ExportId/manifest-summary.checksum  
export-prefix/AWSDynoDB/ExportId/manifest-files.json  
export-prefix/AWSDynoDB/ExportId/manifest-files.checksum
```

You choose an **export-prefix** when you request a table export. This helps you keep files in the destination S3 bucket organized. The **ExportId** is a unique token generated by the service to ensure that multiple exports to the same S3 bucket and **export-prefix** don't overwrite each other.

The export creates at least 1 file per partition. For partitions that are empty, your export request will create an empty file. All of the items in each file are from that particular partition's hashed keyspace.

Note

DynamoDB also creates an empty file named `_started` in the same directory as the manifest files. This file verifies that the destination bucket is writable and that the export has begun. It can safely be deleted.

The summary manifest

The `manifest-summary.json` file contains summary information about the export job. This allows you to know which data files in the shared data folder are associated with this export. Its format is as follows:

```
{  
    "version": "2020-06-30",  
    "exportArn": "arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/  
export/01234567890123-a1b2c3d4",  
    "startTime": "2020-11-04T07:28:34.028Z",  
    "endTime": "2020-11-04T07:33:43.897Z",  
    "tableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog",  
    "tableId": "12345a12-abcd-123a-ab12-1234abc12345",  
    "exportTime": "2020-11-04T07:28:34.028Z",  
    "s3Bucket": "ddb-productcatalog-export",  
    "s3Prefix": "2020-Nov",  
    "s3SseAlgorithm": "AES256",  
    "s3SseKmsKeyId": null,  
    "manifestFilesS3Key": "AWSDynoDB/01693685827463-2d8752fd/manifest-files.json",  
    "billedSizeBytes": 0,  
    "itemCount": 8,  
    "outputFormat": "DYNAMODB_JSON",  
    "exportType": "FULL_EXPORT"  
}
```

The files manifest

The `manifest-files.json` file contains information about the files that contain your exported table data. The file is in [JSON lines](#) format, so newlines are used as item delimiters. In the following example, the details of one data file from a files manifest are formatted on multiple lines for the sake of readability.

```
{  
    "itemCount": 8,  
    "md5Checksum": "sQMSpEILNgoQmarvDFonGQ==",  
    "etag": "af83d6f217c19b8b0fff8023d8ca4716-1",  
    "dataFileS3Key": "AWSDynoDB/01693685827463-2d8752fd/data/asdl123dasas.json.gz"  
}
```

Data files

DynamoDB can export your table data in two formats: DynamoDB JSON and Amazon Ion. Regardless of the format you choose, your data will be written to multiple compressed files named by the keys. These files are also listed in the `manifest-files.json` file.

The directory structure of your S3 bucket after a full export will contain all of your manifest files and data files under the export Id folder.

DestinationBucket/DestinationPrefix

```
.  
### AWSDynamoDB  
### 01693685827463-2d8752fd      // the single full export  
# ### manifest-files.json        // manifest points to files under 'data' subfolder  
# ### manifest-files.checksum  
# ### manifest-summary.json     // stores metadata about request  
# ### manifest-summary.md5  
# ### data                      // The data exported by full export  
# # ### asdl123dasas.json.gz  
# # ...  
# ### _started                  // empty file for permission check
```

DynamoDB JSON

A table export in DynamoDB JSON format consists of multiple Item objects. Each individual object is in DynamoDB's standard marshalled JSON format.

When creating custom parsers for DynamoDB JSON export data, the format is [JSON lines](#). This means that newlines are used as item delimiters. Many AWS services, such as Athena and AWS Glue, will parse this format automatically.

In the following example, a single item from a DynamoDB JSON export has been formatted on multiple lines for the sake of readability.

```
{  
    "Item":{  
        "Authors":{  
            "SS":[  
                "Author1",  
                "Author2"  
            ]  
        },  
        "Dimensions":{  
            "S":"8.5 x 11.0 x 1.5"  
        },  
        "ISBN":{  
            "S":"333-333333333"  
        },  
        "Id":{  
            "N":"103"  
        },  
        "InPublication":{  
            "B":true  
        }  
    }  
}
```

```
        "BOOL":false
    },
    "PageCount":{
        "N":"600"
    },
    "Price":{
        "N":"2000"
    },
    "ProductCategory":{
        "S":"Book"
    },
    "Title":{
        "S":"Book 103 Title"
    }
}
}
```

Amazon Ion

[Amazon Ion](#) is a richly-typed, self-describing, hierarchical data serialization format built to address rapid development, decoupling, and efficiency challenges faced every day while engineering large-scale, service-oriented architectures. DynamoDB supports exporting table data in Ion's [text format](#), which is a superset of JSON.

When you export a table to Ion format, the DynamoDB datatypes used in the table are mapped to [Ion datatypes](#). DynamoDB sets use [Ion type annotations](#) to disambiguate the datatype used in the source table.

DynamoDB to ion datatype conversion

DynamoDB data type	Ion representation
String (S)	string
Boolean (BOOL)	bool
Number (N)	decimal
Binary (B)	blob
Set (SS, NS, BS)	list (with type annotation \$dynamodb_SS, \$dynamodb_NS, or \$dynamodb_BS)

DynamoDB data type	Ion representation
List	list
Map	struct

Items in an Ion export are delimited by newlines. Each line begins with an Ion version marker, followed by an item in Ion format. In the following example, an item from an Ion export has been formatted on multiple lines for the sake of readability.

```
$ion_1_0 {
    Item: {
        Authors:$dynamodb_SS::["Author1", "Author2"],
        Dimensions:"8.5 x 11.0 x 1.5",
        ISBN:"333-333333333",
        Id:103.,
        InPublication:false,
        PageCount:6d2,
        Price:2d3,
        ProductCategory:"Book",
        Title:"Book 103 Title"
    }
}
```

Incremental export output

Manifest files

DynamoDB creates manifest files, along with their checksum files, in the specified S3 bucket for each export request.

```
export-prefix/AWSDynoDB/ExportId/manifest-summary.json
export-prefix/AWSDynoDB/ExportId/manifest-summary.checksum
export-prefix/AWSDynoDB/ExportId/manifest-files.json
export-prefix/AWSDynoDB/ExportId/manifest-files.checksum
```

You choose an **export-prefix** when you request a table export. This helps you keep files in the destination S3 bucket organized. The **ExportId** is a unique token generated by the service to ensure that multiple exports to the same S3 bucket and export-prefix don't overwrite each other.

The export creates at least 1 file per partition. For partitions that are empty, your export request will create an empty file. All of the items in each file are from that particular partition's hashed keyspace.

Note

DynamoDB also creates an empty file named `_started` in the same directory as the manifest files. This file verifies that the destination bucket is writable and that the export has begun. It can safely be deleted.

The summary manifest

The `manifest-summary.json` file contains summary information about the export job. This allows you to know which data files in the shared data folder are associated with this export. Its format is as follows:

```
{  
  "version": "2023-08-01",  
  "exportArn": "arn:aws:dynamodb:us-east-1:599882009758:table/export-test/  
  export/01695097218000-d6299cbd",  
  "startTime": "2023-09-19T04:20:18.000Z",  
  "endTime": "2023-09-19T04:40:24.780Z",  
  "tableArn": "arn:aws:dynamodb:us-east-1:599882009758:table/export-test",  
  "tableId": "b116b490-6460-4d4a-9a6b-5d360abf4fb3",  
  "exportFromTime": "2023-09-18T17:00:00.000Z",  
  "exportToTime": "2023-09-19T04:00:00.000Z",  
  "s3Bucket": "jason-exports",  
  "s3Prefix": "20230919-prefix",  
  "s3SseAlgorithm": "AES256",  
  "s3SseKmsKeyId": null,  
  "manifestFilesS3Key": "20230919-prefix/AWSDynoDB/01693685934212-ac809da5/manifest-  
  files.json",  
  "billedSizeBytes": 20901239349,  
  "itemCount": 169928274,  
  "outputFormat": "DYNAMODB_JSON",  
  "outputView": "NEW_AND_OLD_IMAGES",  
  "exportType": "INCREMENTAL_EXPORT"  
}
```

The files manifest

The `manifest-files.json` file contains information about the files that contain your exported table data. The file is in [JSON lines](#) format, so newlines are used as item delimiters. In the following example, the details of one data file from a files manifest are formatted on multiple lines for the sake of readability.

```
{  
  "itemCount": 8,  
  "md5Checksum": "sQMSpEILNgoQmarvDFonGQ==",  
  "etag": "af83d6f217c19b8b0fff8023d8ca4716-1",  
  "dataFileS3Key": "AWSDynoDB/data/sgad6417s6vss4p7owp0471bcq.json.gz"  
}
```

Data files

DynamoDB can export your table data in two formats: DynamoDB JSON and Amazon Ion. Regardless of the format you choose, your data will be written to multiple compressed files named by the keys. These files are also listed in the `manifest-files.json` file.

The data files for incremental exports are all contained in a common data folder in your S3 bucket. Your manifest files are under your export ID folder.

```
DestinationBucket/DestinationPrefix  
. . .  
### AWSDynoDB  
### 01693685934212-ac809da5      // an incremental export ID  
#   ### manifest-files.json      // manifest points to files under 'data' folder  
#   ### manifest-files.checksum  
#   ### manifest-summary.json    // stores metadata about request  
#   ### manifest-summary.md5  
#   ### _started                // empty file for permission check  
### 01693686034521-ac809da5  
#   ### manifest-files.json  
#   ### manifest-files.checksum  
#   ### manifest-summary.json  
#   ### manifest-summary.md5  
#   ### _started  
### data                      // stores all the data files for incremental  
exports  
#   ### sgad6417s6vss4p7owp0471bcq.json.gz  
#   ...
```

If you export files, each item's output includes a timestamp that represents when that item was updated in your table and a data structure that indicates if it was an `insert`, `update`, or `delete` operation. The timestamp is based on an internal system clock and can vary from your application clock. For incremental exports, you can choose between two export view types for your output structure: **new and old images** or **new images only**.

- **New image** provides the latest state of the item
- **Old image** provides the state of the item right before the specified **start date and time**

View types can be helpful if you want to see how the item was changed within the export period. It can also be useful for efficiently updating your downstream systems, especially if those downstream systems have a partition key that is not the same as your DynamoDB partition key.

You can infer whether an item in your incremental export output was an `insert`, `update`, or `delete` by looking at the structure of the output. The incremental export structure and its corresponding operations are summarized in the table below for both export view types.

Operation	New images only	New and old images
Insert	Keys + new image	Keys + new image
Update	Keys + new image	Key + new image + old image
Delete	Keys	Keys + old image
Insert + delete	No output	No output

DynamoDB JSON

A table export in DynamoDB JSON format consists of a metadata timestamp that indicates the write time of the item, followed by the keys of the item and the values. The following shows an example DynamoDB JSON output using export view type as **New and Old images**.

```
// Ex 1: Insert
// An insert means the item did not exist before the incremental export window
// and was added during the incremental export window

{
  "Metadata": {
```

```
"WriteTimestampMicros": "1680109764000000"
},
"Key": {
    "PK": {
        "S": "CUST#100"
    }
},
"NewImage": {
    "PK": {
        "S": "CUST#100"
    },
    "FirstName": {
        "S": "John"
    },
    "LastName": {
        "S": "Don"
    }
}
}

// Ex 2: Update
// An update means the item existed before the incremental export window
// and was updated during the incremental export window.
// The OldImage would not be present if choosing "New images only".

{
    "Metadata": {
        "WriteTimestampMicros": "1680109764000000"
    },
    "Key": {
        "PK": {
            "S": "CUST#200"
        }
    },
    "OldImage": {
        "PK": {
            "S": "CUST#200"
        },
        "FirstName": {
            "S": "Mary"
        },
        "LastName": {
            "S": "Grace"
        }
    }
}
```

```
},
  "NewImage": {
    "PK": {
      "S": "CUST#200"
    },
    "FirstName": {
      "S": "Mary"
    },
    "LastName": {
      "S": "Smith"
    }
  }
}

// Ex 3: Delete
// A delete means the item existed before the incremental export window
// and was deleted during the incremental export window
// The OldImage would not be present if choosing "New images only".

{
  "Metadata": {
    "WriteTimestampMicros": "1680109764000000"
  },
  "Key": {
    "PK": {
      "S": "CUST#300"
    }
  },
  "OldImage": {
    "PK": {
      "S": "CUST#300"
    },
    "FirstName": {
      "S": "Jose"
    },
    "LastName": {
      "S": "Hernandez"
    }
  }
}

// Ex 4: Insert + Delete
// Nothing is exported if an item is inserted and deleted within the
```

```
// incremental export window.
```

Amazon Ion

[Amazon Ion](#) is a richly-typed, self-describing, hierarchical data serialization format built to address rapid development, decoupling, and efficiency challenges faced every day while engineering large-scale, service-oriented architectures. DynamoDB supports exporting table data in Ion's [text format](#), which is a superset of JSON.

When you export a table to Ion format, the DynamoDB datatypes used in the table are mapped to [Ion datatypes](#). DynamoDB sets use [Ion type annotations](#) to disambiguate the datatype used in the source table.

DynamoDB to ion datatype conversion

DynamoDB data type	Ion representation
String (S)	string
Boolean (BOOL)	bool
Number (N)	decimal
Binary (B)	blob
Set (SS, NS, BS)	list (with type annotation \$dynamodb_SS, \$dynamodb_NS, or \$dynamodb_BS)
List	list
Map	struct

Items in an Ion export are delimited by newlines. Each line begins with an Ion version marker, followed by an item in Ion format. In the following example, an item from an Ion export has been formatted on multiple lines for the sake of readability.

```
$ion_1_0 {
    Record: {
        Keys: {
            ISBN: "333-3333333333"
```

```
        },
        Metadata:{
            WriteTimestampMicros:1684374845117899.
        },
        OldImage:{
            Authors:$dynamodb_SS::["Author1","Author2"],
            ISBN:"333-333333333",
            Id:103.,
            InPublication:false,
            ProductCategory:"Book",
            Title:"Book 103 Title"
        },
        NewImage:{
            Authors:$dynamodb_SS::["Author1","Author2"],
            Dimensions:"8.5 x 11.0 x 1.5",
            ISBN:"333-333333333",
            Id:103.,
            InPublication:true,
            PageCount:6d2,
            Price:2d3,
            ProductCategory:"Book",
            Title:"Book 103 Title"
        }
    }
}
```

DynamoDB zero-ETL integration with Amazon OpenSearch Service

Amazon DynamoDB offers a zero-ETL integration with Amazon OpenSearch Service through the **DynamoDB plugin for OpenSearch Ingestion**. Amazon OpenSearch Ingestion offers a fully managed, no-code experience for ingesting data into Amazon OpenSearch Service.

With the DynamoDB plugin for OpenSearch Ingestion, you can use one or more DynamoDB tables as a source for ingestion to one or more OpenSearch Service indexes. You can browse and configure your OpenSearch Ingestion pipelines with DynamoDB as a source from either OpenSearch Ingestion or DynamoDB Integrations in the AWS Management Console.

- Get started with OpenSearch Ingestion by following along in the [OpenSearch Ingestion getting started guide](#).

- Learn about the prerequisites and all the configuration options for the DynamoDB plugin at [DynamoDB plugin for OpenSearch Ingestion documentation](#).

How it works

The plugin uses [DynamoDB export to Amazon S3](#) to create an initial snapshot to load into OpenSearch. After the snapshot has been loaded, the plugin uses DynamoDB Streams to replicate any further changes in near real time. Every item is processed as an event in OpenSearch Ingestion and can be modified with processor plugins. You can drop attributes or create composite attributes and send them to different indexes through routes.

You must have [point-in-time recovery \(PITR\)](#) enabled to use export to Amazon S3. You must also have [DynamoDB Streams](#) enabled (with the **new & old images** option selected) to be able to use it. It's possible to create a pipeline without taking a snapshot by excluding export settings.

You can also create a pipeline with only a snapshot and no updates by excluding streams settings. The plugin does not use read or write throughput on your table, so it is safe to use without impacting your production traffic. There are limits to the number of parallel consumers on a stream that you should consider before creating this or other integrations. For other considerations, see [the section called "Integration best practices"](#).

For simple pipelines, a single OpenSearch Compute Unit (OCU) can process about 1 MB per second of writes. This is the equivalent of about 1000 write request units (WCU). Depending on your pipeline's complexity and other factors, you might achieve more or less than this.

OpenSearch Ingestion supports a dead-letter queue (DLQ) for events that cause unrecoverable errors. Additionally, the pipeline can resume from where it left off without user intervention even if there's an interruption of service with either DynamoDB, the pipeline, or Amazon OpenSearch Service.

If interruption goes on for longer than 24 hours, this can cause a loss of updates. However, the pipeline would continue to process the updates that were still available when availability is restored. You would need to do a fresh index build to fix any irregularities due to the dropped events unless they were in the dead-letter queue.

For all the settings and details for the plugin, see [OpenSearch Ingestion DynamoDB plugin documentation](#).

Integrated create experience through the console

DynamoDB and OpenSearch Service have an integrated experience in the AWS Management Console, which streamlines the getting started process. When you go through these steps, the service will automatically select the DynamoDB blueprint and add the appropriate DynamoDB information for you.

To create an integration, follow along in the [OpenSearch Ingestion getting started guide](#). When you get to [Step 3: Create a pipeline](#), replace Steps 1 and 2 with the following steps:

1. Navigate to the DynamoDB console.
2. In the left-hand navigation pane, choose **Integration**.
3. Select the DynamoDB table that you'd like to replicate to OpenSearch.
4. Choose **Create**.

From here, you can continue on with the rest of the tutorial.

Next steps

For a better understanding of how DynamoDB integrates with OpenSearch Service, see the following:

- [Getting started with Amazon OpenSearch Ingestion](#)
- [DynamoDB plugin configuration and requirements](#)

Handling breaking changes to your index

OpenSearch can dynamically add new attributes to your index. However, after your mapping template has been set for a given key, you'll need to take additional action to change it. Additionally, if your change requires you to reprocess all the data in your DynamoDB table, you'll need to take steps to initiate a fresh export.

Note

In all these options, you might still run into issues if your DynamoDB table has type conflicts with the mapping template you've specified. Ensure that you have a dead-letter queue (DLQ) enabled (even in development). This makes it easier to understand what might

be wrong with the record that causes a conflict when it's being indexed into your index on OpenSearch.

Topics

- [How it works](#)
- [Delete your index and reset the pipeline \(pipeline-centric option\)](#)
- [Recreate your index and reset the pipeline \(index-centric option\)](#)
- [Create a new index and sink \(online option\)](#)
- [Best practices for avoiding and debugging type conflicts](#)

How it works

Here's a quick overview of the actions taken when handling breaking changes to your index. See the step-by-step procedures in the sections that follow.

- **Stop and start the pipeline:** This option resets the pipeline's state, and the pipeline will restart with a new full export. It is non-destructive, so it does **not** delete your index or any data in DynamoDB. If you don't create a fresh index before you do this, you might see a high number of errors from version conflicts because the export tries to insert older documents than the current `_version` in the index. You can safely ignore these errors. You will not be billed for the pipeline while it is stopped.
- **Update the pipeline:** This option updates the configuration in the pipeline with a [blue/green](#) approach, without losing any state. If you make significant changes to your pipeline (such as adding new routes, indexes, or keys to existing indexes), you might need to do a full reset of the pipeline and recreate your index. This option does **not** perform a full export.
- **Delete and recreate the index:** This option removes your data and mapping settings on your index. You should do this before making any breaking changes to your mappings. It will break any applications that rely on the index until the index is recreated and synchronized. Deleting the index does **not** initiate a fresh export. You should delete your index only after you've updated your pipeline. Otherwise, your index might be recreated before you update your settings.

Delete your index and reset the pipeline (pipeline-centric option)

This method is often the fastest option if you're still in development. You'll delete your index in OpenSearch Service, and then [stop and start](#) your pipeline to initiate a fresh export of all your data. This ensures that there are no mapping template conflicts with existing indexes, and no loss of data from an incomplete processed table.

1. Stop the pipeline either through the AWS Management Console, or by using the `StopPipeline` API operation with the AWS CLI or an SDK.
2. [Update your pipeline configuration](#) with your new changes.
3. Delete your index in OpenSearch Service, either through a REST API call or your OpenSearch Dashboard.
4. Start the pipeline either through the console, or by using the `StartPipeline` API operation with the AWS CLI or an SDK.

 **Note**

This initiates a fresh full export, which will incur additional costs.

5. Monitor for any unexpected issues because a fresh export is generated to create the new index.
6. Confirm that the index matches your expectations in OpenSearch Service.

After the export has completed and it resumes reading from the stream, your DynamoDB table data will now be available in the index.

Recreate your index and reset the pipeline (index-centric option)

This method works well if you need to do a lot of iterations on the index design in OpenSearch Service before resuming the pipeline from DynamoDB. This can be useful for development when you want to iterate very quickly on your search patterns, and want to avoid waiting on fresh exports to complete between each iteration.

1. Stop the pipeline either through the AWS Management Console, or by calling the `StopPipeline` API operation with the AWS CLI or an SDK.
2. Delete and recreate your index in OpenSearch with the mapping template you want to use. You can manually insert some sample data to confirm that your searches are working as

- intended. If your sample data might conflict with any data from DynamoDB, be sure to delete it before moving onto the next step.
3. If you have an indexing template in your pipeline, remove it or replace it with the one you've created already in OpenSearch Service. Ensure that the name of your index matches the name in the pipeline.
 4. Start the pipeline either through console, or by calling the `StartPipeline` API operation with the AWS CLI or an SDK.

 **Note**

This will initiate a fresh full export, which will incur additional costs.

5. Monitor for any unexpected issues because a fresh export is generated to create the new index.

After the export has completed and it resumes reading from the stream, you should be your DynamoDB table data will now be available in the index.

Create a new index and sink (online option)

This method works well if you need to update your mapping template but are currently using your index in production. This creates a brand new index, which you'll need to move your application over to after it's synchronized and validated.

 **Note**

This will create another consumer on the stream. This can be an issue if you also have other consumers like AWS Lambda or global tables. You might need to pause updates to your existing pipeline to create capacity to load the new index.

1. [Create a new pipeline](#) with new settings and a different index name.
2. Monitor the new index for any unexpected issues.
3. Swap the application over to the new index.
4. Stop and delete the old pipeline after validating that everything is working correctly.

Best practices for avoiding and debugging type conflicts

- Always use a dead-letter queue (DLQ) to make it easier to debug when there are type conflicts.
- Always use an index template with mappings and set `include_keys`. While OpenSearch Service dynamically maps new keys, this can cause issues with unexpected behaviors (such as expecting something to be a `GeoPoint`, but it's created as a `string` or `object`) or errors (such as having a number that is a mix of `long` and `float` values).
- If you need to keep your existing index working in production, you can also replace any of the previous [delete index steps](#) with just renaming your index in your pipeline config file. This creates a brand new index. Your application will then need to be updated to point to the new index after it's complete.
- If you have a type conversion issue that you fix with a processor, you can test this with `UpdatePipeline`. To do this, you'll need to do a stop and start or [process your dead-letter queues](#) to fix any previously skipped documents that had errors.

Best practices for integrating with DynamoDB

When integrating DynamoDB with other services, you should always follow the best practices for using each individual service. However, there are some best practices specific to integration that you should consider.

Topics

- [Creating a snapshot in DynamoDB](#)
- [Capturing data change in DynamoDB](#)
- [DynamoDB zero-ETL integration with OpenSearch Service](#)

Creating a snapshot in DynamoDB

- Generally, we recommend using [export to Amazon S3](#) to create snapshots for initial replication. It is both cost effective, and won't compete with your application's traffic for throughput. You can also consider a backup and restore to a new table followed by a scan operation. This will avoid competing for throughput with your application, but will generally be substantially less cost effective than an export.
- Always set a `StartTime` when doing an export. This makes it easy to determine where you'll start your change data capture (CDC) from.

- When using export to S3, set a lifecycle action on the S3 bucket. Typically, an expiration action set at 7 days is safe, but you should follow any guidelines that your company might have. Even if you explicitly delete your items after ingestion, this action can help catch issues, which helps reduce unnecessary costs and prevents policy violations.

Capturing data change in DynamoDB

- If you need near real-time CDC, use [DynamoDB Streams](#) or [Amazon Kinesis Data Streams \(KDS\)](#). When you're deciding which one to use, generally consider which is easiest to use with the downstream service. If you need to provide in-order event processing at a partition-key level, or if you have items that are exceptionally large, use DynamoDB Streams.
- If you don't need near real-time CDC, you can use [export to Amazon S3 with incremental exports](#) to export only the changes that have happened between two points in time.

If you used export to S3 for generating a snapshot, this can be especially helpful because you can use similar code to process incremental exports. Typically, export to S3 is slightly cheaper than the previous streaming options, but cost is typically not the main factor for which option to use.

- You can generally only have two simultaneous consumers of a DynamoDB stream. Consider this when planning your integration strategy.
- Don't use scans to detect changes. This might work on a small scale, but becomes impractical fairly quickly.

DynamoDB zero-ETL integration with OpenSearch Service

DynamoDB has a [DynamoDB zero-ETL integration with Amazon OpenSearch Service](#). For more information, see the [DynamoDB plugin for OpenSearch Ingestion](#) and [specific best practices for Amazon OpenSearch Service](#).

Configuration

- Only index data that you need to perform searches on. Always use a mapping template (`template_type: index_template` and `template_content`) and `include_keys` to implement this.
- Monitor your logs for errors that are related to type conflicts. OpenSearch Service expects all values for a given key to have the same type. It generates exceptions if there's a mismatch. If you

encounter one of these errors, you can add a processor to catch that a given key is always be the same value.

- Generally use the `primary_key` metadata value for the `document_id` value. In OpenSearch Service, the document ID is the equivalent of the primary key in DynamoDB. Using the primary key will make it easy to find your document and ensure that updates are consistently replicated to it without conflicts.

You can use the helper function `getMetadata` to get your primary key (for example, `document_id: "${getMetadata('primary_key')}"`). If you're using a composite primary key, the helper function will concatenate them together for you.

- In general, use the `opensearch_action` metadata value for the `action` setting. This will ensure that updates are replicated in such a way that the data in OpenSearch Service matches the latest state in DynamoDB.

You can use the helper function `getMetadata` to get your primary key (for example, `action: "${getMetadata('opensearch_action')}"`). You can also get the stream event type through `dynamodb_event_name` for use cases like filtering. However, you should typically not use it for the `action` setting.

Observability

- Always use a dead-letter queue (DLQ) on your OpenSearch sinks to handle dropped events. DynamoDB is generally less structured than OpenSearch Service, and it's always possible for something unexpected to happen. With a dead-letter queue, you can recover individual events, and even automate the recovery process. This will help you to avoid needing to rebuild your entire index.
- Always set alerts that your replication delay doesn't go over an expected amount. It is typically safe to assume one minute without the alert being too noisy. This can vary depending on how spiky your write traffic is and your OpenSearch Compute Unit (OCU) settings on the pipeline.

If your replication delay goes over 24 hours, your stream will start to drop events, and you'll have accuracy issues unless you do a full rebuild of your index from scratch.

Scaling

- Use auto scaling for pipelines to help scale up or down the OCUs to best fit the workload.

- For provisioned throughput tables without auto scaling, we recommend setting OCUs based on your write capacity units (WCUs) divided by 1000. Set the minimum to 1 OCU below that amount (but at least 1), and set the maximum to at least 1 OCU above that amount.

- Formula:**

```
OCU_minimum = GREATEST((table_WCU / 1000) - 1, 1)  
OCU_maximum = (table_WCU / 1000) + 1
```

- Example:** Your table has 25000 WCUs provisioned. Your pipeline's OCUs should be set with a minimum of 24 ($25000/1000 - 1$) and maximum of at least 26 ($25000/1000 + 1$).
- For provisioned throughput tables with auto scaling, we recommend setting OCUs based on your minimum and maximum WCUs, divided by 1000. Set the minimum to 1 OCU below the minimum from DynamoDB, and set the maximum to at least 1 OCU above the maximum from DynamoDB.

- Formula:**

```
OCU_minimum = GREATEST((table_minimum_WCU / 1000) - 1, 1)  
OCU_maximum = (table_maximum_WCU / 1000) + 1
```

- Example:** Your table has an auto scaling policy with a minimum of 8000 and maximum of 14000. Your pipeline's OCUs should be set with a minimum of 7 ($8000/1000 - 1$) and a maximum of 15 ($14000/1000 + 1$).
- For on-demand throughput tables, we recommend setting OCUs based on your typical peak and valley for write request units per second. You might need to average over a longer time period, depending on the aggregation that's available to you. Set the minimum to 1 OCU below the minimum from DynamoDB, and set the maximum to at least 1 OCU above the maximum from DynamoDB.

- Formula:**

```
# Assuming we have writes aggregated at the minute level  
OCU_minimum = GREATEST((min(table_writes_1min) / (60 * 1000)) - 1, 1)  
OCU_maximum = (max(table_writes_1min) / (60 * 1000)) + 1
```

- Example:** Your table has an average valley of 300 write request units per second and an average peak of 4300. Your pipeline's OCUs should be set with a minimum of 1 ($300/1000 - 1$, but at least 1) and a maximum of 5 ($4300/1000 + 1$).
- Follow best practices on scaling your destination OpenSearch Service indexes. If your indexes are under-scaled, it will slow down ingestion from DynamoDB, and might cause delays.

Note

GREATEST is a SQL function that, given a set of arguments, returns the argument with the greatest value.

Service, account, and table quotas in Amazon DynamoDB

This section describes current quotas, formerly referred to as limits, within Amazon DynamoDB. Each quota applies on a per-Region basis unless otherwise specified.

Topics

- [Read/write capacity mode and throughput](#)
- [Reserved Capacity](#)
- [Import quotas](#)
- [Contributor Insights](#)
- [Tables](#)
- [Global tables](#)
- [Secondary indexes](#)
- [Partition keys and sort keys](#)
- [Naming rules](#)
- [Data types](#)
- [Items](#)
- [Attributes](#)
- [Expression parameters](#)
- [DynamoDB transactions](#)
- [DynamoDB Streams](#)
- [DynamoDB Accelerator \(DAX\)](#)
- [API-specific limits](#)
- [DynamoDB encryption at rest](#)
- [Table export to Amazon S3](#)
- [Backup and restore](#)

Read/write capacity mode and throughput

You can switch between read/write capacity modes once every 24 hours. The only exception to this is if you switch a provisioned mode table to on-demand mode: you can switch back to provisioned mode in the same 24-hour period.

Capacity unit sizes (for provisioned tables)

One read capacity unit = one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.

One write capacity unit = one write per second, for items up to 1 KB in size.

Transactional read requests require two read capacity units to perform one read per second for items up to 4 KB.

Transactional write requests require two write capacity units to perform one write per second for items up to 1 KB.

Request unit sizes (for on-demand tables)

One read request unit = one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.

One write request unit = one write per second, for items up to 1 KB in size.

Transactional read requests require two read request units to perform one read per second for items up to 4 KB.

Transactional write requests require two write request units to perform one write per second for items up to 1 KB.

Throughput default quotas

AWS places some default quotas on the throughput that your account can provision and consume within a Region.

The account-level read throughput and account-level write throughput quotas apply at the account level. These account-level quotas apply to the sum of the provisioned throughput capacity for all your account's tables and global secondary indexes in a given Region. All the account's available throughput can be provisioned for a single table or across multiple tables. These quotas only apply to tables using the provisioned capacity mode.

The table-level read throughput and table-level write throughput quotas apply differently to tables that use the provisioned capacity mode, and tables that use the on-demand capacity mode.

For provisioned capacity mode tables and GSIs, the quota is the maximum amount of read and write capacity units that can be provisioned for any table or any of its GSIs in the Region. The total

of any individual table and all its GSIs must also remain below the account-level read and write throughput quota. This is in addition to the requirement that the total of all provisioned tables and their GSIs must remain below the account-level read and write throughput quota.

For on-demand capacity mode tables and GSIs, the table-level quota is the maximum read and write capacity units that are available for any table, or any individual GSI within that table. No account-level read and write throughput quotas are applied to tables in on-demand mode.

Below are the throughput quotas that apply on your account, by default.

	On-Demand	Provisioned	Adjustable
Per table	40,000 read request units and 40,000 write request units	40,000 read capacity units and 40,000 write capacity units	Yes
Per account	Not applicable	80,000 read capacity units and 80,000 write capacity units	Yes
Minimum throughput for any table or global secondary index	Not applicable	1 read capacity unit and 1 write capacity unit	Yes

You can use the [Service Quotas console](#), the [AWS API](#) and the [AWS CLI](#) to request quota increases for the adjustable quotas when needed.

For your account-level throughput quotas, you can use the [Service Quotas console](#), the [AWS CloudWatch console](#), the [AWS API](#) and the [AWS CLI](#) to create CloudWatch alarms and be notified automatically when your current usage reaches a specified percentage of your applied quota values. Using CloudWatch you can also monitor your usage by looking at the AccountProvisionedReadCapacityUnits and AccountProvisionedWriteCapacityUnits AWS usage metrics. To learn more about usage metrics, see [AWS usage metrics](#).

Increasing or decreasing throughput (for provisioned tables)

Increasing provisioned throughput

You can increase ReadCapacityUnits or WriteCapacityUnits as often as necessary, using the AWS Management Console or the UpdateTable operation. In a single call, you can increase the provisioned throughput for a table, for any global secondary indexes on that table, or for any combination of these. The new settings do not take effect until the UpdateTable operation is complete.

You can't exceed your per-account quotas when you add provisioned capacity, and DynamoDB doesn't allow you to increase provisioned capacity very rapidly. Aside from these restrictions, you can increase the provisioned capacity for your tables as high as you need. For more information about per-account quotas, see the preceding section, [Throughput default quotas](#).

Decreasing provisioned throughput

For every table and global secondary index in an UpdateTable operation, you can decrease ReadCapacityUnits or WriteCapacityUnits (or both). The new settings don't take effect until the UpdateTable operation is complete.

There is a default quota on the number of provisioned capacity decreases you can perform on your DynamoDB table per day. A day is defined according to Universal Time Coordinated (UTC). On a given day, you can start by performing up to four decreases within one hour as long as you have not performed any other decreases yet during that day. Subsequently, you can perform one additional decrease per hour (once every 60 minutes). This effectively brings the maximum number of decreases in a day to 27 times.

You can use the [Service Quotas console](#), the [AWS API](#) and the [AWS CLI](#) to request quota increases, when needed.

Important

Table and global secondary index decrease limits are decoupled, so any global secondary indexes for a particular table have their own decrease limits. However, if a single request decreases the throughput for a table and a global secondary index, it is rejected if either exceeds the current limits. Requests are not partially processed.

Example

In the first 4 hours of a day, a table with a global secondary index can be modified as follows:

- Decrease the table's WriteCapacityUnits or ReadCapacityUnits (or both) four times.
- Decrease the WriteCapacityUnits or ReadCapacityUnits (or both) of the global secondary index four times.

At the end of that same day, the table and the global secondary index throughput can potentially be decreased a total of 27 times each.

Reserved Capacity

AWS places a default quota on the amount of active reserved capacity that your account can purchase. The quota limit is a combination of reserved capacity for write capacity units (WCUs) and read capacity units (RCUs).

	Active reserved capacity	Adjustable
Per account	1,000,000 provisioned capacity units (WCUs _ RCUs)	Yes

If you attempt to purchase more than 1,000,000 provisioned capacity units in a single purchase, you will receive an error for this service quota limit. If you have active reserved capacity and attempt to purchase additional reserved capacity that would result in more than 1,000,000 active provisioned capacity units, you will receive an error for this service quota limit.

If you need reserved capacity for more than 1,000,000 provisioned capacity units, you can request a quota increase by submitting a request to the [support](#) team.

Import quotas

DynamoDB Import from Amazon S3 can support up to 50 concurrent import jobs with a total import source object size of 15TB at a time in us-east-1, us-west-2, and eu-west-1 regions. In all other regions, up to 50 concurrent import tasks with a total size of 1TB is supported. Each import job can take up to 50,000 Amazon S3 objects in all regions. For more information on import and validation, see [import format quotas and validation](#).

Contributor Insights

When you enable Customer Insights on your DynamoDB table, you're still subject to Contributor Insights rules limits. For more information, see [CloudWatch service quotas](#).

Tables

Table size

There is no practical limit on a table's size. Tables are unconstrained in terms of the number of items or the number of bytes.

Maximum number of tables per account per region

For any AWS account, there is an initial quota of 2,500 tables per AWS Region.

If you need more than 2,500 tables for a single account, please reach out to your AWS account team to explore an increase up to a maximum of 10,000 tables. For more than 10,000, the recommended best practice is to setup multiple accounts, each of which can serve up to 10,000 tables.

You can use the [Service Quotas console](#), the [AWS API](#) and the [AWS CLI](#) to view the default and applied quota values for the maximum number of tables for in your account, and to request quota increases, when needed. You can also request quota increases by cutting a ticket to [AWS support](#)

Using the [Service Quotas console](#), the [AWS API](#) and the [AWS CLI](#) you can create CloudWatch alarms to get notified automatically when your current usage reaches a specified percentage of your current quota. Using CloudWatch you can also monitor your usage by looking at the TableCount AWS usage metrics. To learn more about usage metrics, see [AWS usage metrics](#).

Global tables

AWS places some default quotas on the throughput you can provision or utilize when using global tables.

	On-Demand	Provisioned
Per table	40,000 read request units and 40,000 write request units	40,000 read capacity units and 40,000 write capacity units
Per table, per destination Region, per day	10 TB for all source tables to which a replica was added for this destination Region	10 TB for all source tables to which a replica was added for this destination Region

Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the AWS Region where the write is made originally. Transactions are not supported across Regions in global tables. For example, suppose that you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and you perform a `TransactWriteItems` operation in the US East (N. Virginia) Region. In this case, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

Note

There may be instances where you will need to request a quota limit increase through AWS Support. If any of the following apply to you, please see <https://aws.amazon.com/support>:

- If you are adding a replica for a table that is configured to use more than 40,000 write capacity units (WCU), you must request a service quota increase for your add replica WCU quota.
- If you are adding a replica or replicas to one destination Region within a 24-hour period with a combined total greater than 10TB, you must request a service quota increase for your add replica data backfill quota.
- If you encounter an error similar to the following:
 - Cannot create a replica of table 'example_table' in region 'example_region_A' because its exceeds your current account limit in region 'example_region_B'.

Secondary indexes

Secondary indexes per table

You can define a maximum of 5 local secondary indexes.

There is a default quota of 20 global secondary indexes per table. You can use the [Service Quotas console](#), the [AWS API](#) and the [AWS CLI](#) to check the global secondary indexes per table default and current quotas that apply for your account, and to request quota increases, when needed. You can also request quota increases by cutting a ticket to <https://aws.amazon.com/support>.

You can create or delete only one global secondary index per `UpdateTable` operation.

Projected Secondary Index attributes per table

You can project a total of up to 100 attributes into all of a table's local and global secondary indexes. This only applies to user-specified projected attributes.

In a `CreateTable` operation, if you specify a `ProjectionType` of `INCLUDE`, the total count of attributes specified in `NonKeyAttributes`, summed across all of the secondary indexes, must not exceed 100. If you project the same attribute name into two different indexes, this counts as two distinct attributes when determining the total.

This limit does not apply for secondary indexes with a `ProjectionType` of `KEYS_ONLY` or `ALL`.

Partition keys and sort keys

Partition key length

The minimum length of a partition key value is 1 byte. The maximum length is 2048 bytes.

Partition key values

There is no practical limit on the number of distinct partition key values, for tables or for secondary indexes.

Sort key length

The minimum length of a sort key value is 1 byte. The maximum length is 1024 bytes.

Sort key values

In general, there is no practical limit on the number of distinct sort key values per partition key value.

The exception is for tables with secondary indexes. An item collection is the set of items which have the same value of partition key attribute. In a global secondary index the item collection is independent of the base table (and can have a different partition key attribute), but in a local secondary index the indexed view is colocated in the same partition as the item in the table and shares the same partition key attribute. As a result of this locality, when a table has one or more LSIs, the item collection cannot be distributed to multiple partitions.

For a table with one or more LSIs, item collections cannot exceed 10GB in size. This includes all base table items and all projected LSI views which have the same value of the partition key attribute. 10GB is the maximum size of a partition. For more detailed information, see [Item collection size limit](#).

Naming rules

Table names and Secondary Index names

Names for tables and secondary indexes must be at least 3 characters long, but no greater than 255 characters long. The following are the allowed characters:

- A-Z
- a-z
- 0-9
- _ (underscore)
- - (hyphen)
- . (dot)

Attribute names

In general, an attribute name must be at least one character long, but no greater than 64 KB long.

The following are the exceptions. These attribute names must be no greater than 255 characters long:

- Secondary index partition key names.
 - Secondary index sort key names.
 - The names of any user-specified projected attributes (applicable only to local secondary indexes). In a `CreateTable` operation, if you specify a `ProjectionType` of `INCLUDE`, the names of the attributes in the `NonKeyAttributes` parameter are length-restricted. The `KEYS_ONLY` and `ALL` projection types are not affected.

These attribute names must be encoded using UTF-8, and the total size of each name (after encoding) cannot exceed 255 bytes.

Data types

String

The length of a String is constrained by the maximum item size of 400 KB.

Strings are Unicode with UTF-8 binary encoding. Because UTF-8 is a variable width encoding, DynamoDB determines the length of a String using its UTF-8 bytes.

Number

A Number can have up to 38 digits of precision, and can be positive, negative, or zero.

DynamoDB uses JSON strings to represent Number data in requests and replies. For more information, see [DynamoDB low-level API](#).

If number precision is important, you should pass numbers to DynamoDB using strings that you convert from a number type.

Binary

The length of a Binary is constrained by the maximum item size of 400 KB.

Applications that work with Binary attributes must encode the data in base64 format before sending it to DynamoDB. Upon receipt of the data, DynamoDB decodes it into an unsigned byte array and uses that as the length of the attribute.

Items

Item size

The maximum item size in DynamoDB is 400 KB, which includes both attribute name binary length (UTF-8 length) and attribute value lengths (again binary length). The attribute name counts towards the size limit.

For example, consider an item with two attributes: one attribute named "shirt-color" with value "R" and another attribute named "shirt-size" with value "M". The total size of that item is 23 bytes.

Item size for tables with Local Secondary Indexes

For each local secondary index on a table, there is a 400 KB limit on the total of the following:

- The size of an item's data in the table.
- The size of corresponding entries (including key values and projected attributes) in all local secondary indexes.

Attributes

Attribute name-value pairs per item

The cumulative size of attributes per item must fit within the maximum DynamoDB item size (400 KB).

Number of values in list, map, or set

There is no limit on the number of values in a List, a Map, or a Set, as long as the item containing the values fits within the 400 KB item size limit.

Attribute values

Empty String and Binary attribute values are allowed, if the attribute is not used as a key attribute for a table or index. Empty String and Binary values are allowed inside Set, List, and Map types. An

attribute value cannot be an empty Set (String Set, Number Set, or Binary Set). However, empty Lists and Maps are allowed.

Nested attribute depth

DynamoDB supports nested attributes up to 32 levels deep.

Expression parameters

Expression parameters include `ProjectionExpression`, `ConditionExpression`, `UpdateExpression`, and `FilterExpression`.

Lengths

The maximum length of any expression string is 4 KB. For example, the size of the `ConditionExpression` `a=b` is 3 bytes.

The maximum length of any single expression attribute name or expression attribute value is 255 bytes. For example, `#name` is 5 bytes; `:val` is 4 bytes.

The maximum length of all substitution variables in an expression is 2 MB. This is the sum of the lengths of all `ExpressionAttributeNames` and `ExpressionAttributeValues`.

Operators and operands

The maximum number of operators or functions allowed in an `UpdateExpression` is 300. For example, the `UpdateExpression` `SET a = :val1 + :val2 + :val3` contains two "+" operators.

The maximum number of operands for the IN comparator is 100.

Reserved words

DynamoDB does not prevent you from using names that conflict with reserved words. (For a complete list, see [Reserved words in DynamoDB](#).)

However, if you use a reserved word in an expression parameter, you must also specify `ExpressionAttributeNames`. For more information, see [Expression attribute names in DynamoDB](#).

DynamoDB transactions

DynamoDB transactional API operations have the following constraints:

- A transaction cannot contain more than 100 unique items.
- A transaction cannot contain more than 4 MB of data.
- No two actions in a transaction can work against the same item in the same table. For example, you cannot both ConditionCheck and Update the same item in one transaction.
- A transaction cannot operate on tables in more than one AWS account or Region.
- Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the AWS Region where the write is made originally. Transactions are not supported across Regions in global tables. For example, suppose that you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and you perform a TransactWriteItems operation in the US East (N. Virginia) Region. In this case, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

DynamoDB Streams

Simultaneous readers of a shard in DynamoDB Streams

For single-Region tables that are not global tables, you can design for up to two processes to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling. For global tables, we recommend you limit the number of simultaneous readers to one to avoid request throttling.

Maximum write capacity for a table with DynamoDB Streams enabled

AWS places some default quotas on the write capacity for DynamoDB tables with DynamoDB Streams enabled. These default quotas are applicable only for tables in provisioned read/write capacity mode. The following are throughput quotas that apply to your account by default.

- US East (N. Virginia), US East (Ohio), US West (N. California), US West (Oregon), South America (São Paulo), Europe (Frankfurt), Europe (Ireland), Asia Pacific (Tokyo), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), China (Beijing) Regions:

- Per table – 40,000 write capacity units
- All other Regions:
 - Per table – 10,000 write capacity units

You can use the [Service Quotas console](#), the [AWS API](#) and the [AWS CLI](#) to check the maximum write capacity for a table with DynamoDB Streams enabled default and current quotas that apply on your account, and to request quota increases, when needed. You can also request quota increases by cutting a ticket to [AWS support](#).

Note

The provisioned throughput quotas also apply for DynamoDB tables with DynamoDB Streams enabled. When you request a quota increase on the write capacity for a table with Streams enabled, make sure you also request an increase of the provisioned throughput capacity for this table. For more information, see [Throughput Default Quotas](#). Other quotas also apply when processing higher throughput DynamoDB Streams. For more information, see [Amazon DynamoDB Streams API reference guide](#).

DynamoDB Accelerator (DAX)

AWS region availability

For a list of AWS Regions in which DAX is available, see [DynamoDB Accelerator \(DAX\)](#) in the [AWS General Reference](#).

Nodes

A DAX cluster consists of exactly one primary node, and between zero and ten read replica nodes.

The total number of nodes (per AWS account) cannot exceed 50 in a single AWS Region.

Parameter groups

You can create up to 20 DAX parameter groups per Region.

Subnet groups

You can create up to 50 DAX subnet groups per Region.

Within a subnet group, you can define up to 20 subnets.

API-specific limits

CreateTable/UpdateTable/DeleteTable/PutResourcePolicy/DeleteResourcePolicy

In general, you can have up to 500 [CreateTable](#), [UpdateTable](#), [DeleteTable](#), [PutResourcePolicy](#), and [DeleteResourcePolicy](#) requests running simultaneously in any combination. As a result, the total number of tables in the CREATING, UPDATING, or DELETING state cannot exceed 500.

You can submit up to 2,500 requests per second of mutable (CreateTable, DeleteTable, UpdateTable, PutResourcePolicy, and DeleteResourcePolicy) control plane API requests across a group of tables. However, the PutResourcePolicy and DeleteResourcePolicy requests have lower individual limits. For more information, see the following quotas details for PutResourcePolicy and DeleteResourcePolicy.

CreateTable and PutResourcePolicy requests which include a resource-based policy will count as two additional requests for each KB of the policy. For example, a CreateTable or PutResourcePolicy request with a policy of size 5 KB will count as 11 requests. 1 for the CreateTable request and 10 for the resource-based policy (2×5 KB). Similarly, a policy of size 20 KB will count as 41 requests. 1 for the CreateTable request and 40 for the resource-based policy (2×20 KB).

PutResourcePolicy

You can submit up to 25 PutResourcePolicy API requests per second across a group of tables. After a successful request for an individual table, no new PutResourcePolicy requests are supported for the following 15 seconds.

The maximum size supported for a resource-based policy document is 20 KB. DynamoDB counts whitespaces when calculating the size of a policy against this limit.

DeleteResourcePolicy

You can submit up to 50 DeleteResourcePolicy API requests per second across a group of tables. After a successful PutResourcePolicy request for an individual table, no DeleteResourcePolicy requests are supported for the following 15 seconds.

BatchGetItem

A single BatchGetItem operation can retrieve a maximum of 100 items. The total size of all the items retrieved cannot exceed 16 MB.

BatchWriteItem

A single BatchWriteItem operation can contain up to 25 PutItem or DeleteItem requests. The total size of all the items written cannot exceed 16 MB.

DescribeStream

You can call DescribeStream at a maximum rate of 10 times per second.

DescribeTableReplicaAutoScaling

DescribeTableReplicaAutoScaling method supports only 10 requests per second.

DescribeLimits

DescribeLimits should be called only periodically. You can expect throttling errors if you call it more than once in a minute.

DescribeContributorInsights/ListContributorInsights/UpdateContributorInsights

DescribeContributorInsights, ListContributorInsights and UpdateContributorInsights should be called only periodically. DynamoDB supports up to five requests per second for each of these APIs.

DescribeTable>ListTables/GetResourcePolicy

You can submit up to 2,500 requests per second of a combination of read-only (DescribeTable, ListTables, and GetResourcePolicy) control plane API requests. The GetResourcePolicy API has a lower individual limit of 100 requests per second.

Query

The result set from a Query is limited to 1 MB per call. You can use the `LastEvaluatedKey` from the query response to retrieve more results.

Scan

The result set from a Scan is limited to 1 MB per call. You can use the `LastEvaluatedKey` from the scan response to retrieve more results.

UpdateKinesisStreamingDestination

When performing `UpdateKinesisStreamingDestination` operations, you can set `ApproximateCreationDateTimePrecision` to a new value a maximum of 3 times in a 24 hour period.

UpdateTableReplicaAutoScaling

`UpdateTableReplicaAutoScaling` method supports only ten requests per second.

UpdateTableTimeToLive

The `UpdateTableTimeToLive` method supports only one request to enable or disable Time to Live (TTL) per specified table per hour. This change can take up to one hour to fully process. Any additional `UpdateTimeToLive` calls for the same table during this one hour duration result in a `ValidationException`.

DynamoDB encryption at rest

You can switch between an AWS owned key, an AWS managed key, and a customer managed key up to four times, anytime per 24-hour window, on a per table basis, starting from when the table was created. If there was no change in the past six hours, an additional change is allowed. This effectively brings the maximum number of changes in a day to eight (four changes in the first six hours, and one change for each of the subsequent six hour windows in a day).

You can switch encryption keys to use an AWS owned key as often as necessary, even if the above quota has been exhausted.

These are the quotas unless you request a higher amount. To request a service quota increase, see <https://aws.amazon.com/support>.

Table export to Amazon S3

Full export: up to 300 concurrent export tasks, or up to a total of 100TB from all in-flight table exports, can be exported. Both of these limits are checked before an export is queued.

Incremental export: up to 300 concurrent jobs, or 100TB of table size, in an export period window between 15 minutes minimum and 24 hours maximum, can be exported concurrently.

Backup and restore

When restoring through DynamoDB on-demand backup, you can execute up to 50 concurrent restores that total 50TB. When restoring through AWS Backup, you can execute up to 50 concurrent restores that total 25TB. For more information about backups, see [Using On-Demand backup and restore for DynamoDB](#).

Low-level API reference

The [Amazon DynamoDB API reference](#) contains a complete list of operations supported by:

- [DynamoDB](#).
- [DynamoDB Streams](#).
- [DynamoDB Accelerator \(DAX\)](#).

Troubleshooting Amazon DynamoDB

The following topics provide troubleshooting advice for errors and issues that you might encounter when using Amazon DynamoDB. If you find an issue that is not listed here, you can use the Feedback button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the [AWS Knowledge Center](#).

Topics

- [Troubleshooting latency issues in Amazon DynamoDB](#)
- [Troubleshooting throttling issues in Amazon DynamoDB](#)

Troubleshooting latency issues in Amazon DynamoDB

If your workload appears to experience high latency, you can analyze the CloudWatch SuccessfulRequestLatency metric, and check the average latency to see if it's related to DynamoDB. Some variability in the reported SuccessfulRequestLatency is normal, and occasional spikes (particularly in the Maximum statistic) should not be cause for concern. However, if the Average statistic shows a sharp increase and persists, you should check the AWS Service Health Dashboard and your Personal Health Dashboard for more information. Some possible causes include the size of the item in your table (a 1kb item and a 400kb item will vary in latency) or the size of the query (10 items versus 100 items).

If necessary, consider opening a support case with AWS Support, and continue to assess any available fall-back options for your application (such as evacuation of a Region if you have a multi-Region architecture) according to your runbooks. You should log request IDs for slow requests for providing these IDs to AWS Support when you open a support case.

The SuccessfulRequestLatency metric only measures latency which is internal to the DynamoDB service - client side activity and network trip times are not included. To learn more about overall latency for calls from your client to the DynamoDB service, you can enable latency metric logging in your AWS SDK.

Note

For most singleton operations (operations which apply to a single item by fully specifying the primary key's value), DynamoDB delivers single-digit millisecond Average SuccessfulRequestLatency. This value does not include the transport overhead for the caller code accessing the DynamoDB endpoint. For multi-item data operations, the latency will vary based on factors such as size of the result set, the complexity of the data structures returned, and any condition expressions and filter expressions applied. For repeated multi-item operations to the same data set with the same parameters, DynamoDB will provide highly consistent Average SuccessfulRequestLatency.

Consider one or more of the following strategies to reduce latency:

- **Adjust request timeout and retry behavior:** The path from your client to DynamoDB traverses many components, each of which is designed with redundancy in mind. Think about the scope of network resiliency, TCP packet timeouts, and the distributed architecture of DynamoDB itself. The default SDK behaviors are designed to find the right balance for most applications. If the best possible latency is your highest priority, you should consider adjusting the default request timeout and retry settings for your SDK to closely track to the typical latency for a successful request measured by your client. A request which is taking significantly longer than normal is less likely to ultimately succeed - if you fail fast and make a new request, this is likely to take a different path and may quickly succeed. Keep in mind that there can be downsides to being too aggressive in these settings. A helpful discussion on this topic can be found in [Tuning AWS Java SDK HTTP request settings for latency-aware Amazon DynamoDB applications](#).
- **Reduce the distance between the client and DynamoDB endpoint:** If you have globally dispersed users, consider using [Global tables - multi-Region replication for DynamoDB](#). With global tables, you can specify the AWS Regions where you want the table to be available. Reading data from a local global tables replica can significantly reduce latency for your users. Also, consider using a DynamoDB [gateway endpoint](#) to keep your client traffic within your VPC.
- **Use caching:** If your traffic is read heavy, consider using a caching service, such as [In-memory acceleration with DynamoDB Accelerator \(DAX\)](#). DAX is a fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement, from milliseconds to microseconds, even at millions of requests per second.
- **Reuse connections:** DynamoDB requests are made via an authenticated session which defaults to HTTPS. Initiating the connection takes time so the latency of the first request is higher than

typical. Requests over an already initialized connection deliver DynamoDB's consistent low latency. For this reason, you may wish to make a "keep-alive" GetItem request every 30 seconds if no other requests are made, to avoid the latency of establishing a new connection.

- **Use eventually consistent reads:** If your application doesn't require strongly consistent reads, consider using the default eventually consistent reads. Eventually consistent reads are lower cost and are also less likely to experience transient increases in latency. For more information, see [Read consistency](#).

Troubleshooting throttling issues in Amazon DynamoDB

In service-oriented architectures and distributed systems, limiting the rate at which API calls are processed by various service components is called throttling. This helps to smooth spikes, control for mismatches in component throughput, and allows for more predictable recoveries when there's an unexpected operational event. DynamoDB is designed for these types of architectures, and DynamoDB clients have retries built in for throttled requests. Some degree of throttling is not necessarily a problem for your application, but persistent throttling of a latency-sensitive part of your data workflow can negatively impact user experience and reduce the overall efficiency of the system.

If your read or write operations on your DynamoDB table are being throttled (or if you are getting ProvisionedThroughputExceededException when you perform an operation), consider using the following strategies to help resolve the issue.

Topics

- [Make sure provisioned mode tables have adequate capacity](#)
- [Consider switching to on-demand mode](#)
- [Distribute read and write operations evenly between partition keys](#)
- [Increase your table-level read or write throughput quota](#)

Make sure provisioned mode tables have adequate capacity

DynamoDB reports minute-level metrics to CloudWatch. The metrics are calculated as the sum for a minute and then averaged. However, the DynamoDB rate limits are applied per second. For example, if you provisioned 60 write capacity units for your table, you can perform 3600 write units in one minute, but trying to perform more than 60 write units in any one second might result in some requests being throttled.

To resolve this issue, make sure that your table has enough capacity to serve your traffic and retry throttled requests using [exponential backoff](#). If you are using an AWS SDK, then this logic is implemented by default. For this reason, throttles might occur, and be retried successfully on the second or third retry, while never causing errors in your code. In order to understand throttles as they happen, each AWS-SDK flavor has a way to detect throttles. For example, in Python check the number in the returned `ResponseMetadata.RetryAttempts` field. If this is greater than zero, it indicates there were throttles. A great strategy upon detecting this is to log the Partition Key involved, as it may be a hot key pattern that would be hard to diagnose with CloudWatch alone.

Also, consider using [auto scaling](#) on your provisioned tables to automatically adjust provisioned capacity based on workload. You should also consider updating the scaling policy to have a higher minimum or a lower target utilization, and make sure the maximum setting isn't set too low.

Consider switching to on-demand mode

Amazon DynamoDB auto scaling uses the [Application Auto Scaling](#) service to dynamically adjust provisioned throughput capacity in response to actual traffic patterns. Application Auto Scaling initiates a scale up only when two consecutive data points for consumed capacity units exceed the configured target utilization value within a one-minute span. Application Auto Scaling automatically scales the provisioned capacity only when the consumed capacity is higher than target utilization for two consecutive minutes. Also, a scale-down event is initiated when 15 consecutive data points for consumed capacity in CloudWatch are lower than the target utilization.

After Application Auto Scaling is initiated, an `UpdateTable` API call is invoked to update the provisioned capacity for your DynamoDB table or index. Application Auto Scaling requires consecutive data points with higher target utilization values to scale up the provisioned capacity of the DynamoDB table. During this period, any requests that exceed the provisioned capacity of the table are throttled. Therefore, consider using [On-demand mode](#) for extremely unpredictable (bursty) traffic patterns. For more information, see [Managing throughput capacity automatically with DynamoDB auto scaling](#).

Distribute read and write operations evenly between partition keys

In DynamoDB, a partition key with low cardinality can result in many requests targeting only a few partitions and resulting in a hot partition. A hot partition can cause throttling if the partition limits of 3000 RCU and 1000 WCU per second are exceeded.

To find the most accessed and throttled items in your table, use [Amazon CloudWatch Contributor Insights](#). Amazon CloudWatch Contributor Insights is a diagnostic tool that provides a summarized

view of your DynamoDB tables traffic trends and helps you identify the most frequently accessed partition keys. With this tool, you can continuously monitor the graphs for your table's item access patterns. A hot partition can degrade the overall performance of your table. To avoid this poor performance, distribute the read and write operations as evenly as possible by choosing a high cardinality key. For more information, see [Designing partition keys to distribute your workload](#) and [Choosing the right DynamoDB partition key](#).

 **Note**

Using the Amazon CloudWatch Contributor Insights tool for DynamoDB incurs additional charges. For more information, see [CloudWatch contributor insights for DynamoDB billing](#).

Increase your table-level read or write throughput quota

The table-level [read throughput and table-level write throughput quotas](#) apply at the account level in any Region. These quotas apply for tables with both provisioned capacity mode and on-demand capacity mode. By default, the throughput quota placed on your table is 40,000 read requests units and 40,000 write requests units. If the traffic to your table exceeds this quota, then the table might be throttled. For more information on how to prevent this from happening, see [Monitoring Amazon DynamoDB for operational awareness](#).

To resolve this issue, use the [Service Quotas console](#) to increase the table-level read or write throughput quota for your account.

DynamoDB Appendix

Topics

- [Troubleshooting SSL/TLS connection establishment issues](#)
- [Example tables and data](#)
- [Creating example tables and uploading data](#)
- [DynamoDB example application using the AWS SDK for Python \(Boto\): Tic-tac-toe](#)
- [Exporting and importing DynamoDB data using AWS Data Pipeline](#)
- [Amazon DynamoDB Storage Backend for Titan](#)
- [Reserved words in DynamoDB](#)
- [Legacy conditional parameters](#)
- [Previous low-level API version \(2011-12-05\)](#)
- [AWS SDK for Java 1.x examples](#)

Troubleshooting SSL/TLS connection establishment issues

Amazon DynamoDB is in the process of moving our endpoints to secure certificates signed by the Amazon Trust Services (ATS) Certificate Authority instead of third-party Certificate Authority. In December 2017, we launched the EU-WEST-3 (Paris) Region with the secure certificates issued by the Amazon Trust Services. All new regions launched after December 2017 have endpoints with the certificates issued by the Amazon Trust Services. This guide shows you how to validate and troubleshoot SSL/TLS connection issues.

Testing your application or service

Most AWS SDKs and Command Line Interfaces (CLIs) support the Amazon Trust Services Certificate Authority. If you are using a version of the AWS SDK for Python or CLI released before October 29, 2013, you must upgrade. The .NET, Java, PHP, Go, JavaScript, and C++ SDKs and CLIs do not bundle any certificates, their certificates come from the underlying operating system. The Ruby SDK has included at least one of the required CAs since June 10, 2015. Before that date, the Ruby V2 SDK did not bundle certificates. If you use an unsupported, custom, or modified version of the AWS SDK, or if you use custom trust store, you might not have the support needed for Amazon Trust Services Certificate Authority.

To validate access to DynamoDB endpoints, you will need to develop a test that accesses DynamoDB API or DynamoDB Streams API in the EU-WEST-3 region and validate that the TLS handshake succeeds. The specific endpoints you will need to access in such test are:

- DynamoDB: <https://dynamodb.eu-west-3.amazonaws.com>
- DynamoDB Streams: <https://streams.dynamodb.eu-west-3.amazonaws.com>

If your application does not support Amazon Trust Services Certificate Authority you will see one of the following failures:

- SSL/TLS Negotiation errors
- A long delay before your software receives an error indicating SSL/TLS negotiation failure. The delay time depends on the retry strategy and timeout configuration of your client.

Testing your client browser

To verify that your browser can connect to Amazon DynamoDB, open the following URL: <https://dynamodb.eu-west-3.amazonaws.com>. If the test is successful, you will see a message like this:

healthy: dynamodb.eu-west-3.amazonaws.com

If the test is unsuccessful, it will display an error similar to this: <https://untrusted-root.badssl.com/>.

Updating your software application client

Applications accessing DynamoDB or DynamoDB Streams API endpoints (whether through browsers or programmatically) will need to update the trusted CA list on the client machines if they do not already support any of the following CAs:

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

If the clients already trust ANY of the above three CAs then these will trust certificates used by DynamoDB and no action is required. However, if your clients do not already trust any of the above CAs, the HTTPS connections to the DynamoDB or DynamoDB Streams APIs will fail. For more

information, please visit this blog post: <https://aws.amazon.com/blogs/security/how-to-prepare-for-aws-move-to-its-own-certificate-authority/>.

Updating your client browser

You can update the certificate bundle in your browser simply by updating your browser. Instructions for the most common browsers can be found on the browsers' websites:

- Chrome: <https://support.google.com/chrome/answer/95414?hl=en>
- Firefox: <https://support.mozilla.org/en-US/kb/update-firefox-latest-version>
- Safari: <https://support.apple.com/en-us/HT204416>
- Internet Explorer: <https://support.microsoft.com/en-us/help/17295/windows-internet-explorer-which-version#ie=other>

Manually updating your certificate bundle

If you cannot access the DynamoDB API or DynamoDB Streams API then you need to update your certificate bundle. To do this, you need to import at least one of the required CAs. You can find these at <https://www.amazontrust.com/repository/>.

The following operating systems and programming languages support Amazon Trust Services certificates:

- Microsoft Windows versions that have January 2005 or later updates installed, Windows Vista, Windows 7, Windows Server 2008, and newer versions.
- MacOS X 10.4 with Java for MacOS X 10.4 Release 5, MacOS X 10.5 and newer versions.
- Red Hat Enterprise Linux 5 (March 2007), Linux 6, and Linux 7 and CentOS 5, CentOS 6, and CentOS 7
- Ubuntu 8.10
- Debian 5.0
- Amazon Linux (all versions)
- Java 1.4.2_12, Java 5 update 2, and all newer versions, including Java 6, Java 7, and Java 8

If you are still unable to connect please consult your software documentation, OS Vendor or contact AWS Support <https://aws.amazon.com/support> for further assistance.

Example tables and data

The *Amazon DynamoDB Developer Guide* uses sample tables to illustrate various aspects of DynamoDB.

Table name	Primary key
<i>ProductCatalog</i>	Simple primary key: <ul style="list-style-type: none">• Id (Number)
<i>Forum</i>	Simple primary key: <ul style="list-style-type: none">• Name (String)
<i>Thread</i>	Composite primary key: <ul style="list-style-type: none">• ForumName (String)• Subject (String)
<i>Reply</i>	Composite primary key: <ul style="list-style-type: none">• Id (String)• ReplyDateTime (String)

The *Reply* table has a global secondary index named *PostedBy-Message-Index*. This index will facilitate queries on two non-key attributes of the *Reply* table.

Index name	Primary key
<i>PostedBy-Message-Index</i>	Composite primary key: <ul style="list-style-type: none">• PostedBy (String)• Message (String)

For more information about these tables, see [Step 1: Create a table](#) and [Step 2: Write data to a table using the console or AWS CLI](#).

Sample data files

Topics

- [ProductCatalog sample data](#)
- [Forum sample data](#)
- [Thread sample data](#)
- [Reply sample data](#)

The following sections show the sample data files that are used for loading the *ProductCatalog*, *Forum*, *Thread* and *Reply* tables.

Each data file contains multiple PutRequest elements, each of which contain a single item. These PutRequest elements are used as input to the BatchWriteItem operation, using the AWS Command Line Interface (AWS CLI).

For more information, see [Step 2: Write data to a table using the console or AWS CLI](#) in [Creating tables and loading data for code examples in DynamoDB](#).

ProductCatalog sample data

```
{  
    "ProductCatalog": [  
        {  
            "PutRequest": {  
                "Item": {  
                    "Id": {  
                        "N": "101"  
                    },  
                    "Title": {  
                        "S": "Book 101 Title"  
                    },  
                    "ISBN": {  
                        "S": "111-1111111111"  
                    },  
                    "Authors": {  
                        "L": [  
                            {  
                                "S": "Author1"  
                            }  
                        ]  
                    }  
                }  
            }  
        ]  
    }  
}
```

```
        },
        "Price": {
            "N": "2"
        },
        "Dimensions": {
            "S": "8.5 x 11.0 x 0.5"
        },
        "PageCount": {
            "N": "500"
        },
        "InPublication": {
            "BOOL": true
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "102"
            },
            "Title": {
                "S": "Book 102 Title"
            },
            "ISBN": {
                "S": "222-2222222222"
            },
            "Authors": {
                "L": [
                    {
                        "S": "Author1"
                    },
                    {
                        "S": "Author2"
                    }
                ]
            },
            "Price": {
                "N": "20"
            }
        }
    }
}
```

```
        "Dimensions": {
            "S": "8.5 x 11.0 x 0.8"
        },
        "PageCount": {
            "N": "600"
        },
        "InPublication": {
            "BOOL": true
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "103"
            },
            "Title": {
                "S": "Book 103 Title"
            },
            "ISBN": {
                "S": "333-3333333333"
            },
            "Authors": {
                "L": [
                    {
                        "S": "Author1"
                    },
                    {
                        "S": "Author2"
                    }
                ]
            },
            "Price": {
                "N": "2000"
            },
            "Dimensions": {
                "S": "8.5 x 11.0 x 1.5"
            },
            "PageCount": {
```

```
        "N": "600"
    },
    "InPublication": {
        "BOOL": false
    },
    "ProductCategory": {
        "S": "Book"
    }
}
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "201"
            },
            "Title": {
                "S": "18-Bike-201"
            },
            "Description": {
                "S": "201 Description"
            },
            "BicycleType": {
                "S": "Road"
            },
            "Brand": {
                "S": "Mountain A"
            },
            "Price": {
                "N": "100"
            },
            "Color": {
                "L": [
                    {
                        "S": "Red"
                    },
                    {
                        "S": "Black"
                    }
                ]
            },
            "ProductCategory": {
                "S": "Bicycle"
            }
        }
    }
}
```

```
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "202"
            },
            "Title": {
                "S": "21-Bike-202"
            },
            "Description": {
                "S": "202 Description"
            },
            "BicycleType": {
                "S": "Road"
            },
            "Brand": {
                "S": "Brand-Company A"
            },
            "Price": {
                "N": "200"
            },
            "Color": {
                "L": [
                    {
                        "S": "Green"
                    },
                    {
                        "S": "Black"
                    }
                ]
            },
            "ProductCategory": {
                "S": "Bicycle"
            }
        }
    },
    {
        "PutRequest": {
            "Item": {
```

```
"Id": {  
    "N": "203"  
},  
"Title": {  
    "S": "19-Bike-203"  
},  
"Description": {  
    "S": "203 Description"  
},  
"BicycleType": {  
    "S": "Road"  
},  
"Brand": {  
    "S": "Brand-Company B"  
},  
"Price": {  
    "N": "300"  
},  
"Color": {  
    "L": [  
        {  
            "S": "Red"  
        },  
        {  
            "S": "Green"  
        },  
        {  
            "S": "Black"  
        }  
    ]  
},  
"ProductCategory": {  
    "S": "Bicycle"  
}  
}  
}  
},  
{  
    "PutRequest": {  
        "Item": {  
            "Id": {  
                "N": "204"  
            },  
            "Title": {  
                "S": "19-Bike-204"  
            }  
        }  
    }  
}
```

```
        "S": "18-Bike-204"
    },
    "Description": {
        "S": "204 Description"
    },
    "BicycleType": {
        "S": "Mountain"
    },
    "Brand": {
        "S": "Brand-Company B"
    },
    "Price": {
        "N": "400"
    },
    "Color": {
        "L": [
            {
                "S": "Red"
            }
        ]
    },
    "ProductCategory": {
        "S": "Bicycle"
    }
}
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "205"
            },
            "Title": {
                "S": "18-Bike-204"
            },
            "Description": {
                "S": "205 Description"
            },
            "BicycleType": {
                "S": "Hybrid"
            },
            "Brand": {
                "S": "Brand-Company C"
            }
        }
    }
}
```

```
        },
        "Price": {
            "N": "500"
        },
        "Color": {
            "L": [
                {
                    "S": "Red"
                },
                {
                    "S": "Black"
                }
            ]
        },
        "ProductCategory": {
            "S": "Bicycle"
        }
    }
}
]
```

Forum sample data

```
{
    "Forum": [
        {
            "PutRequest": {
                "Item": {
                    "Name": {"S": "Amazon DynamoDB"},
                    "Category": {"S": "Amazon Web Services"},
                    "Threads": {"N": "2"},
                    "Messages": {"N": "4"},
                    "Views": {"N": "1000"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Name": {"S": "Amazon S3"},
                    "Category": {"S": "Amazon Web Services"}
                }
            }
        }
    ]
}
```

```
        }
    }
}
]
```

Thread sample data

```
{
    "Thread": [
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon DynamoDB"
                    },
                    "Subject": {
                        "S": "DynamoDB Thread 1"
                    },
                    "Message": {
                        "S": "DynamoDB thread 1 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-22T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "0"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "index"
                            },
                            {

```

```
        "S": "primarykey"
    },
    {
        "S": "table"
    }
]
}
}
},
{
    "PutRequest": {
        "Item": {
            "ForumName": {
                "S": "Amazon DynamoDB"
            },
            "Subject": {
                "S": "DynamoDB Thread 2"
            },
            "Message": {
                "S": "DynamoDB thread 2 message"
            },
            "LastPostedBy": {
                "S": "User A"
            },
            "LastPostedDateTime": {
                "S": "2015-09-15T19:58:22.514Z"
            },
            "Views": {
                "N": "3"
            },
            "Replies": {
                "N": "0"
            },
            "Answered": {
                "N": "0"
            },
            "Tags": {
                "L": [
                    {
                        "S": "items"
                    },
                    {
                        "S": "attributes"
                    }
                ]
            }
        }
    }
}
```

```
        },
        {
            "S": "throughput"
        }
    ]
}
}
},
{
    "PutRequest": {
        "Item": {
            "ForumName": {
                "S": "Amazon S3"
            },
            "Subject": {
                "S": "S3 Thread 1"
            },
            "Message": {
                "S": "S3 thread 1 message"
            },
            "LastPostedBy": {
                "S": "User A"
            },
            "LastPostedDateTime": {
                "S": "2015-09-29T19:58:22.514Z"
            },
            "Views": {
                "N": "0"
            },
            "Replies": {
                "N": "0"
            },
            "Answered": {
                "N": "0"
            },
            "Tags": {
                "L": [
                    {
                        "S": "largeobjects"
                    },
                    {
                        "S": "multipart upload"
                    }
                ]
            }
        }
    }
}
```

```
        ]
    }
}
]
}
```

Reply sample data

```
{
  "Reply": [
    {
      "PutRequest": {
        "Item": {
          "Id": {
            "S": "Amazon DynamoDB#DynamoDB Thread 1"
          },
          "ReplyDateTime": {
            "S": "2015-09-15T19:58:22.947Z"
          },
          "Message": {
            "S": "DynamoDB Thread 1 Reply 1 text"
          },
          "PostedBy": {
            "S": "User A"
          }
        }
      }
    },
    {
      "PutRequest": {
        "Item": {
          "Id": {
            "S": "Amazon DynamoDB#DynamoDB Thread 1"
          },
          "ReplyDateTime": {
            "S": "2015-09-22T19:58:22.947Z"
          },
          "Message": {
            "S": "DynamoDB Thread 1 Reply 2 text"
          },
          "PostedBy": {

```

```
        "S": "User B"
    }
}
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "S": "Amazon DynamoDB#DynamoDB Thread 2"
            },
            "ReplyDateTime": {
                "S": "2015-09-29T19:58:22.947Z"
            },
            "Message": {
                "S": "DynamoDB Thread 2 Reply 1 text"
            },
            "PostedBy": {
                "S": "User A"
            }
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "S": "Amazon DynamoDB#DynamoDB Thread 2"
            },
            "ReplyDateTime": {
                "S": "2015-10-05T19:58:22.947Z"
            },
            "Message": {
                "S": "DynamoDB Thread 2 Reply 2 text"
            },
            "PostedBy": {
                "S": "User A"
            }
        }
    }
}
]
```

Creating example tables and uploading data

Topics

- [Creating example tables and uploading data using the AWS SDK for Java](#)
- [Creating example tables and uploading data using the AWS SDK for .NET](#)

In [Creating tables and loading data for code examples in DynamoDB](#), you first create tables using the DynamoDB console and then use the AWS CLI to add data to the tables. This appendix provides code to both create the tables and add data programmatically.

Creating example tables and uploading data using the AWS SDK for Java

The following Java code example creates tables and uploads data to the tables. The resulting table structure and data is shown in [Creating tables and loading data for code examples in DynamoDB](#). For step-by-step instructions to run this code using Eclipse, see [Java code examples](#).

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.HashSet;
import java.util.TimeZone;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.LocalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
```

```
public class CreateTablesLoadData {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    static String productCatalogTableName = "ProductCatalog";
    static String forumTableName = "Forum";
    static String threadTableName = "Thread";
    static String replyTableName = "Reply";

    public static void main(String[] args) throws Exception {

        try {

            deleteTable(productCatalogTableName);
            deleteTable(forumTableName);
            deleteTable(threadTableName);
            deleteTable(replyTableName);

            // Parameter1: table name
            // Parameter2: reads per second
            // Parameter3: writes per second
            // Parameter4/5: partition key and data type
            // Parameter6/7: sort key and data type (if applicable)

            createTable(productCatalogTableName, 10L, 5L, "Id", "N");
            createTable(forumTableName, 10L, 5L, "Name", "S");
            createTable(threadTableName, 10L, 5L, "ForumName", "S", "Subject", "S");
            createTable(replyTableName, 10L, 5L, "Id", "S", "ReplyDateTime", "S");

            loadSampleProducts(productCatalogTableName);
            loadSampleForums(forumTableName);
            loadSampleThreads(threadTableName);
            loadSampleReplies(replyTableName);

        } catch (Exception e) {
            System.err.println("Program failed:");
            System.err.println(e.getMessage());
        }
        System.out.println("Success.");
    }
}
```

```
}

private static void deleteTable(String tableName) {
    Table table = dynamoDB.getTable(tableName);
    try {
        System.out.println("Issuing DeleteTable request for " + tableName);
        table.delete();
        System.out.println("Waiting for " + tableName + " to be deleted...this may
take a while...");
        table.waitForDelete();
    } catch (Exception e) {
        System.err.println("DeleteTable request failed for " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
    String partitionKeyName, String partitionKeyType) {

    createTable(tableName, readCapacityUnits, writeCapacityUnits, partitionKeyName,
partitionKeyType, null, null);
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
    String partitionKeyName, String partitionKeyType, String sortKeyName,
String sortKeyType) {

    try {

        ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
        keySchema.add(new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH)); // //
Partition

        // key

        ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
        attributeDefinitions
            .add(new AttributeDefinition().withAttributeName(partitionKeyName)
                .withAttributeType(partitionKeyType));
    }
}
```

```
        if (sortKeyName != null) {
            keySchema.add(new
KeySchemaElement().withAttributeName(sortKeyName).withKeyType(KeyType.RANGE)); // Sort

                // key
            attributeDefinitions
                .add(new
AttributeDefinition().withAttributeName(sortKeyName).withAttributeType(sortKeyType));
        }

        CreateTableRequest request = new
CreateTableRequest().withTableName(tableName).withKeySchema(keySchema)
                .withProvisionedThroughput(new
ProvisionedThroughput().withReadCapacityUnits(readCapacityUnits)
                .withWriteCapacityUnits(writeCapacityUnits));

        // If this is the Reply table, define a local secondary index
        if (replyTableName.equals(tableName)) {

            attributeDefinitions
                .add(new
AttributeDefinition().withAttributeName("PostedBy").withAttributeType("S"));

            ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
ArrayList<LocalSecondaryIndex>();
            localSecondaryIndexes.add(new
LocalSecondaryIndex().withIndexName("PostedBy-Index")
                .withKeySchema(
                    new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH), // Partition

                    // key
                    new
KeySchemaElement().withAttributeName("PostedBy").withKeyType(KeyType.RANGE)) // Sort

                    // key
                    .withProjection(new
Projection().withProjectionType(ProjectionType.KEYS_ONLY)));

            request.setLocalSecondaryIndexes(localSecondaryIndexes);
        }
    }
}
```

```
        request.setAttributeDefinitions(attributeDefinitions);

        System.out.println("Issuing CreateTable request for " + tableName);
        Table table = dynamoDB.createTable(request);
        System.out.println("Waiting for " + tableName + " to be created...this may
take a while...");
        table.waitForActive();

    } catch (Exception e) {
        System.err.println("CreateTable request failed for " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void loadSampleProducts(String tableName) {

    Table table = dynamoDB.getTable(tableName);

    try {

        System.out.println("Adding data to " + tableName);

        Item item = new Item().withPrimaryKey("Id", 101).withString("Title", "Book
101 Title")
            .withString("ISBN", "111-1111111111")
            .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1"))).withNumber("Price", 2)
            .withString("Dimensions", "8.5 x 11.0 x
0.5").withNumber("PageCount", 500)
            .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 102).withString("Title", "Book 102
Title")
            .withString("ISBN", "222-2222222222")
            .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1", "Author2")))
            .withNumber("Price", 20).withString("Dimensions", "8.5 x 11.0 x
0.8").withNumber("PageCount", 600)
            .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
        table.putItem(item);

    }
```

```
item = new Item().withPrimaryKey("Id", 103).withString("Title", "Book 103
Title")
        .withString("ISBN", "333-333333333")
        .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1", "Author2")))
        // Intentional. Later we'll run Scan to find price error. Find
        // items > 1000 in price.
        .withNumber("Price", 2000).withString("Dimensions", "8.5 x 11.0 x
1.5").withNumber("PageCount", 600)
        .withBoolean("InPublication", false).withString("ProductCategory",
"Book");
    table.putItem(item);

    // Add bikes.

    item = new Item().withPrimaryKey("Id", 201).withString("Title", "18-
Bike-201")
        // Size, followed by some title.
        .withString("Description", "201
Description").withString("BicycleType", "Road")
        .withString("Brand", "Mountain A")
        // Trek, Specialized.
        .withNumber("Price", 100).withStringSet("Color", new
HashSet<String>(Arrays.asList("Red", "Black")))
        .withString("ProductCategory", "Bicycle");
    table.putItem(item);

    item = new Item().withPrimaryKey("Id", 202).withString("Title", "21-
Bike-202")
        .withString("Description", "202
Description").withString("BicycleType", "Road")
        .withString("Brand", "Brand-Company A").withNumber("Price", 200)
        .withStringSet("Color", new HashSet<String>(Arrays.asList("Green",
"Black")))
        .withString("ProductCategory", "Bicycle");
    table.putItem(item);

    item = new Item().withPrimaryKey("Id", 203).withString("Title", "19-
Bike-203")
        .withString("Description", "203
Description").withString("BicycleType", "Road")
        .withString("Brand", "Brand-Company B").withNumber("Price", 300)
        .withStringSet("Color", new HashSet<String>(Arrays.asList("Red",
"Green", "Black")))
    table.putItem(item);
```

```
        .withString("ProductCategory", "Bicycle");
    table.putItem(item);

    item = new Item().withPrimaryKey("Id", 204).withString("Title", "18-
Bike-204")
        .withString("Description", "204
Description").withString("BicycleType", "Mountain")
        .withString("Brand", "Brand-Company B").withNumber("Price", 400)
        .withStringSet("Color", new HashSet<String>(Arrays.asList("Red")))
        .withString("ProductCategory", "Bicycle");
    table.putItem(item);

    item = new Item().withPrimaryKey("Id", 205).withString("Title", "20-
Bike-205")
        .withString("Description", "205
Description").withString("BicycleType", "Hybrid")
        .withString("Brand", "Brand-Company C").withNumber("Price", 500)
        .withStringSet("Color", new HashSet<String>(Arrays.asList("Red",
"Black")))
        .withString("ProductCategory", "Bicycle");
    table.putItem(item);

} catch (Exception e) {
    System.err.println("Failed to create item in " + tableName);
    System.err.println(e.getMessage());
}

}

private static void loadSampleForums(String tableName) {

    Table table = dynamoDB.getTable(tableName);

    try {

        System.out.println("Adding data to " + tableName);

        Item item = new Item().withPrimaryKey("Name", "Amazon DynamoDB")
            .withString("Category", "Amazon Web
Services").withNumber("Threads", 2).withNumber("Messages", 4)
            .withNumber("Views", 1000);
        table.putItem(item);
    }
}
```

```
        item = new Item().withPrimaryKey("Name", "Amazon
S3").withString("Category", "Amazon Web Services")
            .withNumber("Threads", 0);
    table.putItem(item);

} catch (Exception e) {
    System.err.println("Failed to create item in " + tableName);
    System.err.println(e.getMessage());
}
}

private static void loadSampleThreads(String tableName) {
    try {
        long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000); // 7
        // days
        // ago
        long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000); // 14
        // days
        // ago
        long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000); // 21
        // days
        // ago

        Date date1 = new Date();
        date1.setTime(time1);

        Date date2 = new Date();
        date2.setTime(time2);

        Date date3 = new Date();
        date3.setTime(time3);

        dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

        Table table = dynamoDB.getTable(tableName);

        System.out.println("Adding data to " + tableName);

        Item item = new Item().withPrimaryKey("ForumName", "Amazon DynamoDB")
            .withString("Subject", "DynamoDB Thread 1").withString("Message",
"DynamoDB thread 1 message")
            .withString("LastPostedBy", "User
A").withString("LastPostedDateTime", dateFormatter.format(date2))
```

```
        .withNumber("Views", 0).withNumber("Replies",
0).withNumber("Answered", 0)
        .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"primarykey", "table")));
    table.putItem(item);

    item = new Item().withPrimaryKey("ForumName", "Amazon
DynamoDB").withString("Subject", "DynamoDB Thread 2")
        .withString("Message", "DynamoDB thread 2
message").withString("LastPostedBy", "User A")
        .withString("LastPostedDateTime",
dateFormatter.format(date3)).withNumber("Views", 0)
        .withNumber("Replies", 0).withNumber("Answered", 0)
        .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"partitionkey", "sortkey")));
    table.putItem(item);

    item = new Item().withPrimaryKey("ForumName", "Amazon
S3").withString("Subject", "S3 Thread 1")
        .withString("Message", "S3 Thread 3
message").withString("LastPostedBy", "User A")
        .withString("LastPostedDateTime",
dateFormatter.format(date1)).withNumber("Views", 0)
        .withNumber("Replies", 0).withNumber("Answered", 0)
        .withStringSet("Tags", new
HashSet<String>(Arrays.asList("largeobjects", "multipart upload")));
    table.putItem(item);

} catch (Exception e) {
    System.err.println("Failed to create item in " + tableName);
    System.err.println(e.getMessage());
}

}

private static void loadSampleReplies(String tableName) {
try {
    // 1 day ago
    long time0 = (new Date()).getTime() - (1 * 24 * 60 * 60 * 1000);
    // 7 days ago
    long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000);
    // 14 days ago
    long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000);
    // 21 days ago
}
```

```
long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000);

Date date0 = new Date();
date0.setTime(time0);

Date date1 = new Date();
date1.setTime(time1);

Date date2 = new Date();
date2.setTime(time2);

Date date3 = new Date();
date3.setTime(time3);

dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

Table table = dynamoDB.getTable(tableName);

System.out.println("Adding data to " + tableName);

// Add threads.

Item item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB
Thread 1")
    .withString("ReplyDateTime", (dateFormatter.format(date3)))
    .withString("Message", "DynamoDB Thread 1 Reply 1
text").withString("PostedBy", "User A");
table.putItem(item);

item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 1")
    .withString("ReplyDateTime", dateFormatter.format(date2))
    .withString("Message", "DynamoDB Thread 1 Reply 2
text").withString("PostedBy", "User B");
table.putItem(item);

item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
    .withString("ReplyDateTime", dateFormatter.format(date1))
    .withString("Message", "DynamoDB Thread 2 Reply 1
text").withString("PostedBy", "User A");
table.putItem(item);

item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
    .withString("ReplyDateTime", dateFormatter.format(date0))
```

```
        .withString("Message", "DynamoDB Thread 2 Reply 2
text").withString("PostedBy", "User A");
    table.putItem(item);

} catch (Exception e) {
    System.err.println("Failed to create item in " + tableName);
    System.err.println(e.getMessage());

}
}

}
```

Creating example tables and uploading data using the AWS SDK for .NET

The following C# code example creates tables and uploads data to the tables. The resulting table structure and data is shown in [Creating tables and loading data for code examples in DynamoDB](#). For step-by-step instructions to run this code in Visual Studio, see [.NET code examples](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class CreateTablesLoadData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                //DeleteAllTables(client);
                DeleteTable("ProductCatalog");
                DeleteTable("Forum");
```

```
        DeleteTable("Thread");
        DeleteTable("Reply");

        // Create tables (using the AWS SDK for .NET low-level API).
        CreateTableProductCatalog();
        CreateTableForum();
        CreateTableThread(); // ForumTitle, Subject */
        CreateTableReply();

        // Load data (using the .NET SDK document API)
        LoadSampleProducts();
        LoadSampleForums();
        LoadSampleThreads();
        LoadSampleReplies();
        Console.WriteLine("Sample complete!");
        Console.WriteLine("Press ENTER to continue");
        Console.ReadLine();
    }

    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
    catch (Exception e) { Console.WriteLine(e.Message); }
}

private static void DeleteTable(string tableName)
{
    try
    {
        var deleteTableResponse = client.DeleteTable(new DeleteTableRequest()
        {
            TableName = tableName
        });
        WaitTillTableDeleted(client, tableName, deleteTableResponse);
    }
    catch (ResourceNotFoundException)
    {
        // There is no such table.
    }
}

private static void CreateTableProductCatalog()
{
    string tableName = "ProductCatalog";

    var response = client.CreateTable(new CreateTableRequest
    {
```

```
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "Id",
                AttributeType = "N"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement
            {
                AttributeName = "Id",
                KeyType = "HASH"
            }
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    });

    WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableForum()
{
    string tableName = "Forum";

    var response = client.CreateTable(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "Name",
                AttributeType = "S"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
```

```
        new KeySchemaElement
        {
            AttributeName = "Name", // forum Title
            KeyType = "HASH"
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    );
}

WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableThread()
{
    string tableName = "Thread";

    var response = client.CreateTable(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "ForumName", // Hash attribute
                AttributeType = "S"
            },
            new AttributeDefinition
            {
                AttributeName = "Subject",
                AttributeType = "S"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement
            {
                AttributeName = "ForumName", // Hash attribute
                KeyType = "HASH"
            },
            new KeySchemaElement
```

```
        {
            AttributeName = "Subject", // Range attribute
            KeyType = "RANGE"
        }
    },
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = 10,
        WriteCapacityUnits = 5
    }
});

WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableReply()
{
    string tableName = "Reply";
    var response = client.CreateTable(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "Id",
                AttributeType = "S"
            },
            new AttributeDefinition
            {
                AttributeName = "ReplyDateTime",
                AttributeType = "S"
            },
            new AttributeDefinition
            {
                AttributeName = "PostedBy",
                AttributeType = "S"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement()
            {
                AttributeName = "Id",

```

```
        KeyType = "HASH"
    },
    new KeySchemaElement()
    {
        AttributeName = "ReplyDateTime",
        KeyType = "RANGE"
    }
},
LocalSecondaryIndexes = new List<LocalSecondaryIndex>()
{
    new LocalSecondaryIndex()
    {
        IndexName = "PostedBy_index",

        KeySchema = new List<KeySchemaElement>() {
            new KeySchemaElement() {
                AttributeName = "Id", KeyType = "HASH"
            },
            new KeySchemaElement() {
                AttributeName = "PostedBy", KeyType =
"RANGE"
            }
        },
        Projection = new Projection() {
            ProjectionType = ProjectionType.KEYS_ONLY
        }
    }
},
ProvisionedThroughput = new ProvisionedThroughput
{
    ReadCapacityUnits = 10,
    WriteCapacityUnits = 5
}
});

WaitTillTableCreated(client, tableName, response);
}

private static void WaitTillTableCreated(AmazonDynamoDBClient client, string
tableName,
                                         CreateTableResponse response)
{
    var tableDescription = response.TableDescription;
```

```
string status = tableDescription.TableStatus;

Console.WriteLine(tableName + " - " + status);

// Let us wait until table is created. Call DescribeTable.
while (status != "ACTIVE")
{
    System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
    try
    {
        var res = client.DescribeTable(new DescribeTableRequest
        {
            TableName = tableName
        });
        Console.WriteLine("Table name: {0}, status: {1}",
res.Table.TableName,
                res.Table.TableStatus);
        status = res.Table.TableStatus;
    }
    // Try-catch to handle potential eventual-consistency issue.
    catch (ResourceNotFoundException)
    { }
}
}

private static void WaitTillTableDeleted(AmazonDynamoDBClient client, string
tableName,
                                         DeleteTableResponse response)
{
    var tableDescription = response.TableDescription;

    string status = tableDescription.TableStatus;

    Console.WriteLine(tableName + " - " + status);

    // Let us wait until table is created. Call DescribeTable
    try
    {
        while (status == "DELETING")
        {
            System.Threading.Thread.Sleep(5000); // wait 5 seconds

            var res = client.DescribeTable(new DescribeTableRequest
```

```
        {
            TableName = tableName
        });
        Console.WriteLine("Table name: {0}, status: {1}",
res.Table.TableName,
                res.Table.TableStatus);
        status = res.Table.TableStatus;
    }
}

catch (ResourceNotFoundException)
{
    // Table deleted.
}
}

private static void LoadSampleProducts()
{
    Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
    // ***** Add Books *****
    var book1 = new Document();
    book1["Id"] = 101;
    book1["Title"] = "Book 101 Title";
    book1["ISBN"] = "111-1111111111";
    book1["Authors"] = new List<string> { "Author 1" };
    book1["Price"] = -2; // *** Intentional value. Later used to illustrate
scan.
    book1["Dimensions"] = "8.5 x 11.0 x 0.5";
    book1["PageCount"] = 500;
    book1["InPublication"] = true;
    book1["ProductCategory"] = "Book";
    productCatalogTable.PutItem(book1);

    var book2 = new Document();

    book2["Id"] = 102;
    book2["Title"] = "Book 102 Title";
    book2["ISBN"] = "222-2222222222";
    book2["Authors"] = new List<string> { "Author 1", "Author 2" } ; ;
    book2["Price"] = 20;
    book2["Dimensions"] = "8.5 x 11.0 x 0.8";
    book2["PageCount"] = 600;
    book2["InPublication"] = true;
    book2["ProductCategory"] = "Book";
    productCatalogTable.PutItem(book2);
```

```
var book3 = new Document();
book3["Id"] = 103;
book3["Title"] = "Book 103 Title";
book3["ISBN"] = "333-333333333";
book3["Authors"] = new List<string> { "Author 1", "Author2", "Author
3" }; ;
book3["Price"] = 2000;
book3["Dimensions"] = "8.5 x 11.0 x 1.5";
book3["PageCount"] = 700;
book3["InPublication"] = false;
book3["ProductCategory"] = "Book";
productCatalogTable.PutItem(book3);

// ***** Add bikes. *****
var bicycle1 = new Document();
bicycle1["Id"] = 201;
bicycle1["Title"] = "18-Bike 201"; // size, followed by some title.
bicycle1["Description"] = "201 description";
bicycle1["BicycleType"] = "Road";
bicycle1["Brand"] = "Brand-Company A"; // Trek, Specialized.
bicycle1["Price"] = 100;
bicycle1["Color"] = new List<string> { "Red", "Black" };
bicycle1["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle1);

var bicycle2 = new Document();
bicycle2["Id"] = 202;
bicycle2["Title"] = "21-Bike 202Brand-Company A";
bicycle2["Description"] = "202 description";
bicycle2["BicycleType"] = "Road";
bicycle2["Brand"] = "";
bicycle2["Price"] = 200;
bicycle2["Color"] = new List<string> { "Green", "Black" };
bicycle2["ProductCategory"] = "Bicycle";
productCatalogTable.PutItem(bicycle2);

var bicycle3 = new Document();
bicycle3["Id"] = 203;
bicycle3["Title"] = "19-Bike 203";
bicycle3["Description"] = "203 description";
bicycle3["BicycleType"] = "Road";
bicycle3["Brand"] = "Brand-Company B";
bicycle3["Price"] = 300;
```

```
bicycle3["Color"] = new List<string> { "Red", "Green", "Black" };
bicycle3["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle3);

var bicycle4 = new Document();
bicycle4["Id"] = 204;
bicycle4["Title"] = "18-Bike 204";
bicycle4["Description"] = "204 description";
bicycle4["BicycleType"] = "Mountain";
bicycle4["Brand"] = "Brand-Company B";
bicycle4["Price"] = 400;
bicycle4["Color"] = new List<string> { "Red" };
bicycle4["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle4);

var bicycle5 = new Document();
bicycle5["Id"] = 205;
bicycle5["Title"] = "20-Title 205";
bicycle5["Description"] = "205 description";
bicycle5["BicycleType"] = "Hybrid";
bicycle5["Brand"] = "Brand-Company C";
bicycle5["Price"] = 500;
bicycle5["Color"] = new List<string> { "Red", "Black" };
bicycle5["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle5);
}

private static void LoadSampleForums()
{
    Table forumTable = Table.LoadTable(client, "Forum");

    var forum1 = new Document();
    forum1["Name"] = "Amazon DynamoDB"; // PK
    forum1["Category"] = "Amazon Web Services";
    forum1["Threads"] = 2;
    forum1["Messages"] = 4;
    forum1["Views"] = 1000;

    forumTable.PutItem(forum1);

    var forum2 = new Document();
    forum2["Name"] = "Amazon S3"; // PK
    forum2["Category"] = "Amazon Web Services";
    forum2["Threads"] = 1;
```

```
        forumTable.PutItem(forum2);
    }

private static void LoadSampleThreads()
{
    Table threadTable = Table.LoadTable(client, "Thread");

    // Thread 1.
    var thread1 = new Document();
    thread1["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
    thread1["Subject"] = "DynamoDB Thread 1"; // Range attribute.
    thread1["Message"] = "DynamoDB thread 1 message text";
    thread1["LastPostedBy"] = "User A";
    thread1["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14,
0, 0, 0));
    thread1["Views"] = 0;
    thread1["Replies"] = 0;
    thread1["Answered"] = false;
    thread1["Tags"] = new List<string> { "index", "primarykey", "table" };

    threadTable.PutItem(thread1);

    // Thread 2.
    var thread2 = new Document();
    thread2["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
    thread2["Subject"] = "DynamoDB Thread 2"; // Range attribute.
    thread2["Message"] = "DynamoDB thread 2 message text";
    thread2["LastPostedBy"] = "User A";
    thread2["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21,
0, 0, 0));
    thread2["Views"] = 0;
    thread2["Replies"] = 0;
    thread2["Answered"] = false;
    thread2["Tags"] = new List<string> { "index", "primarykey", "rangekey" };

    threadTable.PutItem(thread2);

    // Thread 3.
    var thread3 = new Document();
    thread3["ForumName"] = "Amazon S3"; // Hash attribute.
    thread3["Subject"] = "S3 Thread 1"; // Range attribute.
    thread3["Message"] = "S3 thread 3 message text";
    thread3["LastPostedBy"] = "User A";
```

```
        thread3["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0, 0, 0));
        thread3["Views"] = 0;
        thread3["Replies"] = 0;
        thread3["Answered"] = false;
        thread3["Tags"] = new List<string> { "largeobjects", "multipart upload" };
        threadTable.PutItem(thread3);
    }

    private static void LoadSampleReplies()
    {
        Table replyTable = Table.LoadTable(client, "Reply");

        // Reply 1 - thread 1.
        var thread1Reply1 = new Document();
        thread1Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
        thread1Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0, 0)); // Range attribute.
        thread1Reply1["Message"] = "DynamoDB Thread 1 Reply 1 text";
        thread1Reply1["PostedBy"] = "User A";

        replyTable.PutItem(thread1Reply1);

        // Reply 2 - thread 1.
        var thread1reply2 = new Document();
        thread1reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
        thread1reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0)); // Range attribute.
        thread1reply2["Message"] = "DynamoDB Thread 1 Reply 2 text";
        thread1reply2["PostedBy"] = "User B";

        replyTable.PutItem(thread1reply2);

        // Reply 3 - thread 1.
        var thread1Reply3 = new Document();
        thread1Reply3["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
        thread1Reply3["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0, 0, 0)); // Range attribute.
        thread1Reply3["Message"] = "DynamoDB Thread 1 Reply 3 text";
        thread1Reply3["PostedBy"] = "User B";
```

```
    replyTable.PutItem(thread1Reply3);

    // Reply 1 - thread 2.
    var thread2Reply1 = new Document();
    thread2Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash
attribute.
    thread2Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7,
0, 0, 0)); // Range attribute.
    thread2Reply1["Message"] = "DynamoDB Thread 2 Reply 1 text";
    thread2Reply1["PostedBy"] = "User A";

    replyTable.PutItem(thread2Reply1);

    // Reply 2 - thread 2.
    var thread2Reply2 = new Document();
    thread2Reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash
attribute.
    thread2Reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(1,
0, 0, 0)); // Range attribute.
    thread2Reply2["Message"] = "DynamoDB Thread 2 Reply 2 text";
    thread2Reply2["PostedBy"] = "User A";

    replyTable.PutItem(thread2Reply2);
}
}

}
```

DynamoDB example application using the AWS SDK for Python (Boto): Tic-tac-toe

Topics

- [Step 1: Deploy and test locally](#)
- [Step 2: Examine the data model and implementation details](#)
- [Step 3: Deploy in production using the DynamoDB service](#)
- [Step 4: Clean up resources](#)

The Tic-Tac-Toe game is an example web application built on Amazon DynamoDB. The application uses the AWS SDK for Python (Boto) to make the necessary DynamoDB calls to store game data

in a DynamoDB table, and the Python web framework Flask to illustrate end-to-end application development in DynamoDB, including how to model data. It also demonstrates best practices when it comes to modeling data in DynamoDB, including the table you create for the game application, the primary key you define, additional indexes you need based on your query requirements, and the use of concatenated value attributes.

You play the Tic-Tac-Toe application on the web as follows:

1. You log in to the application home page.
2. You then invite another user to play the game as your opponent.

Until another user accepts your invitation, the game status remains as PENDING. After an opponent accepts the invite, the game status changes to IN_PROGRESS.

3. The game begins after the opponent logs in and accepts the invite.
4. The application stores all game moves and status information in a DynamoDB table.
5. The game ends with a win or a draw, which sets the game status to FINISHED.

The end-to-end application building exercise is described in steps:

- **Step 1: Deploy and test locally** – In this section, you download, deploy, and test the application on your local computer. You will create the required tables in the downloadable version of DynamoDB.
- **Step 2: Examine the data model and implementation details** – This section first describes in detail the data model, including the indexes and the use of the concatenated value attribute. Then the section explains how the application works.
- **Step 3: Deploy in production using the DynamoDB service** – This section focuses on deployment considerations in production. In this step, you create a table using the Amazon DynamoDB service and deploy the application using AWS Elastic Beanstalk. When you have the application in production, you also grant appropriate permissions so the application can access the DynamoDB table. The instructions in this section walk you through the end-to-end production deployment.
- **Step 4: Clean up resources** – This section highlights areas that are not covered by this example. The section also provides steps for you to remove the AWS resources you created in the preceding steps so that you avoid incurring any charges.

Step 1: Deploy and test locally

Topics

- [1.1: Download and install the required packages](#)
- [1.2: Test the game application](#)

In this step you download, deploy, and test the Tic-Tac-Toe game application on your local computer. Instead of using the Amazon DynamoDB web service, you will download DynamoDB to your computer, and create the required table there.

1.1: Download and install the required packages

You will need the following to test this application locally:

- Python
- Flask (a microframework for Python)
- AWS SDK for Python (Boto)
- DynamoDB running on your computer
- Git

To get these tools, do the following:

1. Install Python. For step-by-step instructions, see [Download Python](#).

The Tic-Tac-Toe application has been tested using Python version 2.7.

2. Install Flask and AWS SDK for Python (Boto) using the Python Package Installer (PIP):

- Install PIP.

For instructions, see [Install PIP](#). On the installation page, choose the **get-pip.py** link, and then save the file. Then open a command terminal as an administrator, and enter the following at the command prompt.

```
python.exe get-pip.py
```

On Linux, you don't specify the .exe extension. You only specify `python get-pip.py`.

- Using PIP, install the Flask and Boto packages using the following code.

```
pip install Flask  
pip install boto  
pip install configparser
```

3. Download DynamoDB to your computer. For instructions on how to run it, see [Setting up DynamoDB local \(downloadable version\)](#).
4. Download the Tic-Tac-Toe application:
 - a. Install Git. For instructions, see [git downloads](#).
 - b. Run the following code to download the application.

```
git clone https://github.com/awslabs/dynamodb-tictactoe-example-app.git
```

1.2: Test the game application

To test the Tic-Tac-Toe application, you need to run DynamoDB locally on your computer.

To run the tic-tac-toe application

1. Start DynamoDB.
2. Start the web server for the Tic-Tac-Toe application.

To do so, open a command terminal, navigate to the folder where you downloaded the Tic-Tac-Toe application, and run the application locally using the following code.

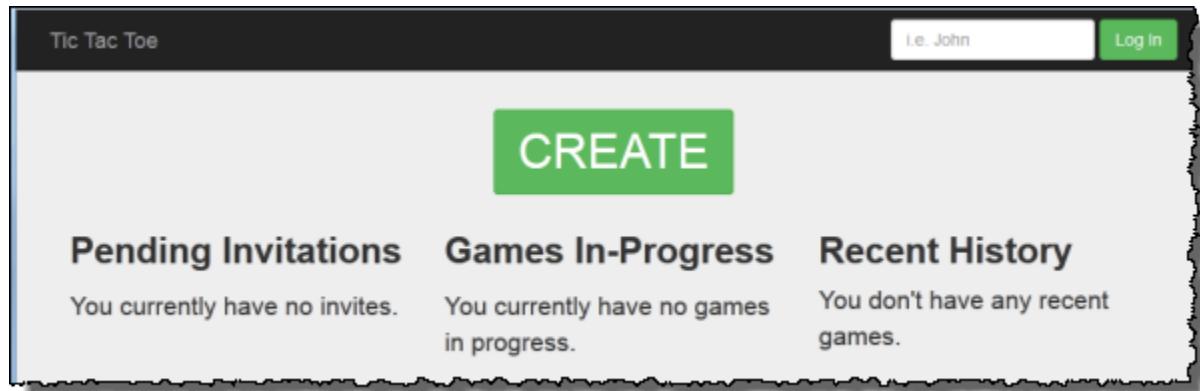
```
python.exe application.py --mode local --serverPort 5000 --port 8000
```

On Linux, you don't specify the .exe extension.

3. Open your web browser, and enter the following.

```
http://localhost:5000/
```

The browser shows the home page.

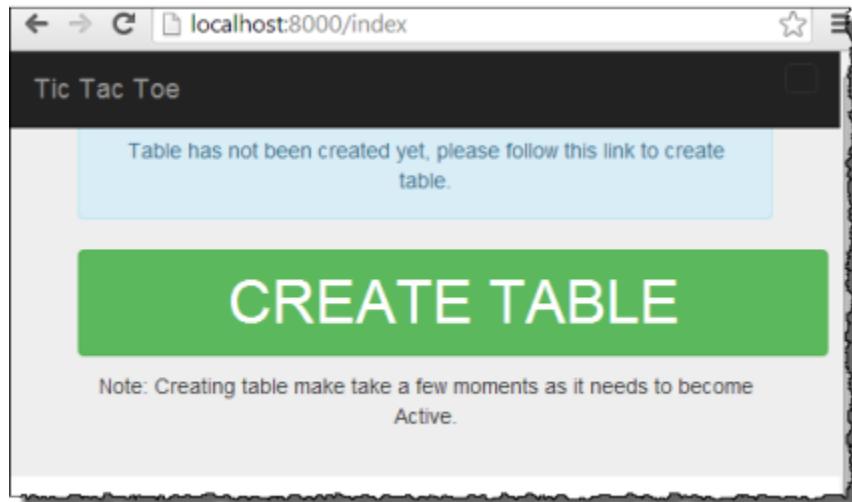


4. Enter **user1** in the **Log in** box to log in as user1.

Note

This example application does not perform any user authentication. The user ID is only used to identify players. If two players log in with the same alias, the application works as if you are playing in two different browsers.

5. If this is your first time playing the game, a page appears requesting you to create the required table (Games) in DynamoDB. Choose **CREATE TABLE**.



6. Choose **CREATE** to create the first tic-tac-toe game.
7. Enter **user2** in the **Choose an Opponent** box, and choose **Create Game!**



Doing this creates the game by adding an item in the Games table. It sets the game status to PENDING.

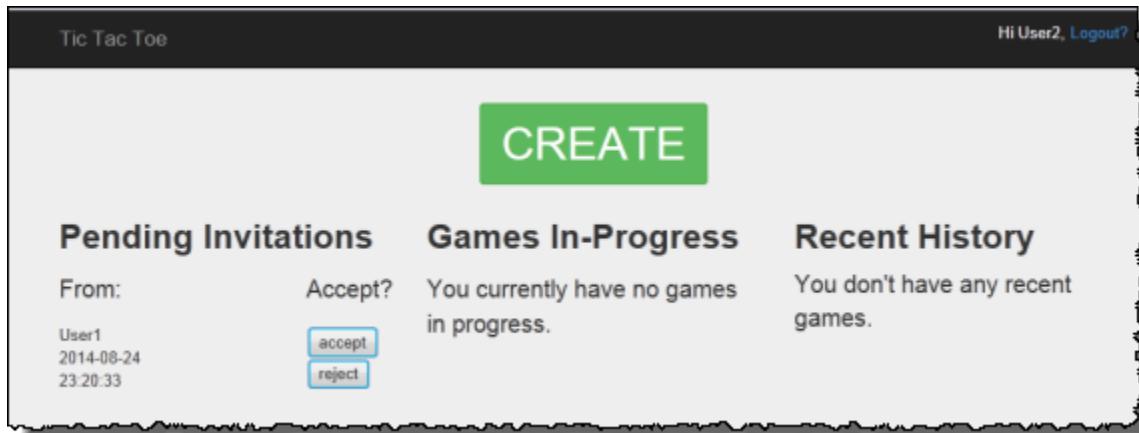
8. Open another browser window, and enter the following.

```
http://localhost:5000/
```

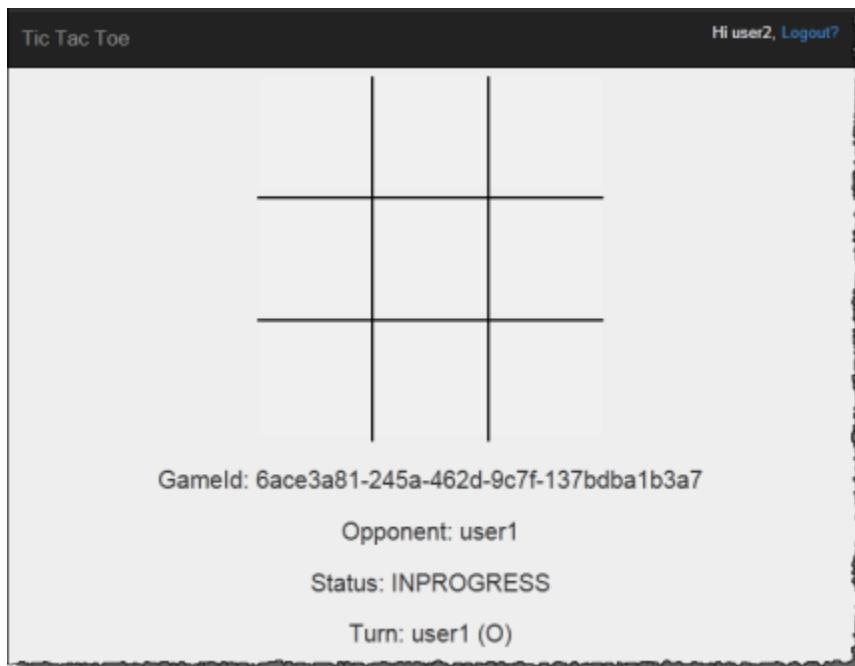
The browser passes information through cookies, so you should use incognito mode or private browsing so that your cookies don't carry over.

9. Log in as user2.

A page appears that shows a pending invitation from user1.



10. Choose **accept** to accept the invitation.



The game page appears with an empty tic-tac-toe grid. The page also shows relevant game information such as the game ID, whose turn it is, and game status.

11. Play the game.

For each user move, the web service sends a request to DynamoDB to conditionally update the game item in the Games table. For example, the conditions ensure that the move is valid, the square that the user chose is available, and that it was the turn of the user who made the move. For a valid move, the update operation adds a new attribute corresponding to the selection on the board. The update operation also sets the value of the existing attribute to the user who can make the next move.

On the game page, the application makes asynchronous JavaScript calls every second, for up to 5 minutes, to check if the game state in DynamoDB has changed. If it has, the application updates the page with new information. After 5 minutes, the application stops making the requests, and you need to refresh the page to get updated information.

Step 2: Examine the data model and implementation details

Topics

- [2.1: Basic data model](#)
- [2.2: Application in action \(code walkthrough\)](#)

2.1: Basic data model

This example application highlights the following DynamoDB data model concepts:

- **Table** – In DynamoDB, a table is a collection of items (that is, records), and each item is a collection of name-value pairs called attributes.

In this Tic-Tac-Toe example, the application stores all game data in a table, Games. The application creates one item in the table per game and stores all game data as attributes. A tic-tac-toe game can have up to nine moves. Because DynamoDB tables do not have a schema in cases where only the primary key is the required attribute, the application can store varying number of attributes per game item.

The Games table has a simple primary key made of one attribute, GameId, of string type. The application assigns a unique ID to each game. For more information on DynamoDB primary keys, see [Primary key](#).

When a user initiates a tic-tac-toe game by inviting another user to play, the application creates a new item in the Games table with attributes storing game metadata, such as the following:

- HostId, the user who initiated the game.
- Opponent, the user who was invited to play.
- The user whose turn it is to play. The user who initiated the game plays first.
- The user who uses the O symbol on the board. The user who initiates the games uses the O symbol.

In addition, the application creates a StatusDate concatenated attribute, marking the initial game state as PENDING. The following screenshot shows an example item as it appears in the DynamoDB console:

Amazon DynamoDB Explore Table Games		
List Tables	Browse Items	Item Details
Edit Item	Copy to New	Delete Item
Attribute	Type	Value
Gameld (Hash Key)	String	"6ffd7f5-e293-4b4a-bacf-6ddde49ef0ae"
HostId	String	"user1"
O	String	"user1"
Opponent	String	"user2"
StatusDate	String	"PENDING_2014-07-06 21:28:02.354807"
Turn	String	"user1"

As the game progresses, the application adds one attribute to the table for each game move. The attribute name is the board position, for example TopLeft or BottomRight. For example, a move might have a TopLeft attribute with the value 0, a TopRight attribute with the value 0, and a BottomRight attribute with the value X. The attribute value is either 0 or X, depending on which user made the move. For example, consider the following board.



- Concatenated value attributes** – The StatusDate attribute illustrates a concatenated value attribute. In this approach, instead of creating separate attributes to store game status (PENDING, IN_PROGRESS, and FINISHED) and date (when the last move was made), you combine them as single attribute, for example IN_PROGRESS_2014-04-30 10:20:32.

The application then uses the StatusDate attribute in creating secondary indexes by specifying StatusDate as a sort key for the index. The benefit of using the StatusDate concatenated value attribute is further illustrated in the indexes discussed next.

- **Global secondary indexes** – You can use the table's primary key, GameId, to efficiently query the table to find a game item. To query the table on attributes other than the primary key attributes, DynamoDB supports the creation of secondary indexes. In this example application, you build the following two secondary indexes:

Global Secondary Indexes										
Index Name	Hash Key	Range Key	Projected Attributes	Status	Read Capacity Units	Write Capacity Units	Last Decrease Time	Last Increase Time	Index Size (Bytes)*	Item Count*
This table has no local secondary indexes.										
hostStatusDate	HostId (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125
oppStatusDate	OpponentId (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125

- **HostId-StatusDate-index**. This index has HostId as a partition key and StatusDate as a sort key. You can use this index to query on HostId, for example to find games hosted by a particular user.
- **OpponentId-StatusDate-index**. This index has OpponentId as a partition key and StatusDate as a sort key. You can use this index to query on Opponent, for example to find games where a particular user is the opponent.

These indexes are called global secondary indexes because the partition key in these indexes is not the same the partition key (GameId), used in the primary key of the table.

Note that both the indexes specify StatusDate as a sort key. Doing this enables the following:

- You can query using the BEGINS_WITH comparison operator. For example, you can find all games with the IN_PROGRESS attribute hosted by a particular user. In this case, the BEGINS_WITH operator checks for the StatusDate value that begins with IN_PROGRESS.
- DynamoDB stores the items in the index in sorted order, by sort key value. So if all status prefixes are the same (for example, IN_PROGRESS), the ISO format used for the date part will have items sorted from oldest to the newest. This approach enables certain queries to be performed efficiently, for example the following:
 - Retrieve up to 10 of the most recent IN_PROGRESS games hosted by the user who is logged in. For this query, you specify the HostId-StatusDate-index index.
 - Retrieve up to 10 of the most recent IN_PROGRESS games where the user logged in is the opponent. For this query, you specify the OpponentId-StatusDate-index index.

For more information about secondary indexes, see [Improving data access with secondary indexes](#).

2.2: Application in action (code walkthrough)

This application has two main pages:

- **Home page** – This page provides the user a simple login, a **CREATE** button to create a new tic-tac-toe game, a list of games in progress, game history, and any active pending game invitations.

The home page is not refreshed automatically; you must refresh the page to refresh the lists.

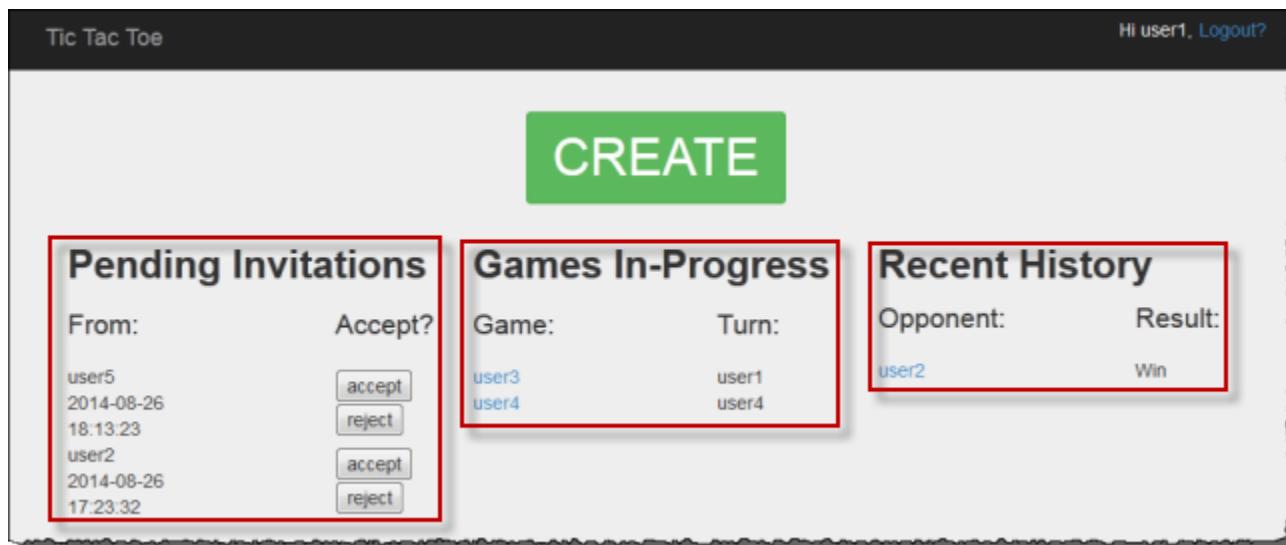
- **Game page** – This page shows the tic-tac-toe grid where users play.

The application updates the game page automatically every second. The JavaScript in your browser calls the Python web server every second to query the Games table whether the game items in the table have changed. If they have, JavaScript triggers a page refresh so that the user sees the updated board.

Let us see in detail how the application works.

Home page

After the user logs in, the application displays the following three lists of information.



- **Invitations** – This list shows up to the 10 most recent invitations from others that are pending acceptance by the user who is logged in. In the preceding screenshot, user1 has invitations from user5 and user2 pending.

- **Games in-progress** – This list shows up to the 10 most recent games that are in progress. These are games that the user is actively playing, which have the status IN_PROGRESS. In the screenshot, user1 is actively playing a tic-tac-toe game with user3 and user4.
- **Recent history** – This list shows up to the 10 most recent games that the user finished, which have the status FINISHED. In game shown in the screenshot, user1 has previously played with user2. For each completed game, the list shows the game result.

In the code, the `index` function (in `application.py`) makes the following three calls to retrieve game status information:

```
inviteGames      = controller.getGameInvites(session["username"])
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
finishedGames    = controller.getGamesWithStatus(session["username"], "FINISHED")
```

Each of these calls returns a list of items from DynamoDB that are wrapped by the Game objects. It is easy to extract data from these objects in the view. The `index` function passes these object lists to the view to render the HTML.

```
return render_template("index.html",
                      user=session["username"],
                      invites=inviteGames,
                      inprogress=inProgressGames,
                      finished=finishedGames)
```

The Tic-Tac-Toe application defines the Game class primarily to store game data retrieved from DynamoDB. These functions return lists of Game objects that enable you to isolate the rest of the application from code related to Amazon DynamoDB items. Thus, these functions help you decouple your application code from the details of the data store layer.

The architectural pattern described here is also referred as the model-view-controller (MVC) UI pattern. In this case, the Game object instances (representing data) are the model, and the HTML page is the view. The controller is divided into two files. The `application.py` file has the controller logic for the Flask framework, and the business logic is isolated in the `gameController.py` file. That is, the application stores everything that has to do with DynamoDB SDK in its own separate file in the dynamodb folder.

Let us review the three functions and how they query the Games table using global secondary indexes to retrieve relevant data.

Using getGameInvites to get the list of pending game invitations

The `getGameInvites` function retrieves the list of the 10 most recent pending invitations. These games have been created by users, but the opponents have not accepted the game invitations. For these games, the status remains PENDING until the opponent accepts the invite. If the opponent declines the invite, the application removes the corresponding item from the table.

The function specifies the query as follows:

- It specifies the OpponentId-StatusDate-index index to use with the following index key values and comparison operators:
 - The partition key is `OpponentId` and takes the index key `user ID`.
 - The sort key is `StatusDate` and takes the comparison operator and index key value `beginswith="PENDING_"`.

You use the `OpponentId-StatusDate-index` index to retrieve games to which the logged-in user is invited—that is, where the logged-in user is the opponent.

- The query limits the result to 10 items.

```
gameInvitesIndex = self.cm.getGamesTable().query(  
    Opponent__eq=user,  
    StatusDate__beginswith="PENDING_",  
    index="OpponentId-StatusDate-index",  
    limit=10)
```

In the index, for each `OpponentId` (the partition key) DynamoDB keeps items sorted by `StatusDate` (the sort key). Therefore, the games that the query returns will be the 10 most recent games.

Using getGamesWithStatus to get the list of games with a specific status

After an opponent accepts a game invitation, the game status changes to IN_PROGRESS. After the game completes, the status changes to FINISHED.

Queries to find games that are either in progress or finished are the same except for the different status value. Therefore, the application defines the `getGamesWithStatus` function, which takes the status value as a parameter.

```
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
```

```
finishedGames = controller.getGamesWithStatus(session["username"], "FINISHED")
```

The following section discusses in-progress games, but the same description also applies to finished games.

A list of in-progress games for a given user includes both the following:

- In-progress games hosted by the user
- In-progress games where the user is the opponent

The `getGamesWithStatus` function runs the following two queries, each time using the appropriate secondary index.

- The function queries the `Games` table using the `HostId-StatusDate-index` index. For the index, the query specifies primary key values—both the partition key (`HostId`) and sort key (`StatusDate`) values, along with comparison operators.

```
hostGamesInProgress = self.cm.getGamesTable().query(HostId__eq=user,  
                                                StatusDate__beginswith=status,  
                                                index="HostId-StatusDate-index",  
                                                limit=10)
```

Note the Python syntax for comparison operators:

- `HostId__eq=user` specifies the equality comparison operator.
- `StatusDate__beginswith=status` specifies the `BEGINS_WITH` comparison operator.
- The function queries the `Games` table using the `OpponentId-StatusDate-index` index.

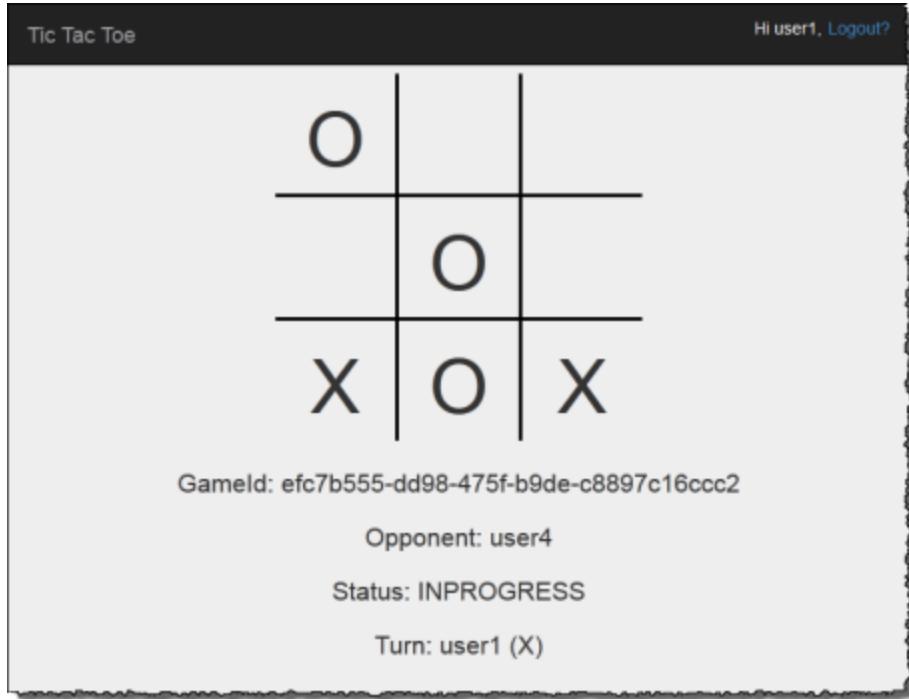
```
oppGamesInProgress = self.cm.getGamesTable().query(Opponent__eq=user,  
                                                StatusDate__beginswith=status,  
                                                index="OpponentId-StatusDate-index",  
                                                limit=10)
```

- The function then combines the two lists, sorts, and for the first 0 to 10 items creates a list of the `Game` objects and returns the list to the calling function (that is, the index).

```
games = self.mergeQueries(hostGamesInProgress,  
                           oppGamesInProgress)  
return games
```

Game page

The game page is where the user plays tic-tac-toe games. It shows the game grid along with game-relevant information. The following screenshot shows an example game in progress:



The application displays the game page in the following situations:

- The user creates a game inviting another user to play.

In this case, the page shows the user as host and the game status as PENDING, waiting for the opponent to accept.

- The user accepts one of the pending invitations on the home page.

In this case, the page shows the user as the opponent and game status as IN_PROGRESS.

A user selection on the board generates a form POST request to the application. That is, Flask calls the `selectSquare` function (in `application.py`) with the HTML form data. This function, in turn, calls the `updateBoardAndTurn` function (in `gameController.py`) to update the game item as follows:

- It adds a new attribute specific to the move.
- It updates the `Turn` attribute value to the user whose turn is next.

```
controller.updateBoardAndTurn(item, value, session["username"])
```

The function returns true if the item update was successful; otherwise, it returns false. Note the following about the `updateBoardAndTurn` function:

- The function calls the `update_item` function of the SDK for Python to make a finite set of updates to an existing item. The function maps to the `UpdateItem` operation in DynamoDB. For more information, see [UpdateItem](#).

 **Note**

The difference between the `UpdateItem` and `PutItem` operations is that `PutItem` replaces the entire item. For more information, see [PutItem](#).

For the `update_item` call, the code identifies the following:

- The primary key of the Games table (that is, `ItemId`).

```
key = { "GameId" : { "S" : gameId } }
```

- The new attribute to add, specific to the current user move, and its value (for example, `TopLeft="X"`).

```
attributeUpdates = {
    position : {
        "Action" : "PUT",
        "Value" : { "S" : representation }
    }
}
```

- Conditions that must be true for the update to take place:
 - The game must be in progress. That is, the `StatusDate` attribute value must begin with `IN_PROGRESS`.
 - The current turn must be a valid user turn as specified by the `Turn` attribute.
 - The square that the user chose must be available. That is, the attribute corresponding to the square must not exist.

```
expectations = {"StatusDate" : {"AttributeValueList": [{"S" : "IN_PROGRESS_"}]},  
    "ComparisonOperator": "BEGINS_WITH"},  
    "Turn" : {"Value" : {"S" : current_player}},  
    position : {"Exists" : False}}
```

Now the function calls `update_item` to update the item.

```
self.cm.db.update_item("Games", key=key,  
    attribute_updates=attributeUpdates,  
    expected=expectations)
```

After the function returns, the `selectSquare` function calls `redirect`, as shown in the following example.

```
redirect("/game="+gameId)
```

This call causes the browser to refresh. As part of this refresh, the application checks to see if the game has ended in a win or draw. If it has, the application updates the game item accordingly.

Step 3: Deploy in production using the DynamoDB service

Topics

- [3.1: Create an IAM role for Amazon EC2](#)
- [3.2: Create the games table in Amazon DynamoDB](#)
- [3.3: Bundle and deploy the tic-tac-toe application code](#)
- [3.4: Set up the AWS Elastic Beanstalk environment](#)

In the preceding sections, you deployed and tested the Tic-Tac-Toe application locally on your computer using DynamoDB local. Now, you deploy the application in production as follows:

- Deploy the application using AWS Elastic Beanstalk, an easy-to-use service for deploying and scaling web applications and services. For more information, see [Deploying a flask application to AWS Elastic Beanstalk](#).

Elastic Beanstalk launches one or more Amazon Elastic Compute Cloud (Amazon EC2) instances, which you configure through Elastic Beanstalk, on which your Tic-Tac-Toe application will run.

- Using the Amazon DynamoDB service, create a Games table that exists on AWS rather than locally on your computer.

In addition, you also have to configure permissions. Any AWS resources you create, such as the Games table in DynamoDB, are private by default. Only the resource owner, that is the AWS account that created the Games table, can access this table. Thus, by default your Tic-Tac-Toe application cannot update the Games table.

To grant necessary permissions, you create an AWS Identity and Access Management (IAM) role and grant this role permissions to access the Games table. Your Amazon EC2 instance first assumes this role. In response, AWS returns temporary security credentials that the Amazon EC2 instance can use to update the Games table on behalf of the Tic-Tac-Toe application. When you configure your Elastic Beanstalk application, you specify the IAM role that the Amazon EC2 instance or instances can assume. For more information about IAM roles, see [IAM roles for amazon EC2](#) in the *Amazon EC2 User Guide for Linux Instances*.

 **Note**

Before you create Amazon EC2 instances for the Tic-Tac-Toe application, you must first decide the AWS Region where you want Elastic Beanstalk to create the instances. After you create the Elastic Beanstalk application, you provide the same Region name and endpoint in a configuration file. The Tic-Tac-Toe application uses information in this file to create the Games table and send subsequent requests in a specific AWS Region. Both the DynamoDB Games table and the Amazon EC2 instances that Elastic Beanstalk launches must be in the same Region. For a list of available Regions, see [Amazon DynamoDB](#) in the *Amazon Web Services General Reference*.

In summary, you do the following to deploy the Tic-Tac-Toe application in production:

1. Create an IAM role using the IAM service. You attach a policy to this role granting permissions for DynamoDB actions to access the Games table.
2. Bundle the Tic-Tac-Toe application code and a configuration file, and create a .zip file. You use this .zip file to give the Tic-Tac-Toe application code to Elastic Beanstalk to put on your servers. For more information about creating a bundle, see [Creating an application source bundle](#) in the *AWS Elastic Beanstalk Developer Guide*.

In the configuration file (`beanstalk.config`), you provide AWS Region and endpoint information. The Tic-Tac-Toe application uses this information to determine which DynamoDB Region to talk to.

3. Set up the Elastic Beanstalk environment. Elastic Beanstalk launches an Amazon EC2 instance or instances and deploys your Tic-Tac-Toe application bundle on them. After the Elastic Beanstalk environment is ready, you provide the configuration file name by adding the `CONFIG_FILE` environment variable.
4. Create the DynamoDB table. Using the Amazon DynamoDB service, you create the Games table on AWS, rather than locally on your computer. Remember, this table has a simple primary key made of the GameId partition key of string type.
5. Test the game in production.

3.1: Create an IAM role for Amazon EC2

Creating an IAM role of the **Amazon EC2** type allows the Amazon EC2 instance that is running your Tic-Tac-Toe application to assume the correct role and make application requests to access the Games table. When creating the role, choose the **Custom Policy** option and copy and paste the following policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dynamodb>ListTables"  
            ],  
            "Effect": "Allow",  
            "Resource": "*"  
        },  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:922852403271:table/Games",  
                "arn:aws:dynamodb:us-west-2:922852403271:table/Games/index/*"  
            ]  
        }  
    ]  
}
```

```
    }  
]  
}
```

For further instructions, see [Creating a role for an AWS service \(AWS Management Console\)](#) in the [IAM User Guide](#).

3.2: Create the games table in Amazon DynamoDB

The Games table in DynamoDB stores game data. If the table does not exist, the application creates the table for you. In this case, let the application create the Games table.

3.3: Bundle and deploy the tic-tac-toe application code

If you followed this example's steps, then you already have the downloaded the Tic-Tac-Toe application. If not, download the application and extract all the files to a folder on your local computer. For instructions, see [Step 1: Deploy and test locally](#).

After you extract all files, you will have a code folder. To hand off this folder to Elastic Beanstalk, you bundle the contents of this folder as a .zip file. First, you add a configuration file to that folder. Your application uses the Region and endpoint information to create a DynamoDB table in the specified Region and make subsequent table operation requests using the specified endpoint.

1. Switch to the folder where you downloaded the Tic-Tac-Toe application.
2. In the root folder of the application, create a text file named beanstalk.config with the following content.

```
[dynamodb]  
region=<AWS region>  
endpoint=<Amazon DynamoDB endpoint>
```

For example, you might use the following content.

```
[dynamodb]  
region=us-west-2  
endpoint=dynamodb.us-west-2.amazonaws.com
```

For a list of available Regions, see [Amazon DynamoDB](#) in the [Amazon Web Services General Reference](#).

Important

The Region specified in the configuration file is the location where the Tic-Tac-Toe application creates the Games table in DynamoDB. You must create the Elastic Beanstalk application discussed in the next section in the same Region.

Note

When you create your Elastic Beanstalk application, you request to launch an environment where you can choose the environment type. To test the Tic-Tac-Toe example application, you can choose the **Single Instance** environment type, skip the following, and go to the next step.

However, the **Load balancing, autoscaling** environment type provides a highly available and scalable environment, something you should consider when you create and deploy other applications. If you choose this environment type, you also need to generate a UUID and add it to the configuration file, as shown following.

```
[dynamodb]
region=us-west-2
endpoint=dynamodb.us-west-2.amazonaws.com
[flask]
secret_key= 284e784d-1a25-4a19-92bf-8eeb7a9example
```

In client-server communication, when the server sends a response, for security's sake the server sends a signed cookie that the client sends back to the server in the next request. When there is only one server, the server can locally generate an encryption key when it starts. When there are many servers, they all need to know the same encryption key; otherwise, they won't be able to read cookies set by the peer servers. By adding `secret_key` to the configuration file, you tell all servers to use this encryption key.

3. Zip the content of the root folder of the application (which includes the `beanstalk.config` file)—for example, `TicTacToe.zip`.
4. Upload the `.zip` file to an Amazon Simple Storage Service (Amazon S3) bucket. In the next section, you provide this `.zip` file to Elastic Beanstalk to upload on the server or servers.

For instructions on how to upload to an Amazon S3 bucket, see [Create a bucket](#) and [Add an object to a bucket](#) in the *Amazon Simple Storage Service User Guide*.

3.4: Set up the AWS Elastic Beanstalk environment

In this step, you create an Elastic Beanstalk application, which is a collection of components including environments. For this example, you launch one Amazon EC2 instance to deploy and run your Tic-Tac-Toe application.

1. Enter the following custom URL to set up an Elastic Beanstalk console to set up the environment.

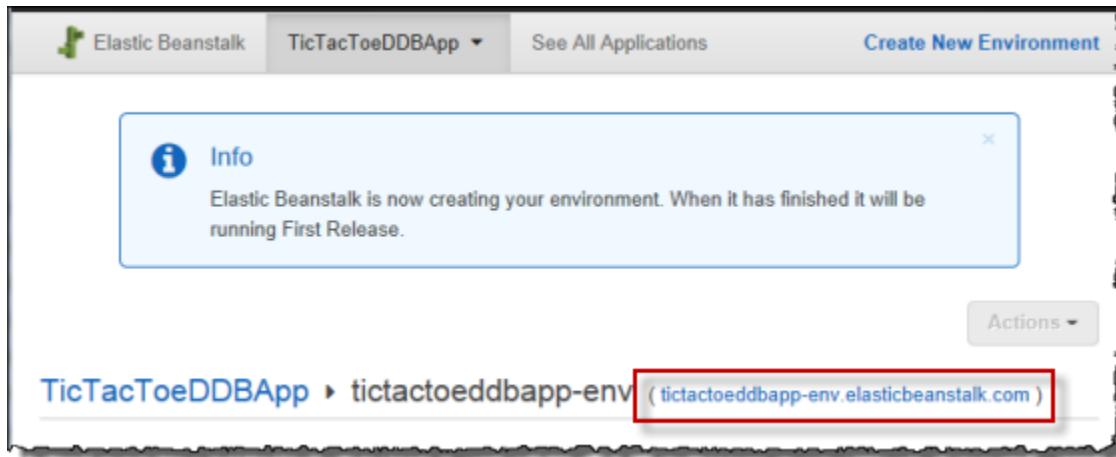
```
https://console.aws.amazon.com/elasticbeanstalk/?region=<AWS-Region>#/newApplication  
?applicationName=TicTacToe<your-name>  
&solutionStackName=Python  
&sourceBundleUrl=https://s3.amazonaws.com/<bucket-name>/TicTacToe.zip  
&environmentType=SingleInstance  
&instanceType=t1.micro
```

For more information about custom URLs, see [Constructing a Launch Now URL](#) in the *AWS Elastic Beanstalk Developer Guide*. For the URL, note the following:

- You must provide an AWS Region name (the same as the one you provided in the configuration file), an Amazon S3 bucket name, and the object name.
- For testing, the URL requests the **SingleInstance** environment type, and **t1.micro** as the instance type.
- The application name must be unique. Thus, in the preceding URL, we suggest you prepend your name to the `applicationName`.

Doing this opens the Elastic Beanstalk console. In some cases, you might need to sign in.

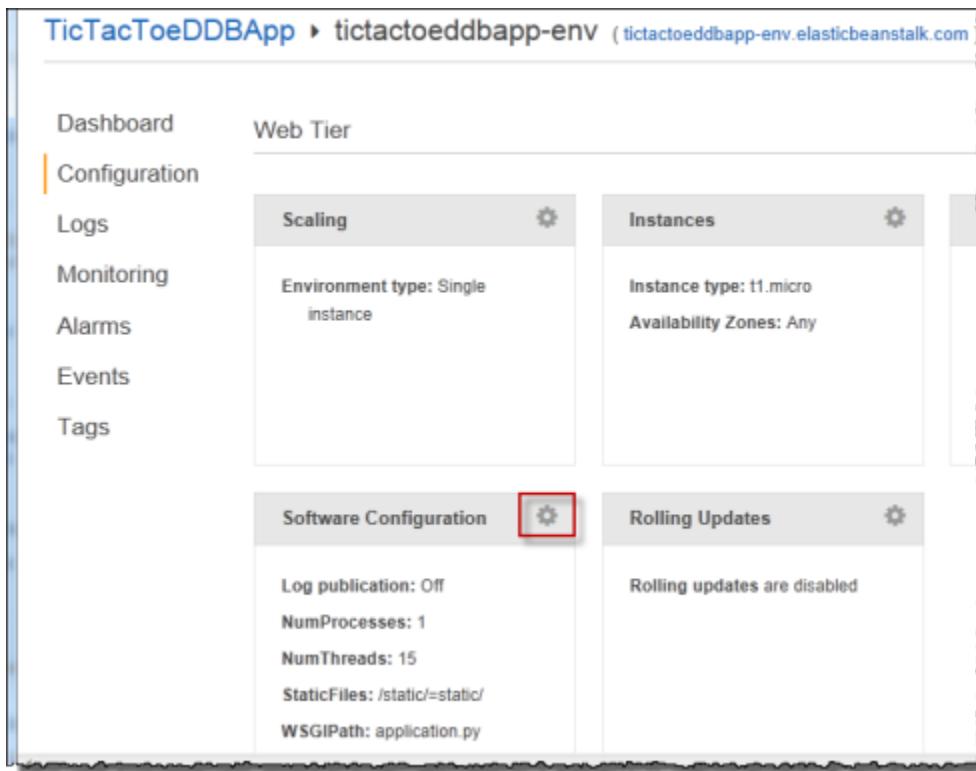
2. In the Elastic Beanstalk console, choose **Review and Launch**, and then choose **Launch**.
3. Note the URL for future reference. This URL opens your Tic-Tac-Toe application home page.



4. Configure the Tic-Tac-Toe application so it knows the location of the configuration file.

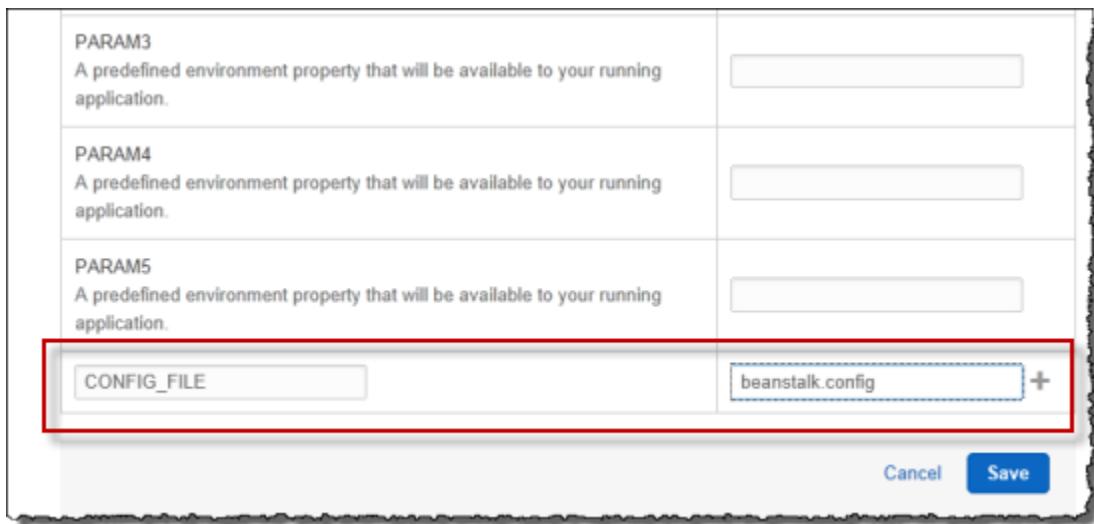
After Elastic Beanstalk creates the application, choose **Configuration**.

- Choose the gear icon next to **Software Configuration**, as shown in the following screenshot.



- At the end of the **Environment Properties** section, enter **CONFIG_FILE** and its value **beanstalk.config**, and then choose **Save**.

It might take a few minutes for this environment update to complete.

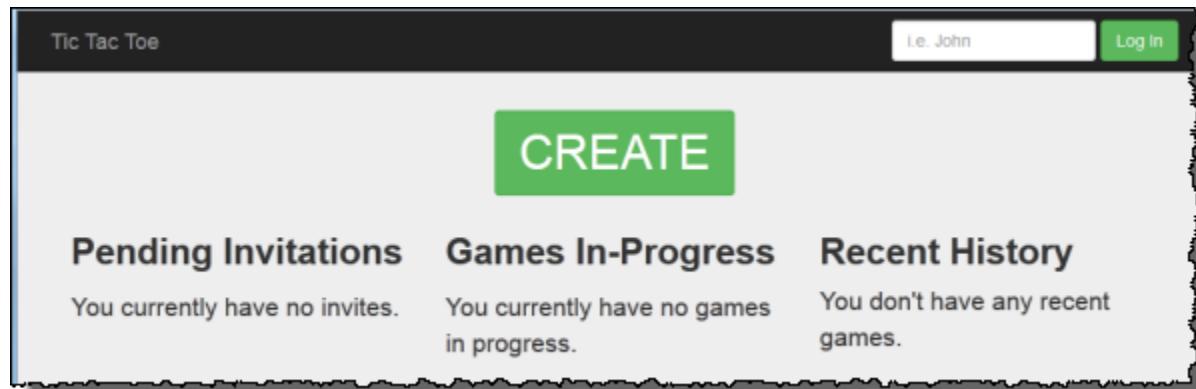


After the update completes, you can play the game.

5. In the browser, enter the URL you copied in the previous step, as shown in the following example.

`http://<app-name>.elasticbeanstalk.com`

Doing this opens the application home page.



6. Log in as testuser1, and choose **CREATE** to start a new tic-tac-toe game.
7. Enter **testuser2** in the **Choose an Opponent** box.



8. Open another browser window.

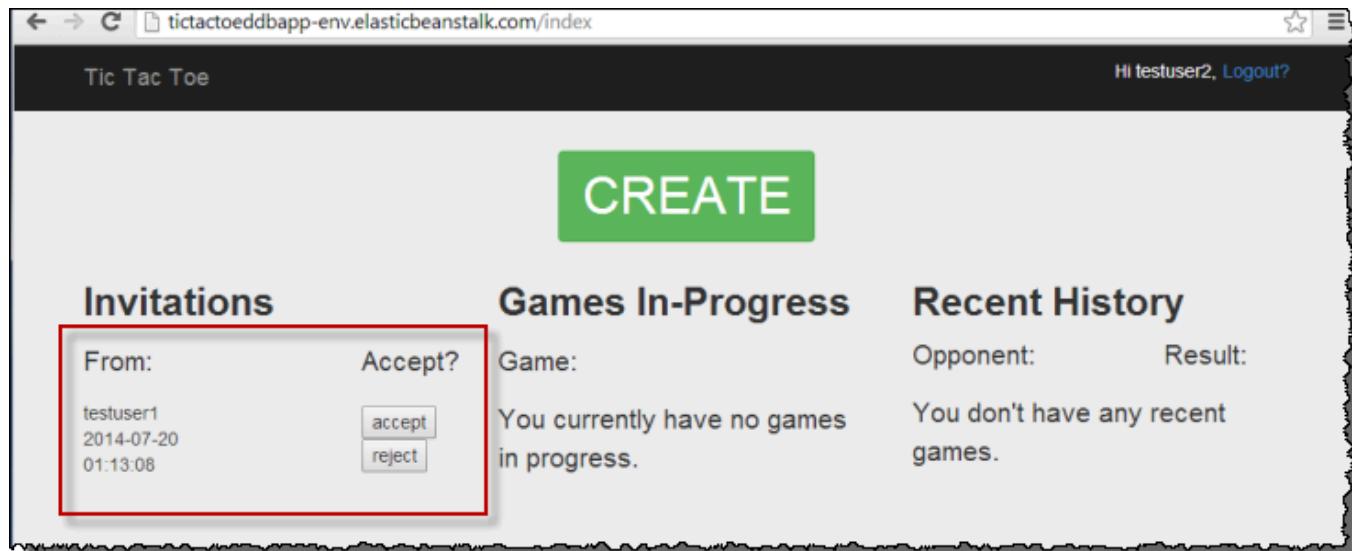
Make sure that you clear all cookies in your browser window so you won't be logged in as same user.

9. Enter the same URL to open the application home page, as shown in the following example.

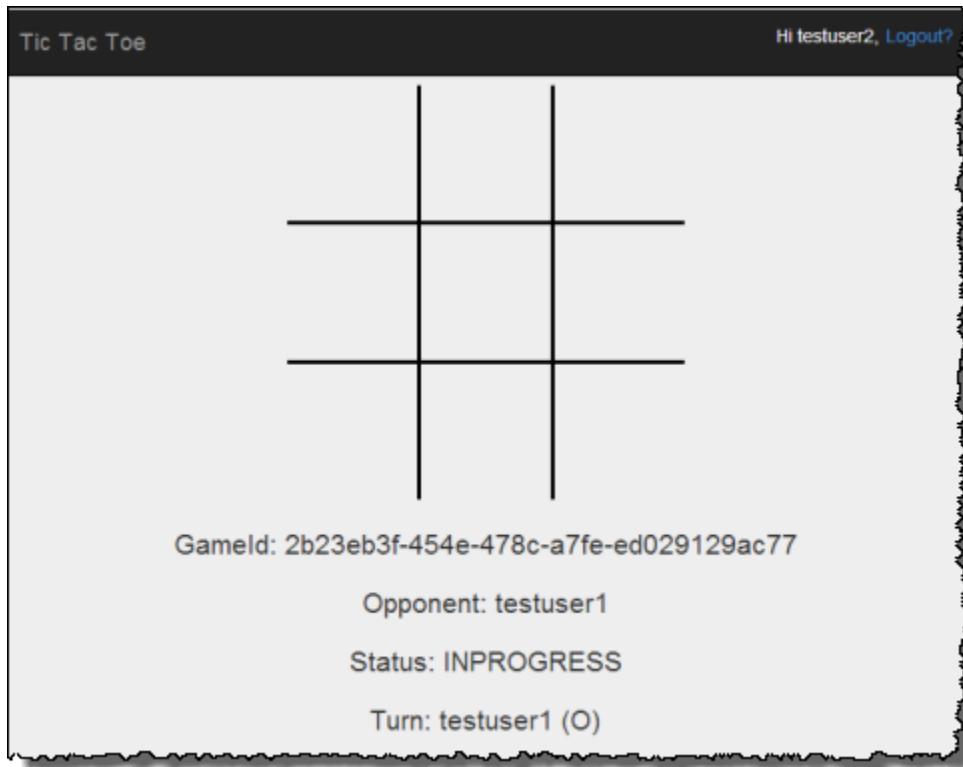
`http://<env-name>.elasticbeanstalk.com`

10. Log in as testuser2.

11. For the invitation from testuser1 in the list of pending invitations, choose **accept**.



12. Now the game page appears.



Both testuser1 and testuser2 can play the game. For each move, the application saves the move in the corresponding item in the Games table.

Step 4: Clean up resources

Now you have completed the Tic-Tac-Toe application deployment and testing. The application covers end-to-end web application development on Amazon DynamoDB, except for user authentication. The application uses the login information on the home page only to add a player name when creating a game. In a production application, you would add the necessary code to perform user login and authentication.

If you are done testing, you can remove the resources you created to test the Tic-Tac-Toe application to avoid incurring any charges.

To remove the resources you created

1. Remove the Games table that you created in DynamoDB.
2. Terminate the Elastic Beanstalk environment to free up the Amazon EC2 instances.
3. Delete the IAM role that you created.

4. Remove the object that you created in Amazon S3.

Exporting and importing DynamoDB data using AWS Data Pipeline

You can use AWS Data Pipeline to export data from a DynamoDB table to a file in an Amazon S3 bucket. You can also use the console to import data from Amazon S3 into a DynamoDB table, in the same AWS region or in a different region.

 **Note**

DynamoDB Console now natively supports importing from Amazon S3 and exporting to Amazon S3. These flows are not compatible with AWS Data Pipeline import flow. For more information see [Import from Amazon S3](#), [Export from Amazon S3](#), and the blog post [Export Amazon DynamoDB table data to your data lake in Amazon S3](#).

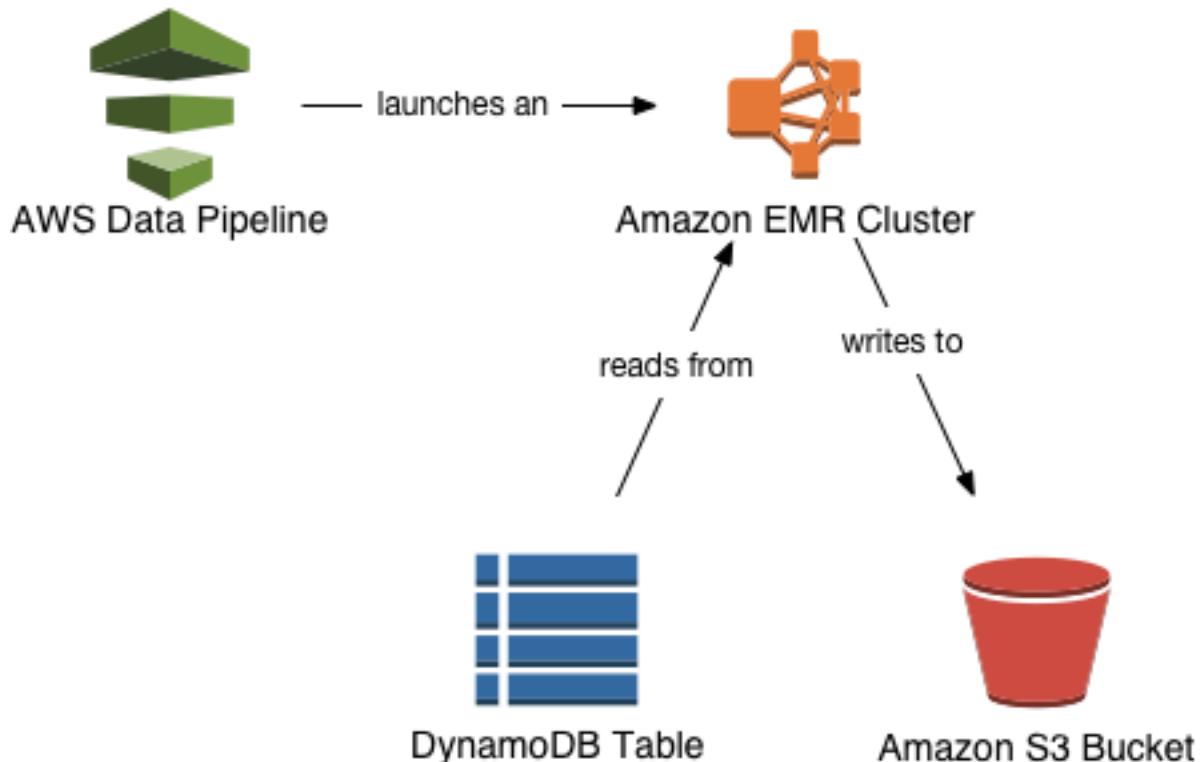
The ability to export and import data is useful in many scenarios. For example, suppose you want to maintain a baseline set of data, for testing purposes. You could put the baseline data into a DynamoDB table and export it to Amazon S3. Then, after you run an application that modifies the test data, you could "reset" the data set by importing the baseline from Amazon S3 back into the DynamoDB table. Another example involves accidental deletion of data, or even an accidental DeleteTable operation. In these cases, you could restore the data from a previous export file in Amazon S3. You can even copy data from a DynamoDB table in one AWS region, store the data in Amazon S3, and then import the data from Amazon S3 to an identical DynamoDB table in a second region. Applications in the second region could then access their nearest DynamoDB endpoint and work with their own copy of the data, with reduced network latency.

 **Important**

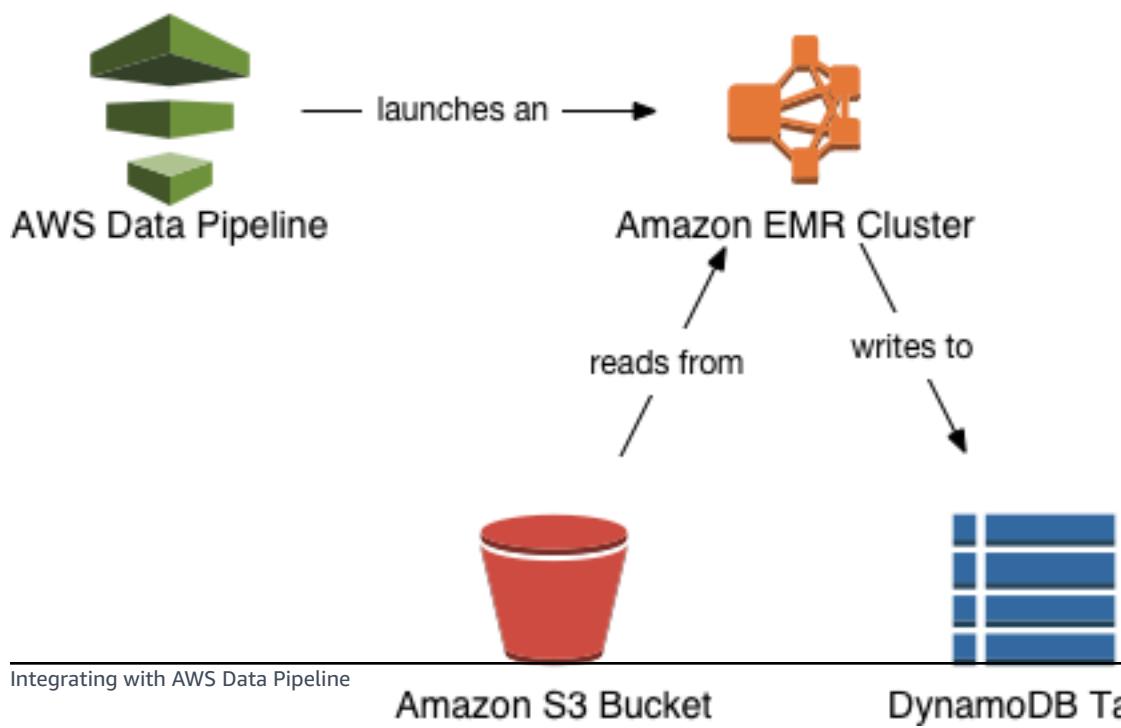
DynamoDB Backup and Restore is a fully managed feature. You can back up tables from a few megabytes to hundreds of terabytes of data, with no impact on the performance and availability of your production applications. You can restore your table with a single click in the AWS Management Console or a single API call. We highly recommend that you use DynamoDB's native backup and restore feature instead of using AWS Data Pipeline. For more information, see [Using On-Demand backup and restore for DynamoDB](#).

The following diagram shows an overview of exporting and importing DynamoDB data using AWS Data Pipeline.

Exporting Data from DynamoDB to Amazon S3



Importing Data from Amazon S3 to DynamoDB



To export a DynamoDB table, you use the AWS Data Pipeline console to create a new pipeline. The pipeline launches an Amazon EMR cluster to perform the actual export. Amazon EMR reads the data from DynamoDB, and writes the data to an export file in an Amazon S3 bucket.

The process is similar for an import, except that the data is read from the Amazon S3 bucket and written to the DynamoDB table.

Important

When you export or import DynamoDB data, you will incur additional costs for the underlying AWS services that are used:

- **AWS Data Pipeline**— manages the import/export workflow for you.
- **Amazon S3**— contains the data that you export from DynamoDB, or import into DynamoDB.
- **Amazon EMR**— runs a managed Hadoop cluster to perform reads and writes between DynamoDB to Amazon S3. The cluster configuration is one `m3.xlarge` instance leader node and one `m3.xlarge` instance core node.

For more information see [AWS Data Pipeline pricing](#), [Amazon EMR pricing](#), and [Amazon S3 pricing](#).

Prerequisites to export and import data

When you use AWS Data Pipeline for exporting and importing data, you must specify the actions that the pipeline is allowed to perform, and which resources the pipeline can consume. The permitted actions and resources are defined using AWS Identity and Access Management (IAM) roles.

You can also control access by creating IAM policies and attaching them to users, roles or groups. These policies let you specify which users are allowed to import and export your DynamoDB data.

Important

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	<p>Following the instructions for the interface that you want to use.</p> <ul style="list-style-type: none">For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the <i>AWS Command Line Interface User Guide</i>.For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the <i>AWS SDKs and Tools Reference Guide</i>.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the <i>IAM User Guide</i> .

Which user needs programmatic access?	To	By
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. <ul style="list-style-type: none">For the AWS CLI, see Authenticating using IAM user credentials in the <i>AWS Command Line Interface User Guide</i>.For AWS SDKs and tools, see Authenticate using long-term credentials in the <i>AWS SDKs and Tools Reference Guide</i>.For AWS APIs, see Managing access keys for IAM users in the <i>IAM User Guide</i>.

Creating IAM roles for AWS Data Pipeline

In order to use AWS Data Pipeline, the following IAM roles must be present in your AWS account:

- ***DataPipelineDefaultRole*** — the actions that your pipeline can take on your behalf.
- ***DataPipelineDefaultResourceRole*** — the AWS resources that the pipeline will provision on your behalf. For exporting and importing DynamoDB data, these resources include an Amazon EMR cluster and the Amazon EC2 instances associated with that cluster.

If you've never used AWS Data Pipeline before, you'll need to create *DataPipelineDefaultRole* and *DataPipelineDefaultResourceRole* yourself. Once you've created these roles, you can use them any time to export or import DynamoDB data.

Note

If you have previously used the AWS Data Pipeline console to create a pipeline, then *DataPipelineDefaultRole* and *DataPipelineDefaultResourceRole* were created for you at that time. No further action is required; you can skip this section and begin creating pipelines using the DynamoDB console. For more information, see [Exporting data from DynamoDB to Amazon S3](#) and [Importing data from Amazon S3 to DynamoDB](#).

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. From the IAM Console Dashboard, click **Roles**.
3. Click **Create Role** and do the following:
 - a. In the **AWS Service** trusted entity, choose **Data Pipeline**.
 - b. In the **Select your use case** panel, choose **Data Pipeline** and then choose **Next:Permissions**.
 - c. Notice that the **AWSDataPipelineRole** policy is automatically attached. Choose **Next:Review**.
 - d. In the **Role name** field, type **DataPipelineDefaultRole** as the role name and choose **Create role**.
4. Click **Create Role** and do the following:
 - a. In the **AWS Service** trusted entity, choose **Data Pipeline**.
 - b. In the **Select your use case** panel, choose **EC2 Role for Data Pipeline** and then choose **Next:Permissions**.
 - c. Notice that the **AmazonEC2RoleForDataPipelineRole** policy is automatically attached. Choose **Next:Review**.
 - d. In the **Role name** field, type **DataPipelineDefaultResourceRole** as the role name and choose **Create role**.

Now that you have created these roles, you can begin creating pipelines using the DynamoDB console. For more information, see [Exporting data from DynamoDB to Amazon S3](#) and [Importing data from Amazon S3 to DynamoDB](#).

Granting users and groups permission to perform export and import tasks using AWS Identity and Access Management

If you want to allow other users, roles or groups to export and import your DynamoDB table data, you can create an IAM policy and attach it to the users or groups that you designate. The policy contains only the necessary permissions for performing these tasks.

Granting full access

The following procedure describes how to attach the AWS managed policies `AmazonDynamoDBFullAccess`, `AWSDataPipeline_FullAccess` and an Amazon EMR inline policy to a user. These managed policies provides full access to AWS Data Pipeline and to DynamoDB resources, and used with the Amazon EMR inline policy, allow the user to perform the actions described in this documentation.

Note

To limit the scope of the suggested permissions, the inline policy above is enforcing the usage of the tag `dynamodbdatapipeline`. If you want to utilize this documentation without this limitation, you can remove the `Condition` section of the suggested policy.

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. From the IAM Console Dashboard, click **Users** and select the user you want to modify.
3. In the **Permissions** tab, click **Add Policy**.
4. In the **Attach permissions** panel, click **Attach existing policies directly**.
5. Select both `AmazonDynamoDBFullAccess` and `AWSDataPipeline_FullAccess` and click **Next:Review**.
6. Click **Add permissions**.
7. Back on **Permissions** tab, click **Add inline policy**.
8. In the **Create a policy** page, click **JSON** tab.
9. Paste the content below.

```
{  
    "Version": "2012-10-17",
```

```
"Statement": [  
    {  
        "Sid": "EMR",  
        "Effect": "Allow",  
        "Action": [  
            "elasticmapreduce:DescribeStep",  
            "elasticmapreduce:DescribeCluster",  
            "elasticmapreduce:RunJobFlow",  
            "elasticmapreduce:TerminateJobFlows"  
        ],  
        "Resource": "*",  
        "Condition": {  
            "Null": {  
                "elasticmapreduce:RequestTag/dynamodbdatipeline": "false"  
            }  
        }  
    }  
]
```

10. Click **Review policy**.

11. Type EMRforDynamoDBDataPipeline on the name field.

12. Click **Create policy**.

 **Note**

You can use a similar procedure to attach this managed policy to a role or group, rather than to a user.

Restricting access to particular DynamoDB tables

If you want to restrict access so that a user can only export or import a subset of your tables, you'll need to create a customized IAM policy document. You can use the process described on [Granting full access](#) as a starting point for your custom policy, and then modify the policy so that a user can only work with the tables that you specify.

For example, suppose that you want to allow a user to export and import only the *Forum*, *Thread*, and *Reply* tables. This procedure describes how to create a custom policy so that a user can work with those tables, but no others.

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. From the IAM Console Dashboard, click **Policies** and then click **Create Policy**.
3. In the **Create Policy** panel, go to **Copy an AWS Managed Policy** and click **Select**.
4. In the **Copy an AWS Managed Policy** panel, go to **AmazonDynamoDBFullAccess** and click **Select**.
5. In the **Review Policy** panel, do the following:
 - a. Review the autogenerated **Policy Name** and **Description**. If you want, you can modify these values.
 - b. In the **Policy Document** text box, edit the policy to restrict access to specific tables. By default, the policy permits all DynamoDB actions on all of your tables:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "cloudwatch:DeleteAlarms",  
                "cloudwatch:DescribeAlarmHistory",  
                "cloudwatch:DescribeAlarms",  
                "cloudwatch:DescribeAlarmsForMetric",  
                "cloudwatch:GetMetricStatistics",  
                "cloudwatch>ListMetrics",  
                "cloudwatch:PutMetricAlarm",  
                "dynamodb:*",  
                "sns>CreateTopic",  
                "sns>DeleteTopic",  
                "sns>ListSubscriptions",  
                "sns>ListSubscriptionsByTopic",  
                "sns>ListTopics",  
                "sns:Subscribe",  
                "sns:Unsubscribe"  
            ],  
            "Effect": "Allow",  
            "Resource": "*",  
            "Sid": "DDBConsole"  
        },  
        ...remainder of document omitted...
```

To restrict the policy, first remove the following line:

```
"dynamodb:*",
```

Next, construct a new Action that allows access to only the *Forum*, *Thread* and *Reply* tables:

```
{  
    "Action": [  
        "dynamodb:*"  
    ],  
    "Effect": "Allow",  
    "Resource": [  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Forum",  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Thread",  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Reply"  
    ]  
},
```

 **Note**

Replace `us-west-2` with the region in which your DynamoDB tables reside.
Replace `123456789012` with your AWS account number.

Finally, add the new Action to the policy document:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Forum",  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Thread",  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Reply"  
            ]  
        }  
    ]  
}
```

```
},
{
    "Action": [
        "cloudwatch:DeleteAlarms",
        "cloudwatch:DescribeAlarmHistory",
        "cloudwatch:DescribeAlarms",
        "cloudwatch:DescribeAlarmsForMetric",
        "cloudwatch:GetMetricStatistics",
        "cloudwatch>ListMetrics",
        "cloudwatch:PutMetricAlarm",
        "sns>CreateTopic",
        "sns>DeleteTopic",
        "sns>ListSubscriptions",
        "sns>ListSubscriptionsByTopic",
        "sns>ListTopics",
        "sns>Subscribe",
        "sns>Unsubscribe"
    ],
    "Effect": "Allow",
    "Resource": "*",
    "Sid": "DDBConsole"
},
...remainder of document omitted...
```

6. When the policy settings are as you want them, click **Create Policy**.

After you have created the policy, you can attach it to a user.

1. From the IAM Console Dashboard, click **Users** and select the user you want to modify.
2. In the **Permissions** tab, click **Attach Policy**.
3. In the **Attach Policy** panel, select the name of your policy and click **Attach Policy**.

Note

You can use a similar procedure to attach your policy to a role or group, rather than to a user.

Exporting data from DynamoDB to Amazon S3

This section describes how to export data from one or more DynamoDB tables to an Amazon S3 bucket. You need to create the Amazon S3 bucket before you can perform the export.

Important

If you have never used AWS Data Pipeline before, you will need to set up two IAM roles before following this procedure. For more information, see [Creating IAM roles for AWS Data Pipeline](#).

1. Sign in to the AWS Management Console and open the AWS Data Pipeline console at <https://console.aws.amazon.com/datapipeline/>.
2. If you do not already have any pipelines in the current AWS region, choose **Get started now**. Otherwise, if you already have at least one pipeline, choose **Create new pipeline**.
3. On the **Create Pipeline** page, do the following:
 - a. In the **Name** field, type a name for your pipeline. For example: MyDynamoDBExportPipeline.
 - b. For the **Source** parameter, select **Build using a template**. From the drop-down template list, choose **Export DynamoDB table to S3**.
 - c. In the **Source DynamoDB table name** field, type the name of the DynamoDB table that you want to export.
 - d. In the **Output S3 Folder** text box, enter an Amazon S3 URI where the export file will be written. For example: s3://mybucket/exports

The format of this URI is `s3://bucketname/folder` where:

- `bucketname` is the name of your Amazon S3 bucket.
 - `folder` is the name of a folder within that bucket. If the folder does not exist, it will be created automatically. If you do not specify a name for the folder, a name will be assigned for it in the form `s3://bucketname/region/tablename`.
- e. In the **S3 location for logs** text box, enter an Amazon S3 URI where the log file for the export will be written. For example: s3://mybucket/logs/

The URI format for **S3 Log Folder** is the same as for **Output S3 Folder**. The URI must resolve to a folder; log files cannot be written to the top level of the S3 bucket.

4. Add a tag with the Key dynamodbdatapipeline and the Value true.
5. When the settings are as you want them, click **Activate**.

Your pipeline will now be created; this process can take several minutes to complete. You can monitor the progress in the AWS Data Pipeline console.

When the export has finished, you can go to the [Amazon S3 console](#) to view your export file. The output file name is an identifier value with no extension, such as this example: ae10f955-fb2f-4790-9b11-fbfea01a871e_000000. The internal format of this file is described at [File Structure](#) in the *AWS Data Pipeline Developer Guide*.

Importing data from Amazon S3 to DynamoDB

This section assumes that you have already exported data from a DynamoDB table, and that the export file has been written to your Amazon S3 bucket. The internal format of this file is described at [File Structure](#) in the *AWS Data Pipeline Developer Guide*. Note that this is the *only* file format that DynamoDB can import using AWS Data Pipeline.

We will use the term *source table* for the original table from which the data was exported, and *destination table* for the table that will receive the imported data. You can import data from an export file in Amazon S3, provided that all of the following are true:

- The destination table already exists. (The import process will not create the table for you.)
- The destination table has the same key schema as the source table.

The destination table does not have to be empty. However, the import process will replace any data items in the table that have the same keys as the items in the export file. For example, suppose you have a *Customer* table with a key of *CustomerId*, and that there are only three items in the table (*CustomerId* 1, 2, and 3). If your export file also contains data items for *CustomerId* 1, 2, and 3, the items in the destination table will be replaced with those from the export file. If the export file also contains a data item for *CustomerId* 4, then that item will be added to the table.

The destination table can be in a different AWS region. For example, suppose you have a *Customer* table in the US West (Oregon) region and export its data to Amazon S3. You could then import that data into an identical *Customer* table in the Europe (Ireland) region. This is referred to as a

cross-region export and import. For a list of AWS regions, go to [Regions and endpoints in the AWS General Reference](#).

Note that the AWS Management Console lets you export multiple source tables at once. However, you can only import one table at a time.

1. Sign in to the AWS Management Console and open the AWS Data Pipeline console at <https://console.aws.amazon.com/datapipeline/>.
2. (Optional) If you want to perform a cross region import, go to the upper right corner of the window and choose the destination region.
3. Choose **Create new pipeline**.
4. On the **Create Pipeline** page, do the following:
 - a. In the **Name** field, type a name for your pipeline. For example: MyDynamoDBImportPipeline.
 - b. For the **Source** parameter, select **Build using a template**. From the drop-down template list, choose **Import DynamoDB backup data from S3**.
 - c. In the **Input S3 Folder** text box, enter an Amazon S3 URI where the export file can be found. For example: s3://mybucket/exports

The format of this URI is `s3://bucketname/folder` where:

- *bucketname* is the name of your Amazon S3 bucket.
- *folder* is the name of the folder that contains the export file.

The import job will expect to find a file at the specified Amazon S3 location. The internal format of the file is described at [Verify data export file in the AWS Data Pipeline Developer Guide](#).

- d. In the **Target DynamoDB table name** field, type the name of the DynamoDB table into which you want to import the data.
- e. In the **S3 location for logs** text box, enter an Amazon S3 URI where the log file for the import will be written. For example: s3://mybucket/logs/

The URI format for **S3 Log Folder** is the same as for **Output S3 Folder**. The URI must resolve to a folder; log files cannot be written to the top level of the S3 bucket.

- f. Add a tag with the Key dynamodbdatapipeline and the Value true.

- When the settings are as you want them, click **Activate**.

Your pipeline will now be created; this process can take several minutes to complete. The import job will begin immediately after the pipeline has been created.

Troubleshooting

This section covers some basic failure modes and troubleshooting for DynamoDB exports.

If an error occurs during an export or import, the pipeline status in the AWS Data Pipeline console will display as **ERROR**. If this happens, click the name of the failed pipeline to go to its detail page. This will show details about all of the steps in the pipeline, and the status of each one. In particular, examine any execution stack traces that you see.

Finally, go to your Amazon S3 bucket and look for any export or import log files that were written there.

The following are some common issues that may cause a pipeline to fail, along with corrective actions. To diagnose your pipeline, compare the errors you have seen with the issues noted below.

- For an import, ensure that the destination table already exists, and the destination table has the same key schema as the source table. These conditions must be met, or the import will fail.
- Ensure that the pipeline has the tag `dynamodbdatipeline`; otherwise, the Amazon EMR API calls will not succeed.
- Ensure that the Amazon S3 bucket you specified has been created, and that you have read and write permissions on it.
- The pipeline might have exceeded its execution timeout. (You set this parameter when you created the pipeline.) For example, you might have set the execution timeout for 1 hour, but the export job might have required more time than this. Try deleting and then re-creating the pipeline, but with a longer execution timeout interval this time.
- Update the manifest file if you restore from a Amazon S3 bucket that is not the original bucket that the export was performed with (contains a copy of the export).
- You might not have the correct permissions for performing an export or import. For more information, see [Prerequisites to export and import data](#).
- You might have reached a resource quota in your AWS account, such as the maximum number of Amazon EC2 instances or the maximum number of AWS Data Pipeline pipelines. For more

information, including how to request increases in these quotas, see [AWS service quotas](#) in the *AWS General Reference*.

 **Note**

For more details on troubleshooting a pipeline, go to [Troubleshooting](#) in the *AWS Data Pipeline Developer Guide*.

Predefined templates for AWS Data Pipeline and DynamoDB

If you would like a deeper understanding of how AWS Data Pipeline works, we recommend that you consult the *AWS Data Pipeline Developer Guide*. This guide contains step-by-step tutorials for creating and working with pipelines; you can use these tutorials as starting points for creating your own pipelines. We recommend that you read the AWS Data Pipeline tutorial, which walks you through the steps required to create an import and export pipeline that you can customize for your requirements. See [Tutorial: Amazon DynamoDB import and export using AWS Data Pipeline](#) in the *AWS Data Pipeline Developer Guide*.

AWS Data Pipeline offers several templates for creating pipelines; the following templates are relevant to DynamoDB.

Exporting data between DynamoDB and Amazon S3

 **Note**

DynamoDB Console now supports its own Export to Amazon S3 flow, however it is not compatible with AWS Data Pipeline import flow. For more information, see [DynamoDB data export to Amazon S3: how it works](#) and the blog post [Export Amazon DynamoDB table data to your data lake in Amazon S3, no code writing required](#).

The AWS Data Pipeline console provides two predefined templates for exporting data between DynamoDB and Amazon S3. For more information about these templates, see the following sections of the *AWS Data Pipeline Developer Guide*:

- [Export DynamoDB to Amazon S3](#)
- [Export Amazon S3 to DynamoDB](#)

Amazon DynamoDB Storage Backend for Titan

The DynamoDB Storage Backend for Titan project has been superseded by the Amazon DynamoDB Storage Backend for JanusGraph, which is available on [GitHub](#).

For up-to-date instructions on the DynamoDB Storage Backend for JanusGraph, see the [README.md](#) file.

Reserved words in DynamoDB

The following keywords are reserved for use by DynamoDB. Do not use any of these words as attribute names in expressions. This list is not case-sensitive.

If you need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word, you can define an expression attribute name to use in the place of the reserved word. For more information, see [Expression attribute names in DynamoDB](#).

```
ABORT
ABSOLUTE
ACTION
ADD
AFTER
AGENT
AGGREGATE
ALL
ALLOCATE
ALTER
ANALYZE
AND
ANY
ARCHIVE
ARE
ARRAY
AS
ASC
ASCII
ASENSITIVE
ASSERTION
ASYMMETRIC
AT
ATOMIC
```

ATTACH
ATTRIBUTE
AUTH
AUTHORIZATION
AUTHORIZE
AUTO
AVG
BACK
BACKUP
BASE
BATCH
BEFORE
BEGIN
BETWEEN
BIGINT
BINARY
BIT
BLOB
BLOCK
BOOLEAN
BOTH
BREADTH
BUCKET
BULK
BY
BYTE
CALL
CALLED
CALLING
CAPACITY
CASCADE
CASCADED
CASE
CAST
CATALOG
CHAR
CHARACTER
CHECK
CLASS
CLOB
CLOSE
CLUSTER
CLUSTERED
CLUSTERING

CLUSTERS
COALESCE
COLLATE
COLLATION
COLLECTION
COLUMN
COLUMNS
COMBINE
COMMENT
COMMIT
COMPACT
COMPILE
COMPRESS
CONDITION
CONFLICT
CONNECT
CONNECTION
CONSISTENCY
CONSISTENT
CONSTRAINT
CONSTRAINTS
CONSTRUCTOR
CONSUMED
CONTINUE
CONVERT
COPY
CORRESPONDING
COUNT
COUNTER
CREATE
CROSS
CUBE
CURRENT
CURSOR
CYCLE
DATA
DATABASE
DATE
DATETIME
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE

DEFAULT
DEFERRABLE
DEFERRED
DEFINE
DEFINED
DEFINITION
DELETE
DELIMITED
DEPTH
DEREF
DESC
DESCRIBE
DESCRIPTOR
DETACH
DETERMINISTIC
DIAGNOSTICS
DIRECTORIES
DISABLE
DISCONNECT
DISTINCT
DISTRIBUTE
DO
DOMAIN
DOUBLE
DROP
DUMP
DURATION
DYNAMIC
EACH
ELEMENT
ELSE
ELSEIF
EMPTY
ENABLE
END
EQUAL
EQUALS
ERROR
ESCAPE
ESCAPED
EVAL
EVALUATE
EXCEEDED
EXCEPT

EXCEPTION
EXCEPTIONS
EXCLUSIVE
EXEC
EXECUTE
EXISTS
EXIT
EXPLAIN
EXPLODE
EXPORT
EXPRESSION
EXTENDED
EXTERNAL
EXTRACT
FAIL
FALSE
FAMILY
FETCH
FIELDS
FILE
FILTER
FILTERING
FINAL
FINISH
FIRST
FIXED
FLATTEN
FLOAT
FOR
FORCE
FOREIGN
FORMAT
FORWARD
FOUND
FREE
FROM
FULL
FUNCTION
FUNCTIONS
GENERAL
GENERATE
GET
GLOB
GLOBAL

GO
GOTO
GRANT
GREATER
GROUP
GROUPING
HANDLER
HASH
HAVE
HAVING
HEAP
HIDDEN
HOLD
HOUR
IDENTIFIED
IDENTITY
IF
IGNORE
IMMEDIATE
IMPORT
IN
INCLUDING
INCLUSIVE
INCREMENT
INCREMENTAL
INDEX
INDEXED
INDEXES
INDICATOR
INFINITE
INITIALLY
INLINE
INNER
INNTER
INOUT
INPUT
INSENSITIVE
INSERT
INSTEAD
INT
INTEGER
INTERSECT
INTERVAL
INTO

INVALIDATE
IS
ISOLATION
ITEM
ITEMS
ITERATE
JOIN
KEY
KEYS
LAG
LANGUAGE
LARGE
LAST
LATERAL
LEAD
LEADING
LEAVE
LEFT
LENGTH
LESS
LEVEL
LIKE
LIMIT
LIMITED
LINES
LIST
LOAD
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOCATION
LOCATOR
LOCK
LOCKS
LOG
LOGED
LONG
LOOP
LOWER
MAP
MATCH
MATERIALIZED
MAX
MAXLEN

MEMBER
MERGE
METHOD
METRICS
MIN
MINUS
MINUTE
MISSING
MOD
MODE
MODIFIES
MODIFY
MODULE
MONTH
MULTI
MULTISET
NAME
NAMES
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NEXT
NO
NONE
NOT
NULL
NULLIF
NUMBER
NUMERIC
OBJECT
OF
OFFLINE
OFFSET
OLD
ON
ONLINE
ONLY
OPAQUE
OPEN
OPERATOR
OPTION
OR

ORDER
ORDINALITY
OTHER
OTHERS
OUT
OUTER
OUTPUT
OVER
OVERLAPS
OVERRIDE
OWNER
PAD
PARALLEL
PARAMETER
PARAMETERS
PARTIAL
PARTITION
PARTITIONED
PARTITIONS
PATH
PERCENT
PERCENTILE
PERMISSION
PERMISSIONS
PIPE
PIPELINED
PLAN
POOL
POSITION
PRECISION
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVATE
PRIVILEGES
PROCEDURE
PROCESSED
PROJECT
PROJECTION
PROPERTY
PROVISIONING
PUBLIC
PUT

QUERY
QUIT
QUORUM
RAISE
RANDOM
RANGE
RANK
RAW
READ
READS
REAL
REBUILD
RECORD
RECURSIVE
REDUCE
REF
REFERENCE
REFERENCES
REFERENCING
REGEXP
REGION
REINDEX
RELATIVE
RELEASE
REMAINDER
RENAME
REPEAT
REPLACE
REQUEST
RESET
RESIGNAL
RESOURCE
RESPONSE
RESTORE
RESTRICT
RESULT
RETURN
RETURNING
RETURNS
REVERSE
REVOKE
RIGHT
ROLE
ROLES

ROLLBACK
ROLLUP
ROUTINE
ROW
ROWS
RULE
RULES
SAMPLE
SATISFIES
SAVE
SAVEPOINT
SCAN
SCHEMA
SCOPE
SCROLL
SEARCH
SECOND
SECTION
SEGMENT
SEGMENTS
SELECT
SELF
SEMI
SENSITIVE
SEPARATE
SEQUENCE
SERIALIZABLE
SESSION
SET
SETS
SHARD
SHARE
SHARED
SHORT
SHOW
SIGNAL
SIMILAR
SIZE
SKEWED
SMALLINT
SNAPSHOT
SOME
SOURCE
SPACE

SPACES
SPARSE
SPECIFIC
SPECIFICTYPE
SPLIT
SQL
SQLCODE
SQLERROR
SQLEXCEPTION
SQLSTATE
SQLWARNING
START
STATE
STATIC
STATUS
STORAGE
STORE
STORED
STREAM
STRING
STRUCT
STYLE
SUB
SUBMULTISET
SUBPARTITION
SUBSTRING
SUBTYPE
SUM
SUPER
SYMMETRIC
SYNONYM
SYSTEM
TABLE
TABLESAMPLE
TEMP
TEMPORARY
TERMINATED
TEXT
THAN
THEN
THROUGHPUT
TIME
TIMESTAMP
TIMEZONE

TINYINT
TO
TOKEN
TOTAL
TOUCH
TRAILING
TRANSACTION
TRANSFORM
TRANSLATE
TRANSLATION
TREAT
TRIGGER
TRIM
TRUE
TRUNCATE
TTL
TUPLE
TYPE
UNDER
UNDO
UNION
UNIQUE
UNIT
UNKNOWN
UNLOGGED
UNNEST
UNPROCESSED
UNSIGNED
UNTIL
UPDATE
UPPER
URL
USAGE
USE
USER
USERS
USING
UUID
VACUUM
VALUE
VALUED
VALUES
VARCHAR
VARIABLE

VARIANCE
VARINT
VARYING
VIEW
VIEWS
VIRTUAL
VOID
WAIT
WHEN
WHENEVER
WHERE
WHILE
WINDOW
WITH
WITHIN
WITHOUT
WORK
WRAPPED
WRITE
YEAR
ZONE

Legacy conditional parameters

This section compares the legacy conditional parameters with expression parameters in DynamoDB.

 **Important**

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). Additionally, DynamoDB does not allow mixing legacy conditional parameters and expression parameters in a single call. For example, calling the Query operation with AttributesToGet and ConditionExpression will result in an error.

The following table shows the DynamoDB API operations that still support these legacy parameters, and which expression parameter to use instead. This table can be helpful if you are considering updating your applications so that they use expression parameters instead.

If you use this API operation ... s...	With these legacy parameter s...	Use this expression parameter instead
BatchGetItem	AttributesToGet	ProjectionExpression
DeleteItem	Expected	ConditionExpression
GetItem	AttributesToGet	ProjectionExpression
PutItem	Expected	ConditionExpression
Query	AttributesToGet	ProjectionExpression
	KeyConditions	KeyConditionExpression
	QueryFilter	FilterExpression
Scan	AttributesToGet	ProjectionExpression
	ScanFilter	FilterExpression
UpdateItem	AttributeUpdates	UpdateExpression
	Expected	ConditionExpression

The following sections provide more information about legacy conditional parameters.

Topics

- [AttributesToGet \(legacy\)](#)
- [AttributeUpdates \(legacy\)](#)
- [ConditionalOperator \(legacy\)](#)
- [Expected \(legacy\)](#)
- [KeyConditions \(legacy\)](#)
- [QueryFilter \(legacy\)](#)
- [ScanFilter \(legacy\)](#)
- [Writing conditions with legacy parameters](#)

AttributesToGet (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). For specific information on the new parameter replacing this one, [use *ProjectionExpression* instead..](#)

The legacy conditional parameter `AttributesToGet` is an array of one or more attributes to retrieve from DynamoDB. If no attribute names are provided, then all attributes will be returned. If any of the requested attributes are not found, they will not appear in the result.

`AttributesToGet` allows you to retrieve attributes of type List or Map; however, it cannot retrieve individual elements within a List or a Map.

Note that `AttributesToGet` has no effect on provisioned throughput consumption. DynamoDB determines capacity units consumed based on item size, not on the amount of data that is returned to an application.

Use *ProjectionExpression* instead – Example

Suppose you wanted to retrieve an item from the *Music* table, but that you only wanted to return some of the attributes. You could use a `GetItem` request with an `AttributesToGet` parameter, as in this AWS CLI example:

```
aws dynamodb get-item \
--table-name Music \
--attributes-to-get '["Artist", "Genre"]' \
--key '{
    "Artist": {"S":"No One You Know"}, 
    "SongTitle": {"S":"Call Me Today"}
}'
```

You can use a *ProjectionExpression* instead:

```
aws dynamodb get-item \
--table-name Music \
--projection-expression "Artist, Genre" \
```

```
--key '{  
    "Artist": {"S":"No One You Know"},  
    "SongTitle": {"S":"Call Me Today"}  
}'
```

AttributeUpdates (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). For specific information on the new parameter replacing this one, [use *UpdateExpression* instead..](#)

In an `UpdateItem` operation, the legacy conditional parameter `AttributeUpdates` contains the names of attributes to be modified, the action to perform on each, and the new value for each. If you are updating an attribute that is an index key attribute for any indexes on that table, the attribute type must match the index key type defined in the `AttributesDefinition` of the table description. You can use `UpdateItem` to update any non-key attributes.

Attribute values cannot be null. String and Binary type attributes must have lengths greater than zero. Set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException` exception.

Each `AttributeUpdates` element consists of an attribute name to modify, along with the following:

- **Value** - The new value, if applicable, for this attribute.
- **Action** - A value that specifies how to perform the update. This action is only valid for an existing attribute whose data type is Number or is a set; do not use ADD for other data types.

If an item with the specified primary key is found in the table, the following values perform the following actions:

- **PUT** - Adds the specified attribute to the item. If the attribute already exists, it is replaced by the new value.
- **DELETE** - Removes the attribute and its value, if no value is specified for DELETE. The data type of the specified value must match the existing value's data type.

If a set of values is specified, then those values are subtracted from the old set. For example, if the attribute value was the set [a, b, c] and the DELETE action specifies [a, c], then the final attribute value is [b]. Specifying an empty set is an error.

- ADD - Adds the specified value to the item, if the attribute does not already exist. If the attribute does exist, then the behavior of ADD depends on the data type of the attribute:
 - If the existing attribute is a number, and if Value is also a number, then Value is mathematically added to the existing attribute. If Value is a negative number, then it is subtracted from the existing attribute.

 **Note**

If you use ADD to increment or decrement a number value for an item that doesn't exist before the update, DynamoDB uses 0 as the initial value.

Similarly, if you use ADD for an existing item to increment or decrement an attribute value that doesn't exist before the update, DynamoDB uses 0 as the initial value. For example, suppose that the item you want to update doesn't have an attribute named *itemcount*, but you decide to ADD the number 3 to this attribute anyway. DynamoDB will create the *itemcount* attribute, set its initial value to 0, and finally add 3 to it.

The result will be a new *itemcount* attribute, with a value of 3.

- If the existing data type is a set, and if Value is also a set, then Value is appended to the existing set. For example, if the attribute value is the set [1, 2], and the ADD action specified [3], then the final attribute value is [1, 2, 3]. An error occurs if an ADD action is specified for a set attribute and the attribute type specified does not match the existing set type.

Both sets must have the same primitive data type. For example, if the existing data type is a set of strings, Value must also be a set of strings.

If no item with the specified key is found in the table, the following values perform the following actions:

- PUT - Causes DynamoDB to create a new item with the specified primary key, and then adds the attribute.
- DELETE - Nothing happens, because attributes cannot be deleted from a nonexistent item. The operation succeeds, but DynamoDB does not create a new item.
- ADD - Causes DynamoDB to create an item with the supplied primary key and number (or set of numbers) for the attribute value. The only data types allowed are Number and Number Set.

If you provide any attributes that are part of an index key, then the data types for those attributes must match those of the schema in the table's attribute definition.

Use `UpdateExpression` instead – Example

Suppose you wanted to modify an item in the *Music* table. You could use an `UpdateItem` request with an `AttributeUpdates` parameter, as in this AWS CLI example:

```
aws dynamodb update-item \  
  --table-name Music \  
  --key '{  
    "SongTitle": {"S":"Call Me Today"},  
    "Artist": {"S":"No One You Know"}  
' \  
  --attribute-updates '{  
    "Genre": {  
      "Action": "PUT",  
      "Value": {"S":"Rock"}  
    }  
'
```

You can use a `UpdateExpression` instead:

```
aws dynamodb update-item \  
  --table-name Music \  
  --key '{  
    "SongTitle": {"S":"Call Me Today"},  
    "Artist": {"S":"No One You Know"}  
' \  
  --update-expression 'SET Genre = :g' \  
  --expression-attribute-values '{  
    ":g": {"S":"Rock"}  
'
```

For more information about updating attributes, see [Update an item in a DynamoDB table](#).

ConditionalOperator (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#).

The legacy conditional parameter `ConditionalOperator` is a logical operator used to apply to the conditions in a `Expected`, `QueryFilter` or `ScanFilter` map:

- AND - If all of the conditions evaluate to true, then the entire map evaluates to true.
- OR - If at least one of the conditions evaluates to true, then the entire map evaluates to true.

If you omit `ConditionalOperator`, then AND is the default.

The operation will succeed only if the entire map evaluates to true.

Note

This parameter does not support attributes of type List or Map.

Expected (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). For specific information on the new parameter replacing this one, see [*ConditionExpression instead..*](#)

The legacy conditional parameter `Expected` is a conditional block for an `UpdateItem` operation. `Expected` is a map of attribute/condition pairs. Each element of the map consists of an attribute name, a comparison operator, and one or more values. DynamoDB compares the attribute with the value(s) you supplied, using the comparison operator. For each `Expected` element, the result of the evaluation is either true or false.

If you specify more than one element in the Expected map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the ConditionalOperator parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

If the Expected map evaluates to true, then the conditional operation succeeds; otherwise, it fails.

Expected contains the following:

- **AttributeValueList** - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the ComparisonOperator being used.

For type Number, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on Unicode with UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

For type Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

- **ComparisonOperator** - A comparator for evaluating attributes in the AttributeValueList. When performing the comparison, DynamoDB uses strongly consistent reads.

The following comparison operators are available:

EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS |
BEGINS_WITH | IN | BETWEEN

The following are descriptions of each comparison operator.

- EQ : Equal. EQ is supported for all data types, including lists and maps.

AttributeValueList can contain only one AttributeValue element of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, {"S": "6"} does not equal {"N": "6"}. Also, {"N": "6"} does not equal {"NS": ["6", "2", "1"]}.

- NE : Not equal. NE is supported for all data types, including lists and maps.

AttributeValueList can contain only one AttributeValue of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains an AttributeValue of a different type than the one provided in the request, the value does not match. For example,

`{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.

- LE : Less than or equal.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- LT : Less than.

AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- GE : Greater than or equal.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- GT : Greater than.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- NOT_NULL : The attribute exists. NOT_NULL is supported for all data types, including lists and maps.

 **Note**

This operator tests for the existence of an attribute, not its data type. If the data type of attribute "a" is null, and you evaluate it using NOT_NULL, the result is a Boolean

true. This result is because the attribute "a" exists; its data type is not relevant to the NOT_NULL comparison operator.

- **NULL** : The attribute does not exist. NULL is supported for all data types, including lists and maps.

 **Note**

This operator tests for the nonexistence of an attribute, not its data type. If the data type of attribute "a" is null, and you evaluate it using NULL, the result is a Boolean false. This is because the attribute "a" exists; its data type is not relevant to the NULL comparison operator.

- **CONTAINS** : Checks for a subsequence, or value in a set.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If the target attribute of the comparison is of type String, then the operator checks for a substring match. If the target attribute of the comparison is of type Binary, then the operator looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operator evaluates to true if it finds an exact match with any member of the set.

CONTAINS is supported for lists: When evaluating "a CONTAINS b", "a" can be a list; however, "b" cannot be a set, a map, or a list.

- **NOT_CONTAINS** : Checks for absence of a subsequence, or absence of a value in a set.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If the target attribute of the comparison is a String, then the operator checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operator checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operator evaluates to true if it does not find an exact match with any member of the set.

NOT_CONTAINS is supported for lists: When evaluating "a NOT CONTAINS b", "a" can be a list; however, "b" cannot be a set, a map, or a list.

- **BEGINS_WITH** : Checks for a prefix.

AttributeValueList can contain only one AttributeValue of type String or Binary (not a Number or a set type). The target attribute of the comparison must be of type String or Binary (not a Number or a set type).

- IN : Checks for matching elements within two sets.

AttributeValueList can contain one or more AttributeValue elements of type String, Number, or Binary (not a set type). These attributes are compared against an existing set type attribute of an item. If any elements of the input set are present in the item attribute, the expression evaluates to true.

- BETWEEN : Greater than or equal to the first value, and less than or equal to the second value.

AttributeValueList must contain two AttributeValue elements of the same type, either String, Number, or Binary (not a set type). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, {"S": "6"} does not compare to {"N": "6"}. Also, {"N": "6"} does not compare to {"NS": ["6", "2", "1"]}

The following parameters can be used instead of AttributeValueList and ComparisonOperator:

- Value - A value for DynamoDB to compare with an attribute.
- Exists - A Boolean value that causes DynamoDB to evaluate the value before attempting the conditional operation:
 - If Exists is true, DynamoDB will check to see if that attribute value already exists in the table. If it is found, then the condition evaluates to true; otherwise the condition evaluates to false.
 - If Exists is false, DynamoDB assumes that the attribute value does not exist in the table. If in fact the value does not exist, then the assumption is valid and the condition evaluates to true. If the value is found, despite the assumption that it does not exist, the condition evaluates to false.

Note that the default value for Exists is true.

The `Value` and `Exists` parameters are incompatible with `AttributeValueList` and `ComparisonOperator`. Note that if you use both sets of parameters at once, DynamoDB will return a `ValidationException` exception.

 **Note**

This parameter does not support attributes of type List or Map.

Use `ConditionExpression` instead – Example

Suppose you wanted to modify an item in the `Music` table, but only if a certain condition was true. You could use an `UpdateItem` request with an `Expected` parameter, as in this AWS CLI example:

```
aws dynamodb update-item \
  --table-name Music \
  --key '{
    "Artist": {"S":"No One You Know"},
    "SongTitle": {"S":"Call Me Today"}
}' \
  --attribute-updates '{
    "Price": {
      "Action": "PUT",
      "Value": {"N":"1.98"}
    }
}' \
  --expected '{
    "Price": {
      "ComparisonOperator": "LE",
      "AttributeValueList": [ {"N":"2.00"} ]
    }
}'
```

You can use a `ConditionExpression` instead:

```
aws dynamodb update-item \
  --table-name Music \
  --key '{
    "Artist": {"S":"No One You Know"},
    "SongTitle": {"S":"Call Me Today"}
}' \
```

```
--update-expression 'SET Price = :p1' \
--condition-expression 'Price <= :p2' \
--expression-attribute-values '{'
    ":p1": {"N":"1.98"},
    ":p2": {"N":"2.00"}
}'
```

KeyConditions (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). For specific information on the new parameter replacing this one, [use KeyConditionExpression instead..](#)

The legacy conditional parameter `KeyConditions` contains selection criteria for a `Query` operation. For a query on a table, you can have conditions only on the table primary key attributes. You must provide the partition key name and value as an EQ condition. You can optionally provide a second condition, referring to the sort key.

Note

If you don't provide a sort key condition, all of the items that match the partition key will be retrieved. If a `FilterExpression` or `QueryFilter` is present, it will be applied after the items are retrieved.

For a query on an index, you can have conditions only on the index key attributes. You must provide the index partition key name and value as an EQ condition. You can optionally provide a second condition, referring to the index sort key.

Each `KeyConditions` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the `ComparisonOperator` being used.

For type `Number`, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on Unicode with UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

- **ComparisonOperator** - A comparator for evaluating attributes. For example: equals, greater than, and less than.

For KeyConditions, only the following comparison operators are supported:

EQ | LE | LT | GE | GT | BEGINS_WITH | BETWEEN

The following are descriptions of these comparison operators.

- EQ : Equal.

AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"}. Also, {"N":"6"} does not equal {"NS":["6", "2", "1"]}.

- LE : Less than or equal.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"}. Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.

- LT : Less than.

AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"}. Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.

- GE : Greater than or equal.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example,

{"S": "6"} does not equal {"N": "6"}. Also, {"N": "6"} does not compare to {"NS": ["6", "2", "1"]}.

- GT : Greater than.

AttributeValueList can contain only one AttributeValue element of type String, Number, or Binary (not a set type). If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, {"S": "6"} does not equal {"N": "6"}. Also, {"N": "6"} does not compare to {"NS": ["6", "2", "1"]}.

- BEGINS_WITH : Checks for a prefix.

AttributeValueList can contain only one AttributeValue of type String or Binary (not a Number or a set type). The target attribute of the comparison must be of type String or Binary (not a Number or a set type).

- BETWEEN : Greater than or equal to the first value, and less than or equal to the second value.

AttributeValueList must contain two AttributeValue elements of the same type, either String, Number, or Binary (not a set type). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an AttributeValue element of a different type than the one provided in the request, the value does not match. For example, {"S": "6"} does not compare to {"N": "6"}. Also, {"N": "6"} does not compare to {"NS": ["6", "2", "1"]}.

Use *KeyConditionExpression* instead – Example

Suppose you wanted to retrieve several items with the same partition key from the *Music* table. You could use a Query request with a KeyConditions parameter, as in this AWS CLI example:

```
aws dynamodb query \
--table-name Music \
--key-conditions '{
    "Artist": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S": "No One You Know"} ]
    },
    "SongTitle": {
        "ComparisonOperator": "BETWEEN",
        "AttributeValueList": [ {"S": "A"}, {"S": "M"} ]
    }
}'
```

```
}
```

You can use a KeyConditionExpression instead:

```
aws dynamodb query \
--table-name Music \
--key-condition-expression 'Artist = :a AND SongTitle BETWEEN :t1 AND :t2' \
--expression-attribute-values '{
    ":a": {"S": "No One You Know"}, 
    ":t1": {"S": "A"}, 
    ":t2": {"S": "M"}'
}'
```

QueryFilter (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). For specific information on the new parameter replacing this one, [use FilterExpression instead..](#)

In a Query operation, the legacy conditional parameter `QueryFilter` is a condition that evaluates the query results after the items are read and returns only the desired values.

This parameter does not support attributes of type List or Map.

Note

A `QueryFilter` is applied after the items have already been read; the process of filtering does not consume any additional read capacity units.

If you provide more than one condition in the `QueryFilter` map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the [ConditionalOperator \(legacy\)](#) parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

Note that `QueryFilter` does not allow key attributes. You cannot define a filter condition on a partition key or a sort key.

Each `QueryFilter` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the operator specified in `ComparisonOperator`.

For type `Number`, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on UTF-8 binary encoding. For example, `a` is greater than `A`, and `a` is greater than `B`.

For type `Binary`, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

For information on specifying data types in JSON, see [DynamoDB low-level API](#).

- `ComparisonOperator` - A comparator for evaluating attributes. For example: equals, greater than, and less than.

The following comparison operators are available:

`EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN`

Use `FilterExpression` instead – Example

Suppose you wanted to query the `Music` table and apply a condition to the matching items. You could use a Query request with a `QueryFilter` parameter, as in this AWS CLI example:

```
aws dynamodb query \
--table-name Music \
--key-conditions '{
    "Artist": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S": "No One You Know"} ]
    }
}' \
--query-filter '{
    "Price": {
        "ComparisonOperator": "GT",
        "AttributeValueList": [ {"N": "10.00"} ]
    }
}'
```

```
        "AttributeValueList": [ {"N": "1.00"} ]  
    }  
}'
```

You can use a [FilterExpression](#) instead:

```
aws dynamodb query \  
--table-name Music \  
--key-condition-expression 'Artist = :a' \  
--filter-expression 'Price > :p' \  
--expression-attribute-values '{  
    ":p": {"N": "1.00"},  
    ":a": {"S": "No One You Know"}  
'
```

ScanFilter (legacy)

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#). For specific information on the new parameter replacing this one, [use *FilterExpression* instead..](#)

In a Scan operation, the legacy conditional parameter `ScanFilter` is a condition that evaluates the scan results and returns only the desired values.

Note

This parameter does not support attributes of type List or Map.

If you specify more than one condition in the `ScanFilter` map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the [ConditionalOperator \(legacy\)](#) parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

Each `ScanFilter` element consists of an attribute name to compare, along with the following:

- **AttributeValueList** - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the operator specified in **ComparisonOperator**.

For type Number, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

For information on specifying data types in JSON, see [DynamoDB low-level API](#).

- **ComparisonOperator** - A comparator for evaluating attributes. For example: equals, greater than, and less than.

The following comparison operators are available:

EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS |
BEGINS_WITH | IN | BETWEEN

Use *FilterExpression* instead – Example

Suppose you wanted to scan the *Music* table and apply a condition to the matching items. You could use a Scan request with a **ScanFilter** parameter, as in this AWS CLI example:

```
aws dynamodb scan \
--table-name Music \
--scan-filter '{
    "Genre": {
        "AttributeValueList": [ {"S": "Rock"} ],
        "ComparisonOperator": "EQ"
    }
}'
```

You can use a **FilterExpression** instead:

```
aws dynamodb scan \
--table-name Music \
--filter-expression 'Genre = :g' \
--expression-attribute-values '{
```

```
":g": {"S":"Rock"}  
}'
```

Writing conditions with legacy parameters

Note

We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see [Using expressions in DynamoDB](#).

The following section describes how to write conditions for use with legacy parameters, such as `Expected`, `QueryFilter`, and `ScanFilter`.

Note

New applications should use expression parameters instead. For more information, see [Using expressions in DynamoDB](#).

Simple conditions

With attribute values, you can write conditions for comparisons against table attributes. A condition always evaluates to true or false, and consists of:

- `ComparisonOperator` — greater than, less than, equal to, and so on.
- `AttributeValueList` (optional) — attribute value(s) to compare against. Depending on the `ComparisonOperator` being used, the `AttributeValueList` might contain one, two, or more values; or it might not be present at all.

The following sections describe the various comparison operators, along with examples of how to use them in conditions.

Comparison operators with no attribute values

- `NOT_NULL` - true if an attribute exists.
- `NULL` - true if an attribute does not exist.

Use these operators to check whether an attribute exists, or doesn't exist. Because there is no value to compare against, do not specify AttributeValueList.

Example

The following expression evaluates to true if the *Dimensions* attribute exists.

```
...  
    "Dimensions": {  
        ComparisonOperator: "NOT_NULL"  
    }  
...
```

Comparison operators with one attribute value

- EQ - true if an attribute is equal to a value.

AttributeValueList can contain only one value of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains a value of a different type than the one specified in the request, the value does not match. For example, the string "3" is not equal to the number 3. Also, the number 3 is not equal to the number set [3, 2, 1].

- NE - true if an attribute is not equal to a value.

AttributeValueList can contain only one value of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains a value of a different type than the one specified in the request, the value does not match.

- LE - true if an attribute is less than or equal to a value.

AttributeValueList can contain only one value of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match.

- LT - true if an attribute is less than a value.

AttributeValueList can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- GE - true if an attribute is greater than or equal to a value.

AttributeValueList can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- GT - true if an attribute is greater than a value.

AttributeValueList can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- CONTAINS - true if a value is present within a set, or if one value contains another.

AttributeValueList can contain only one value of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operator checks for a substring match. If the target attribute of the comparison is Binary, then the operator looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set, then the operator evaluates to true if it finds an exact match with any member of the set.

- NOT_CONTAINS - true if a value is *not* present within a set, or if one value does not contain another value.

AttributeValueList can contain only one value of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operator checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operator checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set, then the operator evaluates to true if it *does not* find an exact match with any member of the set.

- BEGINS_WITH - true if the first few characters of an attribute match the provided value. Do not use this operator for comparing numbers.

AttributeValueList can contain only one value of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).

Use these operators to compare an attribute with a value. You must specify an AttributeValueList consisting of a single value. For most of the operators, this value must be a scalar; however, the EQ and NE operators also support sets.

Examples

The following expressions evaluate to true if:

- A product's price is greater than 100.

```
...  
    "Price": {  
        ComparisonOperator: "GT",  
        AttributeValueList: [ {"N":"100"} ]  
    }  
...
```

- A product category begins with "Bo".

```
...  
    "ProductCategory": {  
        ComparisonOperator: "BEGINS_WITH",  
        AttributeValueList: [ {"S":"Bo"} ]  
    }  
...
```

- A product is available in either red, green, or black:

```
...  
    "Color": {  
        ComparisonOperator: "EQ",  
        AttributeValueList: [  
            [ {"S":"Black"}, {"S":"Red"}, {"S":"Green"} ]  
        ]  
    }  
...
```

 **Note**

When comparing set data types, the order of the elements does not matter. DynamoDB will return only the items with the same set of values, regardless of the order in which you specify them in your request.

Comparison operators with two attribute values

- BETWEEN - true if a value is between a lower bound and an upper bound, endpoints inclusive.

AttributeValueList must contain two elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains a value of a different type than the one specified in the request, the value does not match.

Use this operator to determine if an attribute value is within a range. The AttributeValueList must contain two scalar elements of the same type - String, Number, or Binary.

Example

The following expression evaluates to true if a product's price is between 100 and 200.

```
...
    "Price": {
        ComparisonOperator: "BETWEEN",
        AttributeValueList: [ {"N":"100"}, {"N":"200"} ]
    }
...

```

Comparison operators with *n* attribute values

- IN - true if a value is equal to any of the values in an enumerated list. Only scalar values are supported in the list, not sets. The target attribute must be of the same type and exact value in order to match.

AttributeValueList can contain one or more elements of type String, Number, or Binary (not a set). These attributes are compared against an existing non-set type attribute of an item. If *any* elements of the input set are present in the item attribute, the expression evaluates to true.

AttributeValueList can contain one or more values of type String, Number, or Binary (not a set). The target attribute of the comparison must be of the same type and exact value to match. A String never matches a String set.

Use this operator to determine whether the supplied value is within an enumerated list. You can specify any number of scalar values in AttributeValueList, but they all must be of the same data type.

Example

The following expression evaluates to true if the value for *Id* is 201, 203, or 205.

```
...  
    "Id": {  
        ComparisonOperator: "IN",  
        AttributeValueList: [ {"N":"201"}, {"N":"203"}, {"N":"205"} ]  
    }  
...
```

Using multiple conditions

DynamoDB lets you combine multiple conditions to form complex expressions. You do this by providing at least two expressions, with an optional [ConditionalOperator \(legacy\)](#).

By default, when you specify more than one condition, *all* of the conditions must evaluate to true in order for the entire expression to evaluate to true. In other words, an implicit *AND* operation takes place.

Example

The following expression evaluates to true if a product is a book that has at least 600 pages. Both of the conditions must evaluate to true, since they are implicitly *ANDed* together.

```
...  
    "ProductCategory": {  
        ComparisonOperator: "EQ",  
        AttributeValueList: [ {"S":"Book"} ]  
    },  
    "PageCount": {  
        ComparisonOperator: "GE",  
        AttributeValueList: [ {"N":600} ]  
    }  
...
```

You can use [ConditionalOperator \(legacy\)](#) to clarify that an *AND* operation will take place. The following example behaves in the same manner as the previous one.

```
...  
    "ConditionalOperator" : "AND",  
    "ProductCategory": {
```

```
    "ComparisonOperator": "EQ",
    "AttributeValueList": [ {"N":"Book"} ]
},
"PageCount": {
    "ComparisonOperator": "GE",
    "AttributeValueList": [ {"N":600} ]
}
...
...
```

You can also set `ConditionalOperator` to *OR*, which means that *at least one* of the conditions must evaluate to true.

Example

The following expression evaluates to true if a product is a mountain bike, if it is a particular brand name, or if its price is greater than 100.

```
...
ConditionalOperator : "OR",
"BicycleType": {
    "ComparisonOperator": "EQ",
    "AttributeValueList": [ {"S":"Mountain" } ]
},
"Brand": {
    "ComparisonOperator": "EQ",
    "AttributeValueList": [ {"S":"Brand-Company A" } ]
},
"Price": {
    "ComparisonOperator": "GT",
    "AttributeValueList": [ {"N":100"} ]
}
...
...
```

Note

In a complex expression, the conditions are processed in order, from the first condition to the last.

You cannot use both AND and OR in a single expression.

Other conditional operators

In previous releases of DynamoDB, the `Expected` parameter behaved differently for conditional writes. Each item in the `Expected` map represented an attribute name for DynamoDB to check, along with the following:

- `Value` — a value to compare against the attribute.
- `Exists` — determine whether the value exists prior to attempting the operation.

The `Value` and `Exists` options continue to be supported in DynamoDB; however, they only let you test for an equality condition, or whether an attribute exists. We recommend that you use `ComparisonOperator` and `AttributeValueList` instead, because these options let you construct a much wider range of conditions.

Example

A `DeleteItem` can check to see whether a book is no longer in publication, and only delete it if this condition is true. Here is an AWS CLI example using a legacy condition:

```
aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{
    "Id": {"N":"600"}
}' \
--expected '{
    "InPublication": {
        "Exists": true,
        "Value": {"BOOL":false}
    }
}'
```

The following example does the same thing, but does not use a legacy condition:

```
aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{
    "Id": {"N":"600"}
}' \
```

```
--expected '{  
    "InPublication": {  
        "ComparisonOperator": "EQ",  
        "AttributeValueList": [ {"BOOL":false} ]  
    }  
'}
```

Example

A PutItem operation can protect against overwriting an existing item with the same primary key attributes. Here is an example using a legacy condition:

```
aws dynamodb put-item \  
--table-name ProductCatalog \  
--item '{  
    "Id": {"N":"500"},  
    "Title": {"S":"Book 500 Title"}  
' \  
--expected '{  
    "Id": { "Exists": false }  
'
```

The following example does the same thing, but does not use a legacy condition:

```
aws dynamodb put-item \  
--table-name ProductCatalog \  
--item '{  
    "Id": {"N":"500"},  
    "Title": {"S":"Book 500 Title"}  
' \  
--expected '{  
    "Id": { "ComparisonOperator": "NULL" }  
'
```

Note

For conditions in the Expected map, do not use the legacy Value and Exists options together with ComparisonOperator and AttributeValueList. If you do this, your conditional write will fail.

Previous low-level API version (2011-12-05)

This section documents the operations available in the previous DynamoDB low-level API version (2011-12-05). This version of the low-level API is maintained for backward compatibility with existing applications.

New applications should use the current API version (2012-08-10). For more information, see [Low-level API reference](#).

Note

We recommend that you migrate your applications to the latest API version (2012-08-10), since new DynamoDB features will not be backported to the previous API version.

Topics

- [BatchGetItem](#)
- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTables](#)
- [GetItem](#)
- [ListTables](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)
- [UpdateTable](#)

BatchGetItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The BatchGetItem operation returns the attributes for multiple items from multiple tables using their primary keys. The maximum number of items that can be retrieved for a single operation is 100. Also, the number of items retrieved is constrained by a 1 MB size limit. If the response size limit is exceeded or a partial result is returned because the table's provisioned throughput is exceeded, or because of an internal processing failure, DynamoDB returns an UnprocessedKeys value so you can retry the operation starting with the next item to get. DynamoDB automatically adjusts the number of items returned per page to enforce this limit. For example, even if you ask to retrieve 100 items, but each individual item is 50 KB in size, the system returns 20 items and an appropriate UnprocessedKeys value so you can get the next page of results. If desired, your application can include its own logic to assemble the pages of results into one set.

If no items could be processed because of insufficient provisioned throughput on each of the tables involved in the request, DynamoDB returns a ProvisionedThroughputExceededException error.

Note

By default, BatchGetItem performs eventually consistent reads on every table in the request. You can set the ConsistentRead parameter to true, on a per-table basis, if you want consistent reads instead.

BatchGetItem fetches items in parallel to minimize response latencies.

When designing your application, keep in mind that DynamoDB does not guarantee how attributes are ordered in the returned response. Include the primary key values in the AttributesToGet for the items in your request to help parse the response by item.

If the requested items do not exist, nothing is returned in the response for those items. Requests for non-existent items consume the minimum read capacity units according to the type of read. For more information, see [DynamoDB Item sizes and formats](#).

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{"RequestItems":
  {"Table1":
    {"Keys":
      [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement": {"N":"KeyValue2"}},
       {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement": {"N":"KeyValue4"}},
       {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement": {"N":"KeyValue6"}},
      ],
      "AttributesToGet": ["AttributeName1", "AttributeName2", "AttributeName3"]
    },
    "Table2":
      {"Keys":
        [{"HashKeyElement": {"S":"KeyValue4"}},
         {"HashKeyElement": {"S":"KeyValue5"}},
        ],
        "AttributesToGet": ["AttributeName4", "AttributeName5", "AttributeName6"]
      }
    }
}
```

Name	Description	Required
RequestItems	A container of the table name and corresponding items to get by primary key. While requesting items, each table	Yes

Name	Description	Required
	<p>name can be invoked only once per operation.</p> <p>Type: String</p> <p>Default: None</p>	
Table	<p>The name of the table containing the items to get. The entry is simply a string specifying an existing table with no label.</p> <p>Type: String</p> <p>Default: None</p>	Yes
Table:Keys	<p>The primary key values that define the items in the specified table. For more information about primary keys, see Primary key.</p> <p>Type: Keys</p>	Yes
Table:AttributesToGet	<p>Array of Attribute names within the specified table. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p>	No

Name	Description	Required
Table:ConsistentRead	If set to true, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No

Responses

Syntax

```

HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 855

{"Responses":
  {"Table1":
    {"Items":
      [{"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"N": "AttributeValue"}, "AttributeName3": {"SS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}, {"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"S": "AttributeValue"}, "AttributeName3": {"NS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}, {"AttributeValue"}], "ConsumedCapacityUnits":1},
    "Table2":
    {"Items":
      [{"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"N": "AttributeValue"}, "AttributeName3": {"SS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}, {"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"S": "AttributeValue"}, "AttributeName3": {"NS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}]}],
  }
}

```

```

    "ConsumedCapacityUnits":1}
},
"UnprocessedKeys":
{
  "Table3":
  {
    "Keys":
      [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement": {"N":"KeyValue2"}},
       {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement": {"N":"KeyValue4"}},
       {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement": {"N":"KeyValue6"}}],
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]
  }
}

```

Name	Description
Responses	Table names and the respective item attribute s from the tables. Type: Map
Table	The name of the table containing the items. The entry is simply a string specifying the table with no label. Type: String
Items	Container for the attribute names and values meeting the operation parameters. Type: Map of attribute names to and their data types and values.
ConsumedCapacityUnits	The number of read capacity units consumed, for each table. This value shows the number applied toward your provisioned throughput. Requests for non-existent items consume the minimum read capacity units, depending on the type of read. For more information see

Name	Description
	<p>Managing settings on DynamoDB provisioned capacity tables.</p> <p>Type: Number</p>
UnprocessedKeys	<p>Contains an array of tables and their respective keys that were not processed with the current response, possibly due to reaching a limit on the response size. The UnprocessedKeys value is in the same form as a RequestItems parameter (so the value can be provided directly to a subsequent BatchGetItem operation). For more information, see the above RequestItems parameter.</p> <p>Type: Array</p>
UnprocessedKeys : Table: Keys	<p>The primary key attribute values that define the items and the attributes associated with the items. For more information about primary keys, see Primary key.</p> <p>Type: Array of attribute name-value pairs.</p>
UnprocessedKeys : Table: AttributeNamesToGet	<p>Attribute names within the specified table. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array of attribute names.</p>

Name	Description
UnprocessedKeys : Table: ConsistentRead	If set to true, then a consistent read is used for the specified table, otherwise an eventually consistent read is used. Type: Boolean.

Special errors

Error	Description
ProvisionedThroughputExceededException	Your maximum allowed provisioned throughput has been exceeded.

Examples

The following examples show an HTTP POST request and response using the BatchGetItem operation. For examples using the AWS SDK, see [Working with items and attributes](#).

Sample request

The following sample requests attributes from two different tables.

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0
content-length: 409

{"RequestItems": {
    "comp1": {
        "Keys": [
            {"HashKeyElement": {"S": "Casey"}, "RangeKeyElement": {"N": "1319509152"}},
            {"HashKeyElement": {"S": "Dave"}, "RangeKeyElement": {"N": "1319509155"}},
            {"HashKeyElement": {"S": "Riley"}, "RangeKeyElement": {"N": "1319509158"}}
        ],
        "AttributesToGet": ["user", "status"]
    },
    "comp2": {
        "Keys": [
            {"HashKeyElement": {"S": "Mike"}, "RangeKeyElement": {"N": "1319509153"}},
            {"HashKeyElement": {"S": "Sarah"}, "RangeKeyElement": {"N": "1319509156"}}
        ],
        "AttributesToGet": ["user", "status"]
    }
}}
```

```
{"Keys":  
    [{"HashKeyElement":{"S":"Julie"}}, {"HashKeyElement":{"S":"Mingus"}}],  
    "AttributesToGet":["user", "friends"]}  
}  
}
```

Sample response

The following sample is the response.

```
HTTP/1.1 200 OK  
x-amzn-RequestId: GTPQVRM4VJS792J1UFJTKUBVV4KQNS05AEMVJF66Q9ASUAAJG  
content-type: application/x-amz-json-1.0  
content-length: 373  
Date: Fri, 02 Sep 2011 23:07:39 GMT  
  
{"Responses":  
    {"comp1":  
        {"Items":  
            [{"status":{"S":"online"}, "user":{"S":"Casey"}},  
             {"status":{"S":"working"}, "user":{"S":"Riley"}},  
             {"status":{"S":"running"}, "user":{"S":"Dave"}}],  
        "ConsumedCapacityUnits":1.5},  
    "comp2":  
        {"Items":  
            [{"friends":{"SS":["Elisabeth", "Peter"]}, "user":{"S":"Mingus"}},  
             {"friends":{"SS":["Dave", "Peter"]}, "user":{"S":"Julie"}}],  
        "ConsumedCapacityUnits":1}  
    },  
    "UnprocessedKeys":[]  
}
```

BatchWriteItem

⚠ Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

This operation enables you to put or delete several items across multiple tables in a single call.

To upload one item, you can use `PutItem`, and to delete one item, you can use `DeleteItem`. However, when you want to upload or delete large amounts of data, such as uploading large amounts of data from Amazon EMR (Amazon EMR) or migrating data from another database in to DynamoDB, `BatchWriteItem` offers an efficient alternative.

If you use languages such as Java, you can use threads to upload items in parallel. This adds complexity in your application to handle the threads. Other languages don't support threading. For example, if you are using PHP, you must upload or delete items one at a time. In both situations, `BatchWriteItem` provides an alternative where the specified put and delete operations are processed in parallel, giving you the power of the thread pool approach without having to introduce complexity in your application.

Note that each individual put and delete specified in a `BatchWriteItem` operation costs the same in terms of consumed capacity units. However, because `BatchWriteItem` performs the specified operations in parallel, you get lower latency. Delete operations on non-existent items consume 1 write capacity unit. For more information about consumed capacity units, see [Working with tables and data in DynamoDB](#).

When using `BatchWriteItem`, note the following limitations:

- **Maximum operations in a single request**—You can specify a total of up to 25 put or delete operations; however, the total request size cannot exceed 1 MB (the HTTP payload).
- You can use the `BatchWriteItem` operation only to put and delete items. You cannot use it to update existing items.
- **Not an atomic operation**—Individual operations specified in a `BatchWriteItem` are atomic; however `BatchWriteItem` as a whole is a "best-effort" operation and not an atomic operation. That is, in a `BatchWriteItem` request, some operations might succeed and others might fail. The failed operations are returned in an `UnprocessedItems` field in the response. Some of these failures might be because you exceeded the provisioned throughput configured for the table or a transient failure such as a network error. You can investigate and optionally resend the requests. Typically, you call `BatchWriteItem` in a loop and in each iteration check for unprocessed items, and submit a new `BatchWriteItem` request with those unprocessed items.
- **Does not return any items**—The `BatchWriteItem` is designed for uploading large amounts of data efficiently. It does not provide some of the sophistication offered by `PutItem` and

`DeleteItem`. For example, `DeleteItem` supports the `ReturnValues` field in your request body to request the deleted item in the response. The `BatchWriteItem` operation does not return any items in the response.

- Unlike `PutItem` and `DeleteItem`, `BatchWriteItem` does not allow you to specify conditions on individual write requests in the operation.
- Attribute values must not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests that have empty values will be rejected with a `ValidationException`.

DynamoDB rejects the entire batch write operation if any one of the following is true:

- If one or more tables specified in the `BatchWriteItem` request does not exist.
- If primary key attributes specified on an item in the request does not match the corresponding table's primary key schema.
- If you try to perform multiple operations on the same item in the same `BatchWriteItem` request. For example, you cannot put and delete the same item in the same `BatchWriteItem` request.
- If the total request size exceeds the 1 MB request size (the HTTP payload) limit.
- If any individual item in a batch exceeds the 64 KB item size limit.

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
    "RequestItems" : RequestItems
}

RequestItems
{
    "TableName1" : [ Request, Request, ... ],
    "TableName2" : [ Request, Request, ... ],
}
```

```
    ...
}

Request ::=  
  PutRequest | DeleteRequest

PutRequest ::=  
{  
  "PutRequest" : {  
    "Item" : {  
      "Attribute-Name1" : Attribute-Value,  
      "Attribute-Name2" : Attribute-Value,  
      ...  
    }  
  }  
}

DeleteRequest ::=  
{  
  "DeleteRequest" : {  
    "Key" : PrimaryKey-Value  
  }  
}

PrimaryKey-Value ::= HashTypePK | HashAndRangeTypePK

HashTypePK ::=  
{  
  "HashKeyElement" : Attribute-Value  
}

HashAndRangeTypePK  
{  
  "HashKeyElement" : Attribute-Value,  
  "RangeKeyElement" : Attribute-Value,  
}

Attribute-Value ::= String | Numeric | Binary | StringSet | NumericSet | BinarySet

Numeric ::=  
{  
  "N": "Number"  
}
```

```
String ::=  
{  
    "S": "String"  
}  
  
Binary ::=  
{  
    "B": "Base64 encoded binary data"  
}  
  
StringSet ::=  
{  
    "SS": [ "String1", "String2", ... ]  
}  
  
NumberSet ::=  
{  
    "NS": [ "Number1", "Number2", ... ]  
}  
  
BinarySet ::=  
{  
    "BS": [ "Binary1", "Binary2", ... ]  
}
```

In the request body, the `RequestItems` JSON object describes the operations that you want to perform. The operations are grouped by tables. You can use `BatchWriteItem` to update or delete several items across multiple tables. For each specific write request, you must identify the type of request (`PutItem`, `DeleteItem`) followed by detail information about the operation.

- For a `PutRequest`, you provide the item, that is, a list of attributes and their values.
- For a `DeleteRequest`, you provide the primary key name and value.

Responses

Syntax

The following is the syntax of the JSON body returned in the response.

{

```
"Responses" : ConsumedCapacityUnitsByTable  
"UnprocessedItems" : RequestItems  
}  
  
ConsumedCapacityUnitsByTable  
{  
  "TableName1" : { "ConsumedCapacityUnits" : NumericValue },  
  "TableName2" : { "ConsumedCapacityUnits" : NumericValue },  
  ...  
}
```

RequestItems

This syntax is identical to the one described in the JSON syntax in the request.

Special errors

No errors specific to this operation.

Examples

The following example shows an HTTP POST request and the response of a BatchWriteItem operation. The request specifies the following operations on the Reply and the Thread tables:

- Put an item and delete an item from the Reply table
- Put an item into the Thread table

For examples using the AWS SDK, see [Working with items and attributes](#).

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.BatchGetItem  
content-type: application/x-amz-json-1.0  
  
{  
  "RequestItems":{  
    "Reply": [  
      {  
        "PutRequest":{  
          "Item":{
```

Sample response

The following example response shows a put operation on both the Thread and Reply tables succeeded and a delete operation on the Reply table failed (for reasons such as throttling that is

caused when you exceed the provisioned throughput on the table). Note the following in the JSON response:

- The Responses object shows one capacity unit was consumed on both the Thread and Reply tables as a result of the successful put operation on each of these tables.
- The UnprocessedItems object shows the unsuccessful delete operation on the Reply table. You can then issue a new BatchWriteItem call to address these unprocessed requests.

```
HTTP/1.1 200 OK
x-amzn-RequestId: G8M9ANL0E5QA26AEUHJKJE0ASBVV4KQNS05AEMVJF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 536
Date: Thu, 05 Apr 2012 18:22:09 GMT
```

```
{
    "Responses": {
        "Thread": {
            "ConsumedCapacityUnits": 1.0
        },
        "Reply": {
            "ConsumedCapacityUnits": 1.0
        }
    },
    "UnprocessedItems": {
        "Reply": [
            {
                "DeleteRequest": {
                    "Key": {
                        "HashKeyElement": {
                            "S": "DynamoDB#DynamoDB Thread 4"
                        },
                        "RangeKeyElement": {
                            "S": "oops - accidental row"
                        }
                    }
                }
            ]
        }
    }
}
```

CreateTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The CreateTable operation adds a new table to your account.

The table name must be unique among those associated with the AWS Account issuing the request, and the AWS region that receives the request (such as dynamodb.us-west-2.amazonaws.com). Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-west-1.amazonaws.com, they are completely independent and do not share any data.

The CreateTable operation triggers an asynchronous workflow to begin creating the table. DynamoDB immediately returns the state of the table (CREATING) until the table is in the ACTIVE state. Once the table is in the ACTIVE state, you can perform data plane operations.

Use the [DescribeTables](#) operation to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.CreateTable  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1",  
 "KeySchema":  
     {"HashKeyElement": {"AttributeName": "AttributeName1", "AttributeType": "S"},  
      "RangeKeyElement": {"AttributeName": "AttributeName2", "AttributeType": "N"}},  
 "ProvisionedThroughput": {"ReadCapacityUnits": 5, "WriteCapacityUnits": 10}}
```

}

Name	Description	Required
TableName	<p>The name of the table to create.</p> <p>Allowed characters are a-z, A-Z, 0-9, '_' (underscore), '-' (dash), and '.' (dot). Names can be between 3 and 255 characters long.</p> <p>Type: String</p>	Yes
KeySchema	<p>The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary key.</p> <p>Primary key element names can be between 1 and 255 characters long with no character restrictions.</p> <p>Possible values for the AttributeType are "S" (string), "N" (numeric), or "B" (binary).</p> <p>Type: Map of HashKeyElement , or HashKeyEl</p>	Yes

Name	Description	Required
	ement and RangeKeyE lement for a composite primary key.	
ProvisionedThrough put	New throughput for the specified table, consisting of values for ReadCapac ityUnits and WriteCapa cityUnits . For details, see Managing settings on DynamoDB provisioned capacity tables .	Yes

 **Note**

For current
maximum/minimum
values, see [Service,
account, and table
quotas in Amazon
DynamoDB](#).

Type: Array

Name	Description	Required
ProvisionedThroughput : ReadCapacityUnits	<p>Sets the minimum number of consistent ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> <p>Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent ReadCapacityUnits per second provides 100 eventually consistent ReadCapacityUnits per second.</p> <p>Type: Number</p>	Yes
ProvisionedThroughput : WriteCapacityUnits	<p>Sets the minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> <p>Type: Number</p>	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR000KIRLG0HVAICUFVV4KQNS05AEMVJF66Q9ASUAAJG
```

```
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
  {"CreationDateTime":1.310506263362E9,
   "KeySchema":
     {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
      "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
   "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
   "TableName":"Table1",
   "TableStatus":"CREATING"
  }
}
```

Name	Description
TableDescription	A container for the table properties.
CreationDateTime	Date when the table was created in UNIX epoch time . Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary key . Type: Map of HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.
ProvisionedThroughput	Throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits . See Managing

Name	Description
ProvisionedThroughput :ReadCapacityUnits	<p>settings on DynamoDB provisioned capacity tables.</p> <p>Type: Array</p>
ProvisionedThroughput :WriteCapacityUnits	<p>The minimum number of ReadCapacityUnits consumed per second before DynamoDB balances the load with other operations</p> <p>Type: Number</p>
TableName	<p>The name of the created table.</p> <p>Type: String</p>
TableStatus	<p>The current state of the table (CREATING). Once the table is in the ACTIVE state, you can put data in it.</p> <p>Use the DescribeTables API to check the status of the table.</p> <p>Type: String</p>

Special errors

Error	Description
ResourceInUseException	Attempt to recreate an already existing table.
LimitExceededException	The number of simultaneous table requests (cumulative number of tables in the CREATING, DELETING or UPDATING state) exceeds the maximum allowed. <p>Note For current maximum/minimum values, see Service, account, and table quotas in Amazon DynamoDB.</p>

Examples

The following example creates a table with a composite primary key containing a string and a number. For examples using the AWS SDK, see [Working with tables and data in DynamoDB](#).

Sample request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.CreateTable  
content-type: application/x-amz-json-1.0  
  
{"TableName":"comp-table",  
 "KeySchema":  
     {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},  
      "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},  
 "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10}}
```

```
}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR000KIRLG0HVAICUFVV4KQNS05AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
  {"CreationDateTime":1.310506263362E9,
  "KeySchema":
    {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
     "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
  "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
  "TableName":"comp-table",
  "TableStatus":"CREATING"
}
}
```

Related actions

- [DescribeTables](#)
- [DeleteTable](#)

DeleteItem

A Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Deletes a single item in a table by primary key. You can perform a conditional delete operation that deletes the item if it exists, or if it has an expected attribute value.

Note

If you specify `DeleteItem` without attributes or values, all the attributes for the item are deleted.

Unless you specify conditions, the `DeleteItem` is an idempotent operation; running it multiple times on the same item or attribute does *not* result in an error response.

Conditional deletes are useful for only deleting items and attributes if specific conditions are met. If the conditions are met, DynamoDB performs the delete. Otherwise, the item is not deleted.

You can perform the expected conditional check on one attribute per operation.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.DeleteItem  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1",  
 "Key":  
     {"HashKeyElement": {"S": "AttributeValue1"}, "RangeKeyElement":  
      {"N": "AttributeValue2"}},  
     "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue3"}},  
     "ReturnValues": "ALL_OLD"}  
}
```

Name	Description	Required
TableName	The name of the table containing the item to delete. Type: String	Yes
Key	The primary key that defines the item. For more informati	Yes

Name	Description	Required
	<p>on about primary keys, see Primary key.</p> <p>Type: Map of HashKeyElement to its value and RangeKeyElement to its value.</p>	
Expected	<p>Designates an attribute for a conditional delete. The Expected parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute has a particular value before deleting it.</p> <p>Type: Map of attribute names.</p>	No
Expected:AttributeName	<p>The name of the attribute for the conditional put.</p> <p>Type: String</p>	No

Name	Description	Required
Expected:Attribute Name: ExpectedA ttributeValue	<p>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.</p> <p>The following JSON notation deletes the item if the "Color" attribute doesn't exist for that item:</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;"><pre>"Expected" : {"Color": {"Exists": false}}</pre></div> <p>The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before deleting the item:</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;"><pre>"Expected" : {"Color": {"Exists": true}, {"Value": {"S": "Yellow"}}}</pre></div> <p>By default, if you use the Expected parameter and provide a Value, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify {"Exists":true} , because it is implied. You can shorten the request to:</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;"><pre>"Expected" :</pre></div>	No

Name	Description	Required
	<pre>{"Color":{"Value": "S":"Yellow"}}}</pre> <p>Note If you specify <code>{"Exists":true}</code> without an attribute value to check, DynamoDB returns an error.</p>	
ReturnValues	<p>Use this parameter if you want to get the attribute name-value pairs before they were deleted. Possible parameter values are NONE (default) or ALL_OLD. If ALL_OLD is specified, the content of the old item is returned. If this parameter is not provided or is NONE, nothing is returned.</p> <p>Type: String</p>	No

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CS0C7TJPLR000KIRLG0HVAICUFVV4KQNS05AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 21:31:03 GMT
```

```
{"Attributes":  
  {"AttributeName3":{"SS":["AttributeValue3","AttributeValue4","AttributeValue5"]},  
   "AttributeName2":{"S":"AttributeValue2"},  
   "AttributeName1":{"N":"AttributeValue1"}  
 },  
 "ConsumedCapacityUnits":1  
}
```

Name	Description
Attributes	If the <code>ReturnValues</code> parameter is provided as <code>ALL_OLD</code> in the request, DynamoDB returns an array of attribute name-value pairs (essentially, the deleted item). Otherwise, the response contains an empty set. Type: Array of attribute name-value pairs.
ConsumedCapacityUnits	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. Delete requests on non-existent items consume 1 write capacity unit. For more information see Managing settings on DynamoDB provisioned capacity tables . Type: Number

Special errors

Error	Description
ConditionalCheckFailedException	Conditional check failed. An expected attribute value was not found.

Examples

Sample request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.DeleteItem  
content-type: application/x-amz-json-1.0  
  
{"TableName":"comp-table",  
 "Key":  
     {"HashKeyElement":{"S":"Mingus"}, "RangeKeyElement":{"N":"200"}},  
 "Expected":  
     {"status":{"Value":{"S":"shopping"}},  
 "ReturnValues":"ALL_OLD"  
}
```

Sample response

```
HTTP/1.1 200 OK  
x-amzn-RequestId: U9809LI6BBFJA5N2R0TB0P017JVV4KQNS05AEMVJF66Q9ASUAAJG  
content-type: application/x-amz-json-1.0  
content-length: 353  
Date: Tue, 12 Jul 2011 22:31:23 GMT  
  
{"Attributes":  
    {"friends":{"SS":["Dooley","Ben","Daisy"]},  
     "status":{"S":"shopping"},  
     "time":{"N":"200"},  
     "user":{"S":"Mingus"}  
 },  
 "ConsumedCapacityUnits":1  
}
```

Related actions

- [PutItem](#)

DeleteTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The DeleteTable operation deletes a table and all of its items. After a DeleteTable request, the specified table is in the DELETING state until DynamoDB completes the deletion. If the table is in the ACTIVE state, you can delete it. If a table is in CREATING or UPDATING states, then DynamoDB returns a ResourceInUseException error. If the specified table does not exist, DynamoDB returns a ResourceNotFoundException. If table is already in the DELETING state, no error is returned.

Note

DynamoDB might continue to accept data plane operation requests, such as GetItem and PutItem, on a table in the DELETING state until the table deletion is complete.

Tables are unique among those associated with the AWS Account issuing the request, and the AWS region that receives the request (such as dynamodb.us-west-1.amazonaws.com). Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-west-1.amazonaws.com, they are completely independent and do not share any data; deleting one does not delete the other.

Use the [DescribeTables](#) operation to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
```

```
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0

{"TableName": "Table1"}
```

Name	Description	Required
TableName	The name of the table to delete. Type: String	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNS05AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Sun, 14 Aug 2011 22:56:22 GMT

{"TableDescription":
  {"CreationDateTime":1.313362508446E9,
   "KeySchema":
     {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
      "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
   "ProvisionedThroughput":{"ReadCapacityUnits":10,"WriteCapacityUnits":10},
   "TableName": "Table1",
   "TableStatus": "DELETING"
  }
}
```

Name	Description
TableDescription	A container for the table properties.
CreationDateTime	Date when the table was created.

Name	Description
	Type: Number
KeySchema	<p>The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary key.</p>
	<p>Type: Map of HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.</p>
ProvisionedThroughput	<p>Throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits . See Managing settings on DynamoDB provisioned capacity tables.</p>
ProvisionedThroughput : ReadCapacityUnits	<p>The minimum number of ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p>
	Type: Number
ProvisionedThroughput : WriteCapacityUnits	<p>The minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p>
	Type: Number

Name	Description
TableName	<p>The name of the deleted table.</p> <p>Type: String</p>
TableStatus	<p>The current state of the table (DELETING). Once the table is deleted, subsequent requests for the table return resource not found .</p> <p>Use the DescribeTables operation to check the status of the table.</p> <p>Type: String</p>

Special errors

Error	Description
ResourceInUseException	Table is in state CREATING or UPDATING and can't be deleted.

Examples

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0
content-length: 40

{"TableName":"favorite-movies-table"}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNS05AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 160
Date: Sun, 14 Aug 2011 17:20:03 GMT

{"TableDescription":
  {"CreationDateTime":1.313362508446E9,
  "KeySchema":
    {"HashKeyElement":{"AttributeName":"name","AttributeType":"S"}},
  "TableName":"favorite-movies-table",
  "TableStatus":"DELETING"
}
```

Related actions

- [CreateTable](#)
- [DescribeTables](#)

DescribeTables

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Returns information about the table, including the current status of the table, the primary key schema and when the table was created. `DescribeTable` results are eventually consistent. If you use `DescribeTable` too early in the process of creating a table, DynamoDB returns a `ResourceNotFoundException`. If you use `DescribeTable` too early in the process of updating a table, the new values might not be immediately available.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.DescribeTable  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1"}
```

Name	Description	Required
TableName	The name of the table to describe. Type: String	Yes

Responses

Syntax

```
HTTP/1.1 200  
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375  
content-type: application/x-amz-json-1.0  
Content-Length: 543  
  
{"Table":  
    {"CreationDateTime":1.309988345372E9,  
     "ItemCount":1,  
     "KeySchema":  
         {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},  
          "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},  
     "ProvisionedThroughput": {"LastIncreaseDateTime": Date, "LastDecreaseDateTime": Date, "ReadCapacityUnits":10,"WriteCapacityUnits":10},  
     "TableName": "Table1",  
     "TableSizeBytes":1,  
     "TableStatus": "ACTIVE"}
```

```
}
```

Name	Description
Table	Container for the table being described. Type: String
CreationDateTime	Date when the table was created in UNIX epoch time .
ItemCount	Number of items in the specified table. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separately (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary key .
ProvisionedThroughput	Throughput for the specified table, consisting of values for LastIncreaseDateTime (if applicable), LastDecreaseDateTime (if applicable), ReadCapacityUnits and WriteCapacityUnits . If the throughput for the table has never been increased or decreased, DynamoDB does not return values

Name	Description
	for those elements. See Managing settings on DynamoDB provisioned capacity tables .
	Type: Array
TableName	The name of the requested table. Type: String
TableSizeBytes	Total size of the specified table, in bytes. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
TableStatus	The current state of the table (CREATING, ACTIVE, DELETING or UPDATING). Once the table is in the ACTIVE state, you can add data.

Special errors

No errors are specific to this operation.

Examples

The following examples show an HTTP POST request and response using the `DescribeTable` operation for a table named "comp-table". The table has a composite primary key.

Sample Request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.DescribeTable  
content-type: application/x-amz-json-1.0  
  
{"TableName": "users"}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 543

{"Table":
  {"CreationDateTime":1.309988345372E9,
   "ItemCount":23,
   "KeySchema":
     {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
      "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
   "ProvisionedThroughput":{"LastIncreaseDateTime": 1.309988345384E9,
    "ReadCapacityUnits":10,"WriteCapacityUnits":10},
   "TableName":"users",
   "TableSizeBytes":949,
   "TableStatus":"ACTIVE"
  }
}
```

Related actions

- [CreateTable](#)
- [DeleteTable](#)
- [ListTables](#)

GetItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The GetItem operation returns a set of Attributes for an item that matches the primary key. If there is no matching item, GetItem does not return any data.

The GetItem operation provides an eventually consistent read by default. If eventually consistent reads are not acceptable for your application, use ConsistentRead. Although this operation might take longer than a standard read, it always returns the last updated value. For more information, see [Read consistency](#).

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.GetItem  
content-type: application/x-amz-json-1.0  
  
{"TableName":"Table1",  
 "Key":  
 {"HashKeyElement": {"S":"AttributeValue1"},  
 "RangeKeyElement": {"N":"AttributeValue2"}  
 },  
 "AttributesToGet":["AttributeName3","AttributeName4"],  
 "ConsistentRead":Boolean  
}
```

Name	Description	Required
TableName	The name of the table containing the requested item. Type: String	Yes
Key	The primary key values that define the item. For more information about primary keys, see Primary key .	Yes

Name	Description	Required
	Type: Map of HashKeyElement to its value and RangeKeyElement to its value.	
AttributesToGet	<p>Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p>	No
ConsistentRead	<p>If set to true, then a consistent read is issued, otherwise eventually consistent is used.</p> <p>Type: Boolean</p>	No

Responses

Syntax

```

HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 144

{"Item": {
    "AttributeName3": {"S": "AttributeValue3"}, 
    "AttributeName4": {"N": "AttributeValue4"}, 
    "AttributeName5": {"B": "dmFsdWU="}
},
"ConsumedCapacityUnits": 0.5
}

```

Name	Description
Item	Contains the requested attributes. Type: Map of attribute name-value pairs.
ConsumedCapacityUnits	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. Requests for non-existent items consume the minimum read capacity units, depending on the type of read. For more information see Managing settings on DynamoDB provisioned capacity tables . Type: Number

Special errors

No errors specific to this operation.

Examples

For examples using the AWS SDK, see [Working with items and attributes](#).

Sample request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.GetItem  
content-type: application/x-amz-json-1.0  
  
{"TableName": "comptable",  
 "Key":  
 {"HashKeyElement": {"S": "Julie"},  
 "RangeKeyElement": {"N": "1307654345"}},  
 "AttributesToGet": ["status", "friends"],  
 "ConsistentRead": true  
}
```

Sample response

Notice the ConsumedCapacityUnits value is 1, because the optional parameter ConsistentRead is set to true. If ConsistentRead is set to false (or not specified) for the same request, the response is eventually consistent and the ConsumedCapacityUnits value would be 0.5.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 72

{"Item":
 {"friends":{"SS":["Lynda, Aaron"]},
 "status":{"S":"online"}
 },
 "ConsumedCapacityUnits": 1
}
```

ListTables

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Returns an array of all the tables associated with the current account and endpoint.

Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-east-1.amazonaws.com, they are completely independent and do not share any data. The ListTables operation returns all of the table names associated with the account making the request, for the endpoint that receives the request.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.ListTables  
content-type: application/x-amz-json-1.0  
  
{"ExclusiveStartTableName": "Table1", "Limit": 3}
```

The ListTables operation, by default, requests all of the table names associated with the account making the request, for the endpoint that receives the request.

Name	Description	Required
Limit	A number of maximum table names to return. Type: Integer	No
ExclusiveStartTableName	The name of the table that starts the list. If you already ran a ListTables operation and received an LastEvaluatedTableName value in the response, use that value here to continue the list. Type: String	No

Responses

Syntax

```
HTTP/1.1 200 OK  
x-amzn-RequestId: S1LEK2DPQP80JNHVHL80U2M7KRVV4KQNS05AEMVJF66Q9ASUAAJG  
content-type: application/x-amz-json-1.0
```

```
content-length: 81  
Date: Fri, 21 Oct 2011 20:35:38 GMT
```

```
{"TableNames":["Table1","Table2","Table3"], "LastEvaluatedTableName":"Table3"}
```

Name	Description
TableNames	The names of the tables associated with the current account at the current endpoint. Type: Array
LastEvaluatedTableName	The name of the last table in the current list, only if some tables for the account and endpoint have not been returned. This value does not exist in a response if all table names are already returned. Use this value as the ExclusiveStartTableName in a new request to continue the list until all the table names are returned. Type: String

Special errors

No errors are specific to this operation.

Examples

The following examples show an HTTP POST request and response using the ListTables operation.

Sample request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.ListTables  
content-type: application/x-amz-json-1.0
```

```
{"ExclusiveStartTableName": "comp2", "Limit": 3}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: S1LEK2DPQP80JNHVHL80U2M7KRVV4KQNS05AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT
```

```
{"LastEvaluatedTableName": "comp5", "TableNames": ["comp3", "comp4", "comp5"]}
```

Related actions

- [DescribeTables](#)
- [CreateTable](#)
- [DeleteTable](#)

PutItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Creates a new item, or replaces an old item with a new item (including all the attributes). If an item already exists in the specified table with the same primary key, the new item completely replaces the existing item. You can perform a conditional put (insert a new item if one with the specified primary key doesn't exist), or replace an existing item if it has certain attribute values.

Attribute values may not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException`.

Note

To ensure that a new item does not replace an existing item, use a conditional put operation with `Exists` set to `false` for the primary key attribute, or attributes.

For more information about using `PutItem`, see [Working with items and attributes](#).

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.PutItem  
content-type: application/x-amz-json-1.0  
  
{"TableName":"Table1",  
 "Item":{  
     "AttributeName1":{"S":"AttributeValue1"},  
     "AttributeName2":{"N":"AttributeValue2"},  
     "AttributeName5":{"B":"dmFsdWU="}  
 },  
 "Expected":{"AttributeName3":{"Value": {"S":"AttributeValue"}, "Exists":Boolean}},  
 "ReturnValues":ReturnValuesConstant"}
```

Name	Description	Required
TableName	The name of the table to contain the item. Type: String	Yes
Item	A map of the attributes for the item, and must include the primary key values that define the item. Other attribute name-value pairs can be provided for the item.	Yes

Name	Description	Required
	<p>For more information about primary keys, see Primary key.</p> <p>Type: Map of attribute names to attribute values.</p>	
Expected	<p>Designates an attribute for a conditional put. The Expected parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute value already exists; or if the attribute value exists and has a particular value before changing it.</p> <p>Type: Map of an attribute names to an attribute value, and whether it exists.</p>	No
Expected:AttributeName	<p>The name of the attribute for the conditional put.</p> <p>Type: String</p>	No

Name	Description	Required
Expected:Attribute Name: ExpectedA ttributeValue	<p>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.</p> <p>The following JSON notation replaces the item if the "Color" attribute doesn't already exist for that item:</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;"><pre>"Expected" : {"Color": {"Exists": false}}</pre></div> <p>The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before replacing the item:</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;"><pre>"Expected" : {"Color": {"Exists": true, "Value": {"S": "Yellow"}}}</pre></div> <p>By default, if you use the Expected parameter and provide a Value, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify {"Exists":true} , because it is implied. You can shorten the request to:</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;"><pre>"Expected" :</pre></div>	No

Name	Description	Required
	<pre>{"Color":{"Value": "S":"Yellow"}}}</pre> <p>Note If you specify <code>{"Exists":true}</code> without an attribute value to check, DynamoDB returns an error.</p>	
ReturnValues	<p>Use this parameter if you want to get the attribute name-value pairs before they were updated with the PutItem request. Possible parameter values are NONE (default) or ALL_OLD. If ALL_OLD is specified, and PutItem overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is NONE, nothing is returned.</p> <p>Type: String</p>	No

Responses

Syntax

The following syntax example assumes the request specified a `ReturnValues` parameter of `ALL_OLD`; otherwise, the response has only the `ConsumedCapacityUnits` element.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 85
```

```
{"Attributes":
  {"AttributeName3":{"S":"AttributeValue3"},
   "AttributeName2":{"SS":"AttributeValue2"},
   "AttributeName1":{"SS":"AttributeValue1"},
  },
  "ConsumedCapacityUnits":1
}
```

Name	Description
Attributes	<p>Attribute values before the put operation, but only if the <code>ReturnValues</code> parameter is specified as <code>ALL_OLD</code> in the request.</p> <p>Type: Map of attribute name-value pairs.</p>
ConsumedCapacityUnits	<p>The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing settings on DynamoDB provisioned capacity tables.</p> <p>Type: Number</p>

Special errors

Error	Description
ConditionalCheckFailedException	Conditional check failed. An expected attribute value was not found.
ResourceNotFoundException	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see [Working with items and attributes](#).

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Item":
  {"time":{"N":"300"},
   "feeling":{"S":"not surprised"},
   "user":{"S":"Riley"}
  },
 "Expected":
  {"feeling":{"Value":{"S":"surprised"}, "Exists":true}}
 "ReturnValues":"ALL_OLD"
}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 8952fa74-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 84

{"Attributes":
 {"feeling":{"S":"surprised"},
  "time":{"N":"300"},
  "user":{"S":"Riley"}},
 "ConsumedCapacityUnits":1
}
```

Related actions

- [UpdateItem](#)
- [DeleteItem](#)
- [GetItem](#)

- [BatchGetItem](#)

Query

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

A Query operation gets the values of one or more items and their attributes by primary key (Query is only available for hash-and-range primary key tables). You must provide a specific HashKeyValue, and can narrow the scope of the query using comparison operators on the RangeKeyValue of the primary key. Use the ScanIndexForward parameter to get results in forward or reverse order by range key.

Queries that do not return results consume the minimum read capacity units according to the type of read.

Note

If the total number of items meeting the query parameters exceeds the 1MB limit, the query stops and results are returned to the user with a LastEvaluatedKey to continue the query in a subsequent operation. Unlike a Scan operation, a Query operation never returns an empty result set *and* a LastEvaluatedKey. The LastEvaluatedKey is only provided if the results exceed 1MB, or if you have used the Limit parameter.

The result can be set for a consistent read using the ConsistentRead parameter.

Requests

Syntax

```
// This header is abbreviated.
```

```
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
 "Limit":2,
 "ConsistentRead":true,
 "HashKeyValue":{"S":"AttributeValue1"}, 
 "RangeKeyCondition": {"AttributeValueList":
 [{"N":"AttributeValue2"}]}, "ComparisonOperator":"GT"}
 "ScanIndexForward":true,
 "ExclusiveStartKey":{ 
 "HashKeyElement":{"S":"AttributeName1"}, 
 "RangeKeyElement":{"N":"AttributeName2"}},
 },
 "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

Name	Description	Required
TableName	The name of the table containing the requested items. Type: String	Yes
AttributesToGet	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
Limit	The maximum number of items to return (not necessarily the number of matching items). If Dynamo	No

Name	Description	Required
	<p>DB processes the number of items up to the limit while querying the table, it stops the query and returns the matching values up to that point, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue the query. Also, if the result set size exceeds 1MB before DynamoDB hits this limit, it stops the query and returns the matching values, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue the query.</p> <p>Type: Number</p>	
ConsistentRead	<p>If set to true, then a consistent read is issued, otherwise eventually consistent is used.</p> <p>Type: Boolean</p>	No

Name	Description	Required
Count	<p>If set to true, DynamoDB returns a total number of items that match the query parameters, instead of a list of the matching items and their attributes. You can apply the <code>Limit</code> parameter to count-only queries.</p> <p>Do not set <code>Count</code> to true while providing a list of <code>AttributesToGet</code> ; otherwise, DynamoDB returns a validation error. For more information, see Counting the items in the results.</p> <p>Type: Boolean</p>	No
HashKeyValue	<p>Attribute value of the hash component of the composite primary key.</p> <p>Type: String, Number, or Binary</p>	Yes

Name	Description	Required
RangeKeyCondition	A container for the attribute values and comparison operators to use for the query. A query request does not require a RangeKeyCondition . If you provide only the HashKeyValue , DynamoDB returns all items with the specified hash key element value. Type: Map	No
RangeKeyCondition : AttributeValueList	The attribute values to evaluate for the query parameters. The AttributeValueList contains one attribute value, unless a BETWEEN comparison is specified. For the BETWEEN comparison, the AttributeValueList contains two attribute values. Type: A map of AttributeValue to a ComparisonOperator .	No

Name	Description	Required
RangeKeyCondition : ComparisonOperator	<p>The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a Query operation.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"><p>Note</p><p>String value comparisons for greater than, equals, or less than are based on ASCII character code values. For example, a is greater than A, and aa is greater than B. For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters.</p><p>For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.</p></div> <p>Type: String or Binary</p>	No

Name	Description	Required
	<p>EQ : Equal.</p> <p>For EQ, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not equal {"NS":["6", "2", "1"]}.</p>	
	<p>LE : Less than or equal.</p> <p>For LE, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
	<p>LT : Less than.</p> <p>For LT, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	
	<p>GE : Greater than or equal.</p> <p>For GE, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
	<p>GT : Greater than.</p> <p>For GT, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	
	<p>BEGINS_WITH : checks for a prefix.</p> <p>For BEGINS_WITH , AttributeValueList can contain only oneAttributeValue of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).</p>	

Name	Description	Required
	<p>BETWEEN : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For BETWEEN, Attribute ValueList must contain two AttributeValue elements of the same type , either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not compare to {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
ScanIndexForward	<p>Specifies ascending or descending traversal of the index. DynamoDB returns results reflecting the requested order determined by the range key: If the data type is Number, the results are returned in numeric order; otherwise, the traversal is based on ASCII character code values.</p> <p>Type: Boolean</p> <p>Default is true (ascending).</p>	No
ExclusiveStartKey	<p>Primary key of the item from which to continue an earlier query. An earlier query might provide this value as the LastEvaluatedKey if that query operation was interrupted before completing the query; either because of the result set size or the Limit parameter. The LastEvaluatedKey can be passed back in a new query request to continue the operation from that point.</p> <p>Type: HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.</p>	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items":>[
    "AttributeName1":{"S":"AttributeValue1"},
    "AttributeName2":{"N":"AttributeValue2"},
    "AttributeName3":{"S":"AttributeValue3"}
],{
    "AttributeName1":{"S":"AttributeValue3"},
    "AttributeName2":{"N":"AttributeValue4"},
    "AttributeName3":{"S":"AttributeValue3"},
    "AttributeName5":{"B":"dmFsdWU="}
}],
"LastEvaluatedKey": {"HashKeyElement": {"AttributeValue3": "S"}, "RangeKeyElement": {"AttributeValue4": "N"}},
"ConsumedCapacityUnits":1
}
```

Name	Description
Items	Item attributes meeting the query parameters. Type: Map of attribute names to and their data types and values.
Count	Number of items in the response. For more information, see Counting the items in the results . Type: Number
LastEvaluatedKey	Primary key of the item where the query operation stopped, inclusive of the previous result set. Use this value to start a new

Name	Description
	<p>operation excluding this value in the new request.</p> <p>The <code>LastEvaluatedKey</code> is null when the entire query result set is complete (i.e. the operation processed the “last page”).</p> <p>Type: <code>HashKeyElement</code>, or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key.</p>
ConsumedCapacityUnits	<p>The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing settings on DynamoDB provisioned capacity tables.</p> <p>Type: Number</p>

Special errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see [Query operations in DynamoDB](#).

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
```

```
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"John"},
 "ScanIndexForward":false,
 "ExclusiveStartKey": {
   "HashKeyElement":{"S":"John"},
   "RangeKeyElement":{"S":"The Matrix"}
 }
}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 3647e778-71eb-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items": [
  {"fans":{"SS":["Jody","Jake"]}, "name":{"S":"John"}, "rating":{"S":"****"}, "title":{"S":"The End"}},
  {"fans":{"SS":["Jody","Jake"]}, "name":{"S":"John"}, "rating":{"S":"****"}, "title":{"S":"The Beatles"}}
], "LastEvaluatedKey": {"HashKeyElement": {"S": "John"}, "RangeKeyElement": {"S": "The Beatles"}}, "ConsumedCapacityUnits":1
}
```

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
```

```
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"Airplane"},
 "RangeKeyCondition":{"AttributeValueList":[{"N":"1980"}],"ComparisonOperator":"EQ"},
 "ScanIndexForward":false}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 8b9ee1ad-774c-11e0-9172-d954e38f553a
content-type: application/x-amz-json-1.0
content-length: 119

{"Count":1,"Items":>[
  "fans":{"SS":["Dave","Aaron"]},
  "name":{"S":"Airplane"},
  "rating":{"S":"***"},
  "year":{"N":"1980"}
],
"ConsumedCapacityUnits":1
}
```

Related actions

- [Scan](#)

Scan

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The Scan operation returns one or more items and its attributes by performing a full scan of a table. Provide a ScanFilter to get more specific results.

Note

If the total number of scanned items exceeds the 1MB limit, the scan stops and results are returned to the user with a LastEvaluatedKey to continue the scan in a subsequent operation. The results also include the number of items exceeding the limit. A scan can result in no table data meeting the filter criteria.

The result set is eventually consistent.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.Scan  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1",  
 "Limit": 2,  
 "ScanFilter":{  
     "AttributeName1": {"AttributeValueList":  
         [{"S": "AttributeValue"}]},  
     "ComparisonOperator": "EQ"  
 },  
 "ExclusiveStartKey":{  
     "HashKeyElement": {"S": "AttributeName1"},  
     "RangeKeyElement": {"N": "AttributeName2"}  
 },  
 "AttributesToGet": ["AttributeName1", "AttributeName2", "AttributeName3"]}  
}
```

Name	Description	Required
TableName	<p>The name of the table containing the requested items.</p> <p>Type: String</p>	Yes
AttributesToGet	<p>Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p>	No
Limit	<p>The maximum number of items to evaluate (not necessarily the number of matching items). If Dynamo DB processes the number of items up to the limit while processing the results, it stops and returns the matching values up to that point, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue retrieving items. Also, if the scanned data set size exceeds 1MB before DynamoDB reaches this limit, it stops the scan and returns the matching values up to the limit, and a <code>LastEvaluatedKey</code> to apply in a</p>	No

Name	Description	Required
	<p>subsequent operation to continue the scan.</p> <p>Type: Number</p>	
Count	<p>If set to true, DynamoDB returns a total number of items for the Scan operation, even if the operation has no matching items for the assigned filter. You can apply the Limit parameter to count-only scans.</p> <p>Do not set Count to true while providing a list of AttributesToGet , otherwise DynamoDB returns a validation error. For more information, see Counting the items in the results.</p> <p>Type: Boolean</p>	No
ScanFilter	<p>Evaluates the scan results and returns only the desired values. Multiple conditions are treated as "AND" operations: all conditions must be met to be included in the results.</p> <p>Type: A map of attribute names to values with comparison operators.</p>	No

Name	Description	Required
ScanFilter :Attribute ValueList	<p>The values and conditions to evaluate the scan results for the filter.</p> <p>Type: A map of Attribute Value to a Condition .</p>	No

Name	Description	Required
ScanFilter : ComparisonOperator	<p>The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a scan operation.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"><p>Note</p><p>String value comparisons for greater than, equals, or less than are based on ASCII character code values. For example, a is greater than A, and aa is greater than B. For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters.</p><p>For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.</p></div> <p>Type: String or Binary</p>	No

Name	Description	Required
	<p>EQ : Equal.</p> <p>For EQ, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not equal {"NS":["6", "2", "1"]}.</p>	
	<p>NE : Not Equal.</p> <p>For NE, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not equal {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
	<p>LE : Less than or equal.</p> <p>For LE, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	
	<p>LT : Less than.</p> <p>For LT, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
	<p>GE : Greater than or equal.</p> <p>For GE, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	
	<p>GT : Greater than.</p> <p>For GT, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
	NOT_NULL : Attribute exists. NULL : Attribute does not exist.	
	CONTAINS : checks for a subsequence, or value in a set. For CONTAINS, Attribute ValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operation checks for a substring match. If the target attribute of the comparison is Binary, then the operation looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operation checks for a member of the set (not as a substring).	

Name	Description	Required
	<p>NOT_CONTAINS : checks for absence of a subsequence, or absence of a value in a set.</p> <p>For NOT_CONTAINS , AttributeValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operation checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operation checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operation checks for the absence of a member of the set (not as a substring).</p>	

Name	Description	Required
	<p><code>BEGINS_WITH</code> : checks for a prefix.</p> <p>For <code>BEGINS_WITH</code> , <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).</p>	
	<p><code>IN</code> : checks for exact matches.</p> <p>For <code>IN</code>, <code>AttributeValueList</code> can contain more than one <code>AttributeValue</code> of type String, Number, or Binary (not a set). The target attribute of the comparison must be of the same type and exact value to match. A String never matches a String set.</p>	

Name	Description	Required
	<p>BETWEEN : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For BETWEEN, Attribute ValueList must contain two AttributeValue elements of the same type , either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not compare to {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.</p>	

Name	Description	Required
ExclusiveStartKey	<p>Primary key of the item from which to continue an earlier scan. An earlier scan might provide this value if that scan operation was interrupted before scanning the entire table; either because of the result set size or the Limit parameter. The LastEvaluatedKey can be passed back in a new scan request to continue the operation from that point.</p> <p>Type: HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.</p>	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 229
```

```
{"Count":2,"Items":[{"AttributeNames1":{"S":"AttributeValue1"}, "AttributeNames2":{"S":"AttributeValue2"}, "AttributeNames3":{"S":"AttributeValue3"}}, {"AttributeNames1":{"S":"AttributeValue4"}, "AttributeNames2":{"S":"AttributeValue5"}, "AttributeNames3":{"S":"AttributeValue6"}, "AttributeNames5":{"B":"dmFsdWU="}}]
```

```

}],  

"LastEvaluatedKey":  

{"HashKeyElement":{"S":"AttributeName1"},  

"RangeKeyElement":{"N":"AttributeName2"},  

"ConsumedCapacityUnits":1,  

"ScannedCount":2}  

}

```

Name	Description
Items	<p>Container for the attributes meeting the operation parameters.</p> <p>Type: Map of attribute names to and their data types and values.</p>
Count	<p>Number of items in the response. For more information, see Counting the items in the results.</p> <p>Type: Number</p>
ScannedCount	<p>Number of items in the complete scan before any filters are applied. A high ScannedCount value with few, or no, Count results indicates an inefficient Scan operation. For more information, see Counting the items in the results.</p> <p>Type: Number</p>
LastEvaluatedKey	<p>Primary key of the item where the scan operation stopped. Provide this value in a subsequent scan operation to continue the operation from that point.</p> <p>The LastEvaluatedKey is null when the entire scan result set is complete (i.e. the operation processed the “last page”).</p>

Name	Description
ConsumedCapacityUnits	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing settings on DynamoDB provisioned capacity tables . Type: Number

Special errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see [Working with scans in DynamoDB](#).

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName": "1-hash-rangetable", "ScanFilter": {}}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 4e8a5fa9-71e7-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 465

{"Count":4, "Items": [{}]
```

```
"date": {"S": "1980"},  
"fans": {"SS": ["Dave", "Aaron"]},  
"name": {"S": "Airplane"},  
"rating": {"S": "****"}  
}, {  
"date": {"S": "1999"},  
"fans": {"SS": ["Ziggy", "Laura", "Dean"]},  
"name": {"S": "Matrix"},  
"rating": {"S": "*****"}  
}, {  
"date": {"S": "1976"},  
"fans": {"SS": ["Riley"]},  
"name": {"S": "The Shaggy D.A."},  
"rating": {"S": "***"}  
}, {  
"date": {"S": "1985"},  
"fans": {"SS": ["Fox", "Lloyd"]},  
"name": {"S": "Back To The Future"},  
"rating": {"S": "****"}  
}],  
    "ConsumedCapacityUnits": 0.5  
"ScannedCount": 4}
```

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.Scan  
content-type: application/x-amz-json-1.0  
content-length: 125  
  
{"TableName": "comp5",  
"ScanFilter":  
    {"time":  
        {"AttributeValueList": [{"N": "400"}],  
        "ComparisonOperator": "GT"}  
    }  
}
```

Sample response

```
HTTP/1.1 200 OK
```

```
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNS05AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 262
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":2,
 "Items":[
 {"friends":{"SS":["Dave","Ziggy","Barrie"]},
 "status":{"S":"chatting"},
 "time":{"N":"2000"},
 "user":{"S":"Casey"}},
 {"friends":{"SS":["Dave","Ziggy","Barrie"]},
 "status":{"S":"chatting"},
 "time":{"N":"2000"},
 "user":{"S":"Freddy"}}
 ],
 "ConsumedCapacityUnits":0.5
 "ScannedCount":4
}
```

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Limit":2,
 "ScanFilter":
 {"time":
 {"AttributeValueList":[{"N":"400"}],
 "ComparisonOperator":"GT"}
 },
 "ExclusiveStartKey":
 {"HashKeyElement":{"S":"Freddy"}, "RangeKeyElement":{"N":"2000"}}
}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNS05AEMVJF66Q9ASUAAJG
```

```
content-type: application/x-amz-json-1.0
content-length: 232
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":1,
 "Items":[
 {"friends":{"SS":["Jane","James","John"]},
 "status":{"S":"exercising"},
 "time":{"N":"2200"},
 "user":{"S":"Roger"}}
 ],
 "LastEvaluatedKey":{"HashKeyElement":{"S":"Riley"},"RangeKeyElement":{"N":"250"}},
 "ConsumedCapacityUnits":0.5
 "ScannedCount":2
 }
```

Related actions

- [Query](#)
- [BatchGetItem](#)

UpdateItem

⚠ Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Edits an existing item's attributes. You can perform a conditional update (insert a new attribute name-value pair if it doesn't exist, or replace an existing name-value pair if it has certain expected attribute values).

Note

You cannot update the primary key attributes using `UpdateItem`. Instead, delete the item and use `PutItem` to create a new item with new attributes.

The `UpdateItem` operation includes an `Action` parameter, which defines how to perform the update. You can put, delete, or add attribute values.

Attribute values may not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException`.

If an existing item has the specified primary key:

- **PUT**— Adds the specified attribute. If the attribute exists, it is replaced by the new value.
- **DELETE**— If no value is specified, this removes the attribute and its value. If a set of values is specified, then the values in the specified set are removed from the old set. So if the attribute value contains [a,b,c] and the delete action contains [a,c], then the final attribute value is [b]. The type of the specified value must match the existing value type. Specifying an empty set is not valid.
- **ADD**— Only use the add action for numbers or if the target attribute is a set (including string sets). ADD does not work if the target attribute is a single string value or a scalar binary value. The specified value is added to a numeric value (incrementing or decrementing the existing numeric value) or added as an additional value in a string set. If a set of values is specified, the values are added to the existing set. For example if the original set is [1,2] and supplied value is [3], then after the add operation the set is [1,2,3], not [4,5]. An error occurs if an Add action is specified for a set attribute and the attribute type specified does not match the existing set type.

If you use ADD for an attribute that does not exist, the attribute and its values are added to the item.

If no item matches the specified primary key:

- **PUT**— Creates a new item with specified primary key. Then adds the specified attribute.
- **DELETE**— Nothing happens.

- **ADD**— Creates an item with supplied primary key and number (or set of numbers) for the attribute value. Not valid for a string or a binary type.

Note

If you use ADD to increment or decrement a number value for an item that doesn't exist before the update, DynamoDB uses `0` as the initial value. Also, if you update an item using ADD to increment or decrement a number value for an attribute that doesn't exist before the update (but the item does) DynamoDB uses `0` as the initial value. For example, you use ADD to add `+3` to an attribute that did not exist before the update. DynamoDB uses `0` for the initial value, and the value after the update is `3`.

For more information about using this operation, see [Working with items and attributes](#).

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateItem  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1",  
 "Key":  
     {"HashKeyElement": {"S": "AttributeValue1"},  
      "RangeKeyElement": {"N": "AttributeValue2"}},  
     "AttributeUpdates": {"AttributeName3": {"Value":  
         {"S": "AttributeValue3_New"}, "Action": "PUT"}},  
     "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue3_Current"}},  
     "ReturnValues": "ReturnValuesConstant"  
}
```

Name	Description	Required
TableName	<p>The name of the table containing the item to update.</p> <p>Type: String</p>	Yes
Key	<p>The primary key that defines the item. For more information about primary keys, see Primary key.</p> <p>Type: Map of HashKeyElement to its value and RangeKeyElement to its value.</p>	Yes
AttributeUpdates	<p>Map of attribute name to the new value and action for the update. The attribute names specify the attributes to modify, and cannot contain any primary key attributes.</p> <p>Type: Map of attribute name, value, and an action for the attribute update.</p>	
AttributeUpdates :Action	<p>Specifies how to perform the update. Possible values: PUT (default), ADD or DELETE. The semantics are explained in the UpdateItem description.</p> <p>Type: String</p>	No

Name	Description	Required
	Default: PUT	
Expected	<p>Designates an attribute for a conditional update. The Expected parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute value already exists; or if the attribute value exists and has a particular value before changing it.</p> <p>Type: Map of attribute names.</p>	No
Expected:AttributeName	<p>The name of the attribute for the conditional put.</p> <p>Type: String</p>	No

Name	Description	Required
Expected:Attribute Name: ExpectedA ttributeValue	<p>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.</p> <p>The following JSON notation updates the item if the "Color" attribute doesn't already exist for that item:</p> <pre>"Expected" : {"Color":{"Exists":false}}</pre> <p>The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before updating the item:</p> <pre>"Expected" : {"Color":{"Exists":true}, {"Value": {"S":"Yellow"}}}</pre> <p>By default, if you use the Expected parameter and provide a Value, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify {"Exists":true} , because it is implied. You can shorten the request to:</p> <pre>"Expected" :</pre>	No

Name	Description	Required
	<pre>{"Color":{"Value": {"S":"Yellow"}}}</pre> <p>Note If you specify <code>{"Exists":true}</code> without an attribute value to check, DynamoDB returns an error.</p>	

Name	Description	Required
ReturnValues	<p>Use this parameter if you want to get the attribute name-value pairs before they were updated with the <code>UpdateItem</code> request. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>, <code>UPDATED_OLD</code>, <code>ALL_NEW</code> or <code>UPDATED_NEW</code>. If <code>ALL_OLD</code> is specified, and <code>UpdateItem</code> overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is <code>NONE</code>, nothing is returned. If <code>ALL_NEW</code> is specified, then all the attributes of the new version of the item are returned. If <code>UPDATED_NEW</code> is specified, then the new versions of only the updated attributes are returned.</p> <p>Type: String</p>	No

Responses

Syntax

The following syntax example assumes the request specified a `ReturnValues` parameter of `ALL_OLD`; otherwise, the response has only the `ConsumedCapacityUnits` element.

```
HTTP/1.1 200
```

```
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
```

```
content-type: application/x-amz-json-1.0
content-length: 140

{"Attributes": {
    "AttributeName1": {"S": "AttributeValue1"},
    "AttributeName2": {"S": "AttributeValue2"},
    "AttributeName3": {"S": "AttributeValue3"},
    "AttributeName5": {"B": "dmFsdWU="}
},
"ConsumedCapacityUnits": 1
}
```

Name	Description
Attributes	A map of attribute name-value pairs, but only if the <code>ReturnValues</code> parameter is specified as something other than <code>NONE</code> in the request. Type: Map of attribute name-value pairs.
ConsumedCapacityUnits	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing settings on DynamoDB provisioned capacity tables . Type: Number

Special errors

Error	Description
ConditionalCheckFailedException	Conditional check failed. Attribute ("+ name +") value is ("+ value +") but was expected ("+ expValue +")
ResourceNotFoundExceptions	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see [Working with items and attributes](#).

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateItem  
content-type: application/x-amz-json-1.0  
  
{"TableName": "comp5",  
 "Key":  
     {"HashKeyElement": {"S": "Julie"}, "RangeKeyElement": {"N": "1307654350"}},  
 "AttributeUpdates":  
     {"status": {"Value": {"S": "online"}},  
      "Action": "PUT"},  
     {"Expected": {"status": {"Value": {"S": "offline"}},  
      "ReturnValues": "ALL_NEW"}  
}
```

Sample response

```
HTTP/1.1 200 OK  
x-amzn-RequestId: 5IMH07F01Q9P7Q6QMKMMI3R3QRVV4KQNS05AEMVJF66Q9ASUAAJG  
content-type: application/x-amz-json-1.0  
content-length: 121  
Date: Fri, 26 Aug 2011 21:05:00 GMT  
  
{"Attributes":  
    {"friends": {"SS": ["Lynda", "Aaron"]},  
     "status": {"S": "online"},  
     "time": {"N": "1307654350"},  
     "user": {"S": "Julie"}},  
    "ConsumedCapacityUnits": 1  
}
```

Related actions

- [PutItem](#)
- [DeleteItem](#)

UpdateTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Updates the provisioned throughput for the given table. Setting the throughput for a table helps you manage performance and is part of the provisioned throughput feature of DynamoDB. For more information, see [Managing settings on DynamoDB provisioned capacity tables](#).

The provisioned throughput values can be upgraded or downgraded based on the maximums and minimums listed in [Service, account, and table quotas in Amazon DynamoDB](#).

The table must be in the ACTIVE state for this operation to succeed. UpdateTable is an asynchronous operation; while executing the operation, the table is in the UPDATING state. While the table is in the UPDATING state, the table still has the provisioned throughput from before the call. The new provisioned throughput setting is in effect only when the table returns to the ACTIVE state after the UpdateTable operation.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateTable  
content-type: application/x-amz-json-1.0  
  
{"TableName":"Table1",  
 "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}  
}
```

Name	Description	Required
TableName	<p>The name of the table to update.</p> <p>Type: String</p>	Yes
ProvisionedThroughput	<p>New throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits . See Managing settings on DynamoDB provisioned capacity tables.</p> <p>Type: Array</p>	Yes
ProvisionedThroughput :ReadCapacityUnits	<p>Sets the minimum number of consistent ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> <p>Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent ReadCapacityUnits per second provides 100 eventually consistent ReadCapacityUnits per second.</p> <p>Type: Number</p>	Yes

Name	Description	Required
ProvisionedThroughput put :WriteCapacityUnits	Sets the minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CS0C7TJPLR000KIRLG0HVAICUFVV4KQNS05AEMVJF66Q9ASUAAJG
Content-Type: application/json
Content-Length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
  {"CreationDateTime":1.321657838135E9,
  "KeySchema":
    {"HashKeyElement":{"AttributeName":"AttributeValue1","AttributeType":"S"},
     "RangeKeyElement":{"AttributeName":"AttributeValue2","AttributeType":"N"}},
  "ProvisionedThroughput":
    {"LastDecreaseDateTime":1.321661704489E9,
     "LastIncreaseDateTime":1.321663607695E9,
     "ReadCapacityUnits":5,
     "WriteCapacityUnits":10},
  "TableName":"Table1",
  "TableStatus":"UPDATING"}}
```

Name	Description
CreationDateTime	Date when the table was created.

Name	Description
	Type: Number
KeySchema	<p>The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separately (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary key.</p> <p>Type: Map of HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.</p>
ProvisionedThroughput	<p>Current throughput settings for the specified table, including values for LastIncreaseDateTime (if applicable), LastDecreaseDateTime (if applicable),</p> <p>Type: Array</p>
TableName	<p>The name of the updated table.</p> <p>Type: String</p>
TableStatus	<p>The current state of the table (CREATING, ACTIVE, DELETING or UPDATING), which should be UPDATING.</p> <p>Use the DescribeTables operation to check the status of the table.</p> <p>Type: String</p>

Special errors

Error	Description
ResourceNotFoundException	The specified table was not found.
ResourceInUseException	The table is not in the ACTIVE state.

Examples

Sample request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB low-level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateTable  
content-type: application/x-amz-json-1.0  
  
{"TableName":"comp1",  
 "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}  
}
```

Sample response

```
HTTP/1.1 200 OK  
content-type: application/x-amz-json-1.0  
content-length: 390  
Date: Sat, 19 Nov 2011 00:46:47 GMT  
  
{"TableDescription":  
 {"CreationDateTime":1.321657838135E9,  
 "KeySchema":  
 {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},  
 "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},  
 "ProvisionedThroughput":  
 {"LastDecreaseDateTime":1.321661704489E9,  
 "LastIncreaseDateTime":1.321663607695E9,  
 "ReadCapacityUnits":5,  
 "WriteCapacityUnits":10},  
 "TableName":"comp1",  
 "TableStatus":"UPDATING"}
```

{}

Related actions

- [CreateTable](#)
- [DescribeTables](#)
- [DeleteTable](#)

AWS SDK for Java 1.x examples

This section contains example code for DAX applications using SDK for Java 1.x.

Topics

- [Using DAX with AWS SDK for Java 1.x](#)
- [Modifying an existing SDK for Java 1.x application to use DAX](#)
- [Querying global secondary indexes with SDK for Java 1.x](#)

Using DAX with AWS SDK for Java 1.x

Follow this procedure to run the Java sample for Amazon DynamoDB Accelerator (DAX) on your Amazon EC2 instance.

Note

These instructions are for applications using AWS SDK for Java 1.x. For applications using AWS SDK for Java 2.x, see [Java and DAX](#).

To run the Java sample for DAX

1. Install the Java Development Kit (JDK).

```
sudo yum install -y java-devel
```

2. Download the AWS SDK for Java (.zip file), and then extract it.

```
wget http://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip
```

```
unzip aws-java-sdk.zip
```

3. Download the latest version of the DAX Java client (.jar file).

```
wget http://dax-sdk.s3-website-us-west-2.amazonaws.com/java/DaxJavaClient-latest.jar
```

 **Note**

The client for the DAX SDK for Java is available on Apache Maven. For more information, see [Using the client as an Apache Maven dependency](#).

4. Set your CLASSPATH variable. In this example, replace *sdkVersion* with the actual version number of the AWS SDK for Java (for example, 1.11.112).

```
export SDKVERSION=sdkVersion  
  
export CLASSPATH=$(pwd)/TryDax/java:$(pwd)/DaxJavaClient-latest.jar:$(pwd)/aws-java-sdk-$SDKVERSION/lib/aws-java-sdk-$SDKVERSION.jar:$(pwd)/aws-java-sdk-$SDKVERSION/third-party/lib/*
```

5. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

6. Navigate to the Java code directory and compile the code as follows.

```
cd TryDax/java/  
javac TryDax*.java
```

7. Run the program.

```
java TryDax
```

You should see output similar to the following.

Creating a DynamoDB client

```
Attempting to create table; please wait...
Successfully created table. Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
Writing 10 items for partition key: 4
Writing 10 items for partition key: 5
Writing 10 items for partition key: 6
Writing 10 items for partition key: 7
Writing 10 items for partition key: 8
Writing 10 items for partition key: 9
Writing 10 items for partition key: 10
```

```
Running GetItem, Scan, and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits
```

```
GetItem test - partition key 1 and sort keys 1-10
Total time: 136.681 ms - Avg time: 13.668 ms
Total time: 122.632 ms - Avg time: 12.263 ms
Total time: 167.762 ms - Avg time: 16.776 ms
Total time: 108.130 ms - Avg time: 10.813 ms
Total time: 137.890 ms - Avg time: 13.789 ms
Query test - partition key 5 and sort keys between 2 and 9
Total time: 13.560 ms - Avg time: 2.712 ms
Total time: 11.339 ms - Avg time: 2.268 ms
Total time: 7.809 ms - Avg time: 1.562 ms
Total time: 10.736 ms - Avg time: 2.147 ms
Total time: 12.122 ms - Avg time: 2.424 ms
Scan test - all items in the table
Total time: 58.952 ms - Avg time: 11.790 ms
Total time: 25.507 ms - Avg time: 5.101 ms
Total time: 37.660 ms - Avg time: 7.532 ms
Total time: 26.781 ms - Avg time: 5.356 ms
Total time: 46.076 ms - Avg time: 9.215 ms
```

```
Attempting to delete table; please wait...
```

```
Successfully deleted table.
```

Take note of the timing information—the number of milliseconds required for the GetItem, Query, and Scan tests.

8. In the previous step, you ran the program against the DynamoDB endpoint. Now run the program again, but this time, the GetItem, Query, and Scan operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint is shown in the output, as in the following example.

```
{  
    "Address": "my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com",  
    "Port": 8111,  
    "URL": "dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com"  
}
```

Now run the program again, but this time, specify the cluster endpoint as a command line parameter.

```
java TryDax dax://my-cluster.16fzcv.dax-clusters.us-east-1.amazonaws.com
```

Look at the rest of the output, and take note of the timing information. The elapsed times for GetItem, Query, and Scan should be significantly lower with DAX than with DynamoDB.

For more information about this program, see the following sections:

- [TryDax.java](#)

- [TryDaxHelper.java](#)
- [TryDaxTests.java](#)

Using the client as an Apache Maven dependency

Follow these steps to use the client for the DAX SDK for Java in your application as a dependency.

To use the client as a Maven dependency

1. Download and install Apache Maven. For more information, see [Downloading Apache Maven](#) and [Installing Apache Maven](#).
2. Add the client Maven dependency to your application's Project Object Model (POM) file. In this example, replace x.x.x.x with the actual version number of the client (for example, 1.0.200704.0).

```
<!--Dependency:-->
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>amazon-dax-client</artifactId>
        <version>x.x.x.x</version>
    </dependency>
</dependencies>
```

TryDax.java

The TryDax.java file contains the main method. If you run the program with no command line parameters, it creates an Amazon DynamoDB client and uses that client for all API operations. If you specify a DynamoDB Accelerator (DAX) cluster endpoint on the command line, the program also creates a DAX client and uses it for GetItem, Query, and Scan operations.

You can modify the program in several ways:

- Use the DAX client instead of the DynamoDB client. For more information, see [Java and DAX](#).
- Choose a different name for the test table.
- Modify the number of items written by changing the helper.writeData parameters. The second parameter is the number of partition keys, and the third parameter is the number of sort

keys. By default, the program uses 1–10 for partition key values and 1–10 for sort key values, for a total of 100 items written to the table. For more information, see [TryDaxHelper.java](#).

- Modify the number of GetItem, Query, and Scan tests, and modify their parameters.
- Comment out the lines containing helper.createTable and helper.deleteTable (if you don't want to create and delete the table each time you run the program).

Note

To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see [Using the client as an Apache Maven dependency](#).

Alternatively, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See [Java and DAX](#) for an example of setting your CLASSPATH variable.

```
public class TryDax {  
  
    public static void main(String[] args) throws Exception {  
  
        TryDaxHelper helper = new TryDaxHelper();  
        TryDaxTests tests = new TryDaxTests();  
  
        DynamoDB ddbClient = helper.getDynamoDBClient();  
        DynamoDB daxClient = null;  
        if (args.length >= 1) {  
            daxClient = helper.getDaxClient(args[0]);  
        }  
  
        String tableName = "TryDaxTable";  
  
        System.out.println("Creating table...");  
        helper.createTable(tableName, ddbClient);  
        System.out.println("Populating table...");  
        helper.writeData(tableName, ddbClient, 10, 10);  
  
        DynamoDB testClient = null;  
        if (daxClient != null) {
```

```
        testClient = daxClient;
    } else {
        testClient = ddbClient;
    }

    System.out.println("Running GetItem, Scan, and Query tests...");
    System.out.println("First iteration of each test will result in cache misses");
    System.out.println("Next iterations are cache hits\n");

    // GetItem
    tests.getItemTest(tableName, testClient, 1, 10, 5);

    // Query
    tests.queryTest(tableName, testClient, 5, 2, 9, 5);

    // Scan
    tests.scanTest(tableName, testClient, 5);

    helper.deleteTable(tableName, ddbClient);
}

}
```

TryDaxHelper.java

The TryDaxHelper.java file contains utility methods.

The getDynamoDBClient and getDaxClient methods provide Amazon DynamoDB and DynamoDB Accelerator (DAX) clients. For control plane operations (CreateTable, DeleteTable) and write operations, the program uses the DynamoDB client. If you specify a DAX cluster endpoint, the main program creates a DAX client for performing read operations (GetItem, Query, Scan).

The other TryDaxHelper methods (createTable, writeData, deleteTable) are for setting up and tearing down the DynamoDB table and its data.

You can modify the program in several ways:

- Use different provisioned throughput settings for the table.
- Modify the size of each item written (see the stringSize variable in the writeData method).
- Modify the number of GetItem, Query, and Scan tests and their parameters.

- Comment out the lines containing `helper.CreateTable` and `helper.DeleteTable` (if you don't want to create and delete the table each time you run the program).

Note

To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see [Using the client as an Apache Maven dependency](#).

Or, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See [Java and DAX](#) for an example of setting your CLASSPATH variable.

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
import com.amazonaws.util.EC2MetadataUtils;

public class TryDaxHelper {

    private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

    DynamoDB getDynamoDBClient() {
        System.out.println("Creating a DynamoDB client");
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withRegion(region)
            .build();
        return new DynamoDB(client);
    }

    DynamoDB getDaxClient(String daxEndpoint) {
        System.out.println("Creating a DAX client with cluster endpoint " +
daxEndpoint);
```

```
AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
AmazonDynamoDB client = daxClientBuilder.build();
return new DynamoDB(client);
}

void createTable(String tableName, DynamoDB client) {
    Table table = client.getTable(tableName);
    try {
        System.out.println("Attempting to create table; please wait...");

        table = client.createTable(tableName,
            Arrays.asList(
                new KeySchemaElement("pk", KeyType.HASH), // Partition key
                new KeySchemaElement("sk", KeyType.RANGE)), // Sort key
            Arrays.asList(
                new AttributeDefinition("pk", ScalarAttributeType.N),
                new AttributeDefinition("sk", ScalarAttributeType.N)),
            new ProvisionedThroughput(10L, 10L));
        table.waitForActive();
        System.out.println("Successfully created table. Table status: " +
            table.getDescription().getTableStatus());

    } catch (Exception e) {
        System.err.println("Unable to create table: ");
        e.printStackTrace();
    }
}

void writeData(String tableName, DynamoDB client, int pkmax, int skmax) {
    Table table = client.getTable(tableName);
    System.out.println("Writing data to the table...");

    int stringSize = 1000;
    StringBuilder sb = new StringBuilder(stringSize);
    for (int i = 0; i < stringSize; i++) {
        sb.append('X');
    }
    String someData = sb.toString();

    try {
        for (Integer ipk = 1; ipk <= pkmax; ipk++) {
            System.out.println(("Writing " + skmax + " items for partition key: " +
ipk));
        }
    }
```

```
        for (Integer isk = 1; isk <= skmax; isk++) {
            table.putItem(new Item()
                .withPrimaryKey("pk", ipk, "sk", isk)
                .withString("someData", someData));
        }
    }
} catch (Exception e) {
    System.err.println("Unable to write item:");
    e.printStackTrace();
}
}

void deleteTable(String tableName, DynamoDB client) {
    Table table = client.getTable(tableName);
    try {
        System.out.println("\nAttempting to delete table; please wait...");
        table.delete();
        table.waitForDelete();
        System.out.println("Successfully deleted table.");
    } catch (Exception e) {
        System.err.println("Unable to delete table: ");
        e.printStackTrace();
    }
}
}
```

TryDaxTests.java

The TryDaxTests.java file contains methods that perform read operations against a test table in Amazon DynamoDB. These methods are not concerned with how they access the data (using either the DynamoDB client or the DAX client), so there is no need to modify the application logic.

You can modify the program in several ways:

- Modify the queryTest method so that it uses a different KeyConditionExpression.
- Add a ScanFilter to the scanTest method so that only some of the items are returned to you.

Note

To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see [Using the client as an Apache Maven dependency](#).

Or, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See [Java and DAX](#) for an example of setting your CLASSPATH variable.

```
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;

public class TryDaxTests {

    void getItemTest(String tableName, DynamoDB client, int pk, int sk, int iterations)
    {
        long startTime, endTime;
        System.out.println("GetItem test - partition key " + pk + " and sort keys 1-" + sk);
        Table table = client.getTable(tableName);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            try {
                for (Integer ipk = 1; ipk <= pk; ipk++) {
                    for (Integer isk = 1; isk <= sk; isk++) {
                        table.getItem("pk", ipk, "sk", isk);
                    }
                }
            } catch (Exception e) {
                System.err.println("Unable to get item:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, pk * sk);
        }
    }
}
```

```
    }

}

void queryTest(String tableName, DynamoDB client, int pk, int sk1, int sk2, int iterations) {
    long startTime, endTime;
    System.out.println("Query test - partition key " + pk + " and sort keys between " + sk1 + " and " + sk2);
    Table table = client.getTable(tableName);

    HashMap<String, Object> valueMap = new HashMap<String, Object>();
    valueMap.put(":pkval", pk);
    valueMap.put(":skval1", sk1);
    valueMap.put(":skval2", sk2);

    QuerySpec spec = new QuerySpec()
        .withKeyConditionExpression("pk = :pkval and sk between :skval1 and :skval2")
        .WithValueMap(valueMap);

    for (int i = 0; i < iterations; i++) {
        startTime = System.nanoTime();
        ItemCollection<QueryOutcome> items = table.query(spec);

        try {
            Iterator<Item> iter = items.iterator();
            while (iter.hasNext()) {
                iter.next();
            }
        } catch (Exception e) {
            System.err.println("Unable to query table:");
            e.printStackTrace();
        }
        endTime = System.nanoTime();
        printTime(startTime, endTime, iterations);
    }
}

void scanTest(String tableName, DynamoDB client, int iterations) {
    long startTime, endTime;
    System.out.println("Scan test - all items in the table");
    Table table = client.getTable(tableName);

    for (int i = 0; i < iterations; i++) {
```

```
startTime = System.nanoTime();
ItemCollection<ScanOutcome> items = table.scan();
try {

    Iterator<Item> iter = items.iterator();
    while (iter.hasNext()) {
        iter.next();
    }
} catch (Exception e) {
    System.err.println("Unable to scan table:");
    e.printStackTrace();
}
endTime = System.nanoTime();
printTime(startTime, endTime, iterations);
}

public void printTime(long startTime, long endTime, int iterations) {
    System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) /
(1000000.0));
    System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *
1000000.0));
}
```

Modifying an existing SDK for Java 1.x application to use DAX

If you already have a Java application that uses Amazon DynamoDB, you have to modify it so that it can access your DynamoDB Accelerator (DAX) cluster. You don't have to rewrite the entire application because the DAX Java client is similar to the DynamoDB low-level client included in the AWS SDK for Java.

Note

These instructions are for applications using AWS SDK for Java 1.x. For applications using AWS SDK for Java 2.x, see [Modifying an existing application to use DAX](#).

Suppose that you have a DynamoDB table named `Music`. The partition key for the table is `Artist`, and its sort key is `SongTitle`. The following program reads an item directly from the `Music` table.

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class GetMusicItem {

    public static void main(String[] args) throws Exception {

        // Create a DynamoDB client
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music").withKey(key);

        try {
            System.out.println("Attempting to read the item...");
            GetItemResult result = client.getItem(request);
            System.out.println("GetItem succeeded: " + result);

        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }
    }
}
```

To modify the program, replace the DynamoDB client with a DAX client.

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;
```

```
public class GetMusicItem {  
  
    public static void main(String[] args) throws Exception {  
  
        //Create a DAX client  
  
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();  
        daxClientBuilder.withRegion("us-  
east-1").withEndpointConfiguration("mydaxcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com");  
        AmazonDynamoDB client = daxClientBuilder.build();  
  
        /*  
         * ...  
         * Remaining code omitted (it is identical)  
         * ...  
        */  
  
    }  
}
```

Using the DynamoDB document API

The AWS SDK for Java provides a document interface for DynamoDB. The document API acts as a wrapper around the low-level DynamoDB client. For more information, see [Document interfaces](#).

The document interface can also be used with the low-level DAX client, as shown in the following example.

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.GetItemOutcome;  
import com.amazonaws.services.dynamodbv2.document.Table;  
  
public class GetMusicItemWithDocumentApi {  
  
    public static void main(String[] args) throws Exception {  
  
        //Create a DAX client  
  
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
```

```
daxClientBuilder.withRegion("us-east-1").withEndpointConfiguration("mydaxcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com");
AmazonDynamoDB client = daxClientBuilder.build();

// Document client wrapper
DynamoDB docClient = new DynamoDB(client);

Table table = docClient.getTable("Music");

try {
    System.out.println("Attempting to read the item...");
    GetItemOutcome outcome = table.tgetItemOutcome(
        "Artist", "No One You Know",
        "SongTitle", "Scared of My Shadow");
    System.out.println(outcome.getItem());
    System.out.println("GetItem succeeded: " + outcome);
} catch (Exception e) {
    System.err.println("Unable to read item");
    System.err.println(e.getMessage());
}

}

}
```

DAX async client

The AmazonDaxClient is synchronous. For a long-running DAX API operation, such as a Scan of a large table, this can block program execution until the operation is complete. If your program needs to perform other work while a DAX API operation is in progress, you can use ClusterDaxAsyncClient instead.

The following program shows how to use ClusterDaxAsyncClient, along with Java Future, to implement a non-blocking solution.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import com.amazon.dax.client.dynamodbv2.ClientConfig;
import com.amazon.dax.client.dynamodbv2.ClusterDaxAsyncClient;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.handlers.AsyncHandler;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBAsync;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
```

```
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class DaxAsyncClientDemo {
    public static void main(String[] args) throws Exception {

        ClientConfig daxConfig = new ClientConfig().withCredentialsProvider(new
ProfileCredentialsProvider())
            .withEndpoints("mydaxcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111");

        AmazonDynamoDBAsync client = new ClusterDaxAsyncClient(daxConfig);

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music").withKey(key);

        // Java Futures
        Future<GetItemResult> call = client.getItemAsync(request);
        while (!call.isDone()) {
            // Do other processing while you're waiting for the response
            System.out.println("Doing something else for a few seconds...");
            Thread.sleep(3000);
        }
        // The results should be ready by now

        try {
            call.get();

        } catch (ExecutionException ee) {
            // Futures always wrap errors as an ExecutionException.
            // The *real* exception is stored as the cause of the
            // ExecutionException
            Throwable exception = ee.getCause();
            System.out.println("Error getting item: " + exception.getMessage());
        }

        // Async callbacks
        call = client.getItemAsync(request, new AsyncHandler<GetItemRequest, GetItemResult>()
{
    @Override
```

```
public void onSuccess(GetItemRequest request, GetItemResult getItemResult) {
    System.out.println("Result: " + getItemResult);
}

@Override
public void onError(Exception e) {
    System.out.println("Unable to read item");
    System.err.println(e.getMessage());
    // Callers can also test if exception is an instance of
    // AmazonServiceException or AmazonClientException and cast
    // it to get additional information
}

});

call.get();

}
}
```

Querying global secondary indexes with SDK for Java 1.x

You can use Amazon DynamoDB Accelerator (DAX) to query [global secondary indexes](#) using DynamoDB [programmatic interfaces](#).

The following example demonstrates how to use DAX to query the CreateDateIndex global secondary index that is created in [Example: Global secondary indexes using the AWS SDK for Java document API](#).

The DAXClient class instantiates the client objects that are needed to interact with the DynamoDB programming interfaces.

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.util.EC2MetadataUtils;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;

public class DaxClient {

    private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

    DynamoDB getDaxDocClient(String daxEndpoint) {
```

```
System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
AmazonDynamoDB client = daxClientBuilder.build();

return new DynamoDB(client);
}

DynamoDBMapper getDaxMapperClient(String daxEndpoint) {
    System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
    AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

    daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
    AmazonDynamoDB client = daxClientBuilder.build();

    return new DynamoDBMapper(client);
}
}
```

You can query a global secondary index in the following ways:

- Use the `queryIndex` method on the `QueryIndexDax` class defined in the following example. The `QueryIndexDax` takes as a parameter the `client` object that is returned by the `getDaxDocClient` method on the `DaxClient` class.
- If you are using the [object persistence interface](#), use the `queryIndexMapper` method on the `QueryIndexDax` class defined in the following example. The `queryIndexMapper` takes as a parameter the `client` object that is returned by the `getDaxMapperClient` method defined on the `DaxClient` class.

```
import java.util.Iterator;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import java.util.List;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBQueryExpression;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import java.util.HashMap;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
```

```
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;

public class QueryIndexDax {

    //This is used to query Index using the low-level interface.
    public static void queryIndex(DynamoDB client, String tableName, String indexName) {
        Table table = client.getTable(tableName);

        System.out.println("\n*****");
        System.out.print("Querying index " + indexName + "...");

        Index index = table.getIndex(indexName);

        ItemCollection<QueryOutcome> items = null;

        QuerySpec querySpec = new QuerySpec();

        if (indexName == "CreateDateIndex") {
            System.out.println("Issues filed on 2013-11-01");
            querySpec.withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
                .WithValueMap(new ValueMap().withString(":v_date",
"2013-11-01").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        } else {
            System.out.println("\nNo valid index name provided");
            return;
        }

        Iterator<Item> iterator = items.iterator();

        System.out.println("Query: printing results...");

        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }

    }

    //This is used to query Index using the high-level mapper interface.
```

```
public static void queryIndexMapper(DynamoDBMapper mapper, String tableName, String indexName) {
    HashMap<String,AttributeValue> eav = new HashMap<String,AttributeValue>();
    eav.put(":v_date", new AttributeValue().withS("2013-11-01"));
    eav.put(":v_issue", new AttributeValue().withS("A-"));
    DynamoDBQueryExpression<CreateDate> queryExpression = new
    DynamoDBQueryExpression<CreateDate>()
        .withIndexName("CreateDateIndex").withConsistentRead(false)
        .withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
        .withExpressionAttributeValues(eav);

    List<CreateDate> items = mapper.query(CreateDate.class, queryExpression);
    Iterator<CreateDate> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        CreateDate iterObj = iterator.next();
        System.out.println(iterObj.getCreateDate());
        System.out.println(iterObj.getIssueId());
    }
}
```

The class definition below represents the Issues table and is used in the queryIndexMapper method.

```
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIndexHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIndexRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;

@DynamoDBTable(tableName = "Issues")
public class CreateDate {
    private String createDate;
    @DynamoDBHashKey(attributeName = "IssueId")
    private String issueId;

    @DynamoDBIndexHashKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
    "CreateDate")
    public String getCreateDate() {
        return createDate;
```

```
}

public void setCreateDate(String createDate) {
    this.createDate = createDate;
}

@DynamoDBIndexRangeKey(globalSecondaryIndexName = "CreateDatePicker", attributeName =
"IssueId")
public String getIssueId() {
    return issueId;
}

public void setIssueId(String issueId) {
    this.issueId = issueId;
}
}
```

Document history for DynamoDB

The following table describes the important changes in each release of the *DynamoDB Developer Guide* from July 3, 2018 onward. For notification about updates to this documentation, you can subscribe to the RSS feed (at the top left corner of this page).

Change	Description	Date
<u>Resource-based policies for Amazon DynamoDB resources</u>	DynamoDB now supports resource-based policies for tables, indexes, and streams. Resource-based policies let you define access permissions by specifying who has access to each resource, and the actions they are allowed to perform on each resource. For more information, see <u>Using resource-based policies for DynamoDB</u> .	March 20, 2024
<u>DynamoDB managed policy update</u>	Added a new permission dynamodb:GetResourcePolicy to the AmazonDynamoDBReadOnlyAccess managed policy. This permission provides access to read resource-based policies attached to DynamoDB resources. For more information, see <u>AWS managed policy: AmazonDynamoDBReadOnlyAccess</u> .	March 20, 2024
<u>AWS PrivateLink for Amazon DynamoDB</u>	Amazon DynamoDB now supports AWS PrivateLi	March 19, 2024

nk. With AWS PrivateLink, you can simplify private network connectivity between virtual private clouds (VPCs), DynamoDB, and your on-premises data centers using interface VPC endpoints and private IP addresses. For more information, see [AWS PrivateLink for DynamoDB](#).

[Programming with JavaScript guide](#)

Amazon DynamoDB presents a programming guide for AWS SDK for JavaScript. Learn about the AWS SDK for JavaScript, abstraction layers, configuring connection, handling errors, defining retry policies, managing keep-alive, and more. For more information, see [Programming with JavaScript](#).

[Programming with AWS SDK for Java 2.x guide](#)

Created a new programming guide that goes in depth about high-level, low-level, and document interfaces, HTTP clients and their configuration, error handling, and addresses the most common configuration settings that you should consider when using the SDK for Java 2.x. For more information, see [Programming Amazon DynamoDB with AWS SDK for Java 2.x](#).

<u>Clone tables with NoSQL Workbench</u>	Allow developers to use NoSQL Workbench to copy or clone tables between development environments and regions (DynamoDB Local and DynamoDB web). For more information, see <u>Cloning tables with NoSQL Workbench</u> .	February 26, 2024
<u>Programming with Python guide</u>	Created a new guide that goes in depth about both high level and low level libraries and addresses the most common configuration settings that one should consider when using the Python SDK. For more information, see <u>Programming with Python</u> .	January 5, 2024
<u>Time to live (TTL) topic rewrite</u>	Completely rewrote the TTL section of the guide. The new guide helps you get started with TTL by providing ready-to-use code snippets along the way. The current code snippets provided are in Python and Javascript. For more information, see <u>TTL</u> .	December 20, 2023
<u>Best Practices for Understanding your AWS Billing and Usage Reports</u>	Added a new section that clarifies various usage types and the charges for those usage types in DynamoDB. For more information, see <u>Billing and usage reports</u> .	December 15, 2023

<u>Amazon DynamoDB zero-ETL integration with Amazon OpenSearch Service</u>	Amazon DynamoDB now supports zero-ETL integration with Amazon OpenSearch Service, which lets you perform a search on your DynamoDB data by automatically replicating and transforming it without custom code or infrastructure. For more information, see <u>DynamoDB zero-ETL integration with Amazon OpenSearch Service</u> .	November 28, 2023
<u>Migrating to DynamoDB from a relational database</u>	Created a <u>migration guide</u> to help users understand how to migrate to DynamoDB from a relational database.	November 27, 2023
<u>Generate sample data with NoSQL Workbench</u>	NoSQL Workbench for Amazon DynamoDB now supports creating data models directly from <u>sample data model templates</u> to help you design data schemas for your workloads. You can use this feature to get familiar with NoSQL data modeling best practices when building your applications on DynamoDB.	September 28, 2023

[Incremental Export to S3](#)

You can now export data that was inserted, updated or deleted, in small increments. With [incremental export](#), you can export changed data ranging from a few megabytes to terabytes with a few clicks in the AWS Management Console, an API call, or the AWS Command Line Interface.

September 26, 2023

[Data modeling for DynamoDB](#)

You can now learn more about [data modeling](#) with DynamoDB examples that focus on specific use cases, their access patterns, and step-by-step guidance in realizing those access patterns.

July 14, 2023

[Troubleshooting section](#)

You can now find [troubleshooting content](#) for latency and throttling issues that might occur in your DynamoDB tables.

March 13, 2023

[Deletion protection for Amazon DynamoDB](#)

Deletion protection is now available for Amazon DynamoDB tables in all AWS Regions. DynamoDB now makes it possible for you to protect your tables from accidental deletion when performing regular table management operations.

March 8, 2023

<u>AWS CloudFormation support for KDSD in global tables</u>	Amazon Kinesis Data Streams for DynamoDB now supports AWS CloudFormation for DynamoDB global tables, which means you can enable streaming to an Amazon Kinesis Data Streams on your DynamoDB global tables with CloudFormation templates.	February 15, 2023
<u>DynamoDB local supports 100 actions per transaction</u>	You can now perform up to 100 actions in a single transaction on DynamoDB local.	February 9, 2023
<u>Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload</u>	You can now use the <u>DynamoDB Well-Architected Lens</u> , a collection of design principles and guidance that you can use for designing well-architected DynamoDB workloads.	February 3, 2023
<u>PartiQL GovCloud availability</u>	<u>PartiQL—a SQL-compatible query language for Amazon DynamoDB</u> is now supported in AWS GovCloud (US-East) and AWS GovCloud (US-West)	December 21, 2022
<u>Single installation suite for NoSQL Workbench and DynamoDB local</u>	<u>NoSQL Workbench for DynamoDB</u> now includes a guided <u>DynamoDB local</u> installation process to streamline setting up your DynamoDB local development environment.	December 6, 2022

[Bulk import from S3](#)

Amazon DynamoDB now makes it easier for you to migrate and load data into new DynamoDB tables by [supporting bulk data imports from Amazon S3](#).

August 18, 2022

[Enhanced integration with Service Quotas](#)

[Service Quotas](#) now enables you to proactively manage your account and table quotas. You can view current values, set alarms for when your utilization of a quota exceeds a configurable threshold, and more.

June 15, 2022

[NoSQL Workbench adds table and GSI support](#)

You can now use NoSQL Workbench for table and global secondary index (GSI) [control plane operations](#) such as CreateTable, UpdateTable, and DeleteTable.

June 2, 2022

[Standard-infrequent access table class now available in China](#)

Amazon DynamoDB Standard-Infrequent Access table class is available in China Regions. Reduce your [DynamoDB costs by up to 60 percent](#), by using this new table class for tables that store infrequently accessed data.

April 18, 2022

<u>Increase in default service quotas and table management operations</u>	<u>DynamoDB increased the default quota for the number of tables per account and Region</u> from 256 to 2,500 tables, and increased the number of concurrent table management operations from 50 to 500.	March 9, 2022
<u>Optional limiting of items with PartiQL for DynamoDB</u>	DynamoDB can <u>limit the number of items processed in PartiQL</u> for DynamoDB operations as an optional parameter on each request.	March 8, 2022
<u>AWS Backup integration available in China (Beijing and Ningxia) Regions</u>	<u>AWS Backup</u> now integrates with DynamoDB in the China (Beijing and Ningxia) Regions. You can meet compliance and business continuity requirements more easily through enhanced backup features in AWS Backup, such as cross-account and cross-Region backups.	January 26, 2022
<u>Throughput capacity information through PartiQL API calls</u>	DynamoDB can return the throughput capacity consumed by <u>PartiQL API</u> calls to help you optimize your queries and throughput costs.	January 18, 2022

<u>AWS Backup integration</u>	DynamoDB now helps you meet compliance and business continuity requirements more easily through enhanced backup features in <u>AWS Backup</u> , such as cross-account and cross-Region backups.	November 24, 2021
<u>NoSQL Workbench import/export datasets in CSV</u>	<u>NoSQL Workbench for Amazon DynamoDB</u> now enables you to import and automatically populate sample data to help build and visualize your data models.	October 11, 2021
<u>Filter and retrieve Amazon DynamoDB Streams data-plane activity with AWS CloudTrail</u>	Amazon DynamoDB now provides you more granular control of audit logging by enabling you to <u>filter Streams data-plane API activity in AWS CloudTrail</u> .	September 22, 2021
<u>Updated console</u>	The <u>DynamoDB console</u> is now your default console to help you manage data more easily, simplify your common tasks, and give you faster access to resources and features.	August 25, 2021

[DAX SDK For Java 2.x is now available](#)

[DynamoDB Accelerator \(DAX\) SDK for Java 2.x](#) is now available and is compatible with the AWS SDK for Java 2.x. You can benefit from the latest features, including non-blocking I/O.

July 29, 2021

[NoSQL Workbench feature updates including control plane operations](#)

[NoSQL Workbench for Amazon DynamoDB](#) now helps you run frequent operations more easily to modify and access table data.

July 28, 2021

[DynamoDB global tables are now available in the Asia Pacific Region](#)

[DynamoDB global tables](#) are now available in the Asia Pacific (Osaka) Region. Replicate your DynamoDB tables automatically across your choice of 22 AWS Regions.

July 28, 2021

[DAX is now available in China](#)

[DynamoDB Accelerator \(DAX\)](#) is now available in the China (Beijing) Region, operated by Sinnet.

July 28, 2021

[DAX encryption in transit](#)

[DynamoDB Accelerator \(DAX\)](#) now supports encryption in transit of data between your applications and DAX clusters, and between the nodes within a DAX cluster.

July 24, 2021

[CloudFormation and CloudTrail integration](#)[Integration with AWS](#)

June 18, 2021

[CloudFormation](#) and security enhancements with CloudFormation data-plane logging.

[CloudFormation now supported for global tables](#)[Amazon DynamoDB global](#)

May 14, 2021

tables now support [AWS CloudFormation](#), which means you can create global tables and manage their settings with CloudFormation templates.

[Amazon DynamoDB local support for Java 2.x](#)[You now can use the AWS](#)

May 3, 2021

[SDK for Java 2.x](#) with [DynamoDB local](#), the downloadable version of Amazon DynamoDB. With DynamoDB local, you can develop and test applications by using a version of DynamoDB running in your local development environment without incurring any additional costs.

[NoSQL Workbench now supports AWS CloudFormation](#)[NoSQL Workbench for](#)

April 22, 2021

[Amazon DynamoDB](#) now supports [AWS CloudFormation](#), so you can manage and modify DynamoDB data models with CloudFormation templates. In addition, you now can configure table capacity settings in NoSQL Workbench.

[DynamoDB and AWS Amplify now feature integration](#)

[AWS Amplify](#) now orchestrates multiple DynamoDB global secondary index updates in a single deployment.

April 20, 2021

[AWS CloudTrail to log Amazon DynamoDB Streams data-plane APIs](#)

You now can use [AWS CloudTrail](#) to log [Amazon DynamoDB Streams](#) data-plane API activity, and monitor and investigate item-level changes in your DynamoDB tables.

April 20, 2021

[Amazon Kinesis Data Streams for Amazon DynamoDB now supports AWS CloudFormation](#)

[Amazon Kinesis Data Streams](#) for [Amazon DynamoDB](#) now supports AWS CloudFormation, which means you can enable streaming to an Amazon Kinesis data stream on your DynamoDB tables with CloudFormation templates. By streaming your DynamoDB data changes to a Kinesis data stream, you can build advanced streaming applications with AAmazon Kinesis services.

April 12, 2021

[Amazon Keyspaces now offers FIPS 140-2 compliant endpoints](#)

[Amazon Keyspaces \(for Apache Cassandra\) now offers Federal Information Processing Standards \(FIPS\) 140-2 compliant endpoints](#) to help you run highly regulated workloads more easily. FIPS 140-2 is a US and Canadian government standard that specifies the security requirements for cryptographic modules that protect sensitive information.

April 8, 2021

[Amazon EC2 T3 instances for DAX](#)

DAX now supports [Amazon EC2 T3 instance types](#), which provide a baseline level of CPU performance with the ability to burst above the baseline when needed.

February 15, 2021

[NoSQL Workbench for Amazon DynamoDB support for PartiQL](#)

You now can use the [NoSQL Workbench for DynamoDB](#) to build [PartiQL](#) statements for DynamoDB.

December 4, 2020

[PartiQL for DynamoDB](#)

You now can use [PartiQL for DynamoDB](#)—a SQL-compatible query language—to interact with DynamoDB tables and run ad hoc queries by using the AWS Management Console, AWS Command Line Interface, and DynamoDB APIs for PartiQL.

November 23, 2020

<u>Amazon Kinesis Data Streams for Amazon DynamoDB</u>	You now can use <u>Amazon Kinesis Data Streams for Amazon DynamoDB</u> with your DynamoDB tables to capture item-level changes and replicate them to a Kinesis data stream.	November 23, 2020
<u>DynamoDB table export</u>	You can now <u>export your DynamoDB tables to Amazon S3</u> , enabling you to perform analytics and complex queries on your data with services like Athena, AWS Glue, and Lake Formation.	November 9, 2020
<u>Support for empty values</u>	DynamoDB now supports empty values for non-key String and Binary attributes in DynamoDB tables. Empty value support gives you greater flexibility to use attributes for a broader set of use cases without having to transform such attributes before sending them to DynamoDB. List, Map, and Set data types also support empty String and Binary values.	May 18, 2020
<u>NoSQL Workbench for Amazon DynamoDB support for Linux</u>	NoSQL Workbench for Amazon DynamoDB is now supported on <u>Linux- Ubuntu, Fedora and Debian</u> .	May 4, 2020

[CloudWatch Contributor Insights for DynamoDB – GA](#)

[CloudWatch Contributor Insights for DynamoDB](#)

April 2, 2020

is generally available.
CloudWatch Contributor Insights for DynamoDB is a diagnostic tool that provides an at-a-glance view of your DynamoDB table's traffic trends and helps you identify your table's most frequently accessed keys (also known as hot keys).

[Upgrading global tables](#)

You now can update your global tables from version 2017.11.29 to the [latest version of global tables \(2019.11.21\)](#), with a few clicks in the DynamoDB Console. By upgrading the version of your global tables, you can increase the availability of your DynamoDB tables easily by extending your existing tables into additional AWS Regions, with no table rebuilds required.

March 16, 2020

[NoSQL Workbench for Amazon DynamoDB – GA](#)

[NoSQL Workbench for Amazon DynamoDB](#) is generally available. Use the NoSQL Workbench to design, create, query, and manage DynamoDB tables.

March 2, 2020

<u>DAX cache cluster metrics</u>	DAX support for new <u>CloudWatch metrics</u> , that allow you to better understand your DAX cluster's performance.	February 6, 2020
<u>CloudWatch Contributor Insights for DynamoDB – Preview</u>	<u>CloudWatch Contributor Insights for DynamoDB</u> is a diagnostic tool that provides an at-a-glance view of your DynamoDB table's traffic trends and helps you identify your table's most frequently accessed keys (also known as hot keys).	November 26, 2019
<u>Adaptive capacity support for imbalanced workload</u>	Amazon DynamoDB adaptive capacity now <u>handles</u> imbalanced workloads better by isolating frequently accessed items automatically. If your application drives disproportionately high traffic to one or more items, DynamoDB will rebalance your partitions such that frequently accessed items do not reside on the same partition.	November 26, 2019
<u>Support for customer managed keys</u>	DynamoDB now <u>supports customer managed keys</u> , which means you can have full control over how you encrypt and manage the security of your DynamoDB data.	November 25, 2019

<u>NoSQL Workbench support for DynamoDB local (Downloadable Version)</u>	The NoSQL Workbench now supports connecting to <u>DynamoDB local (Downloadable Version)</u> to design, create, query, and manage DynamoDB tables.	November 8, 2019
<u>NoSQL Workbench - Preview</u>	This is the initial release of NoSQL Workbench for DynamoDB. Use NoSQL Workbench to design, create, query, and manage DynamoDB tables. For more information, see <u>NoSQL Workbench for Amazon DynamoDB (Preview)</u> .	September 16, 2019
<u>DAX adds support for transactional operations using Python and .NET</u>	DAX supports the <code>TransactWriteItems</code> and <code>TransactGetItems</code> APIs for applications written in Go, Java, .NET, Node.js, and Python. For more information, see <u>In-Memory Acceleration with DAX</u> .	February 14, 2019

<u>Amazon DynamoDB local (Downloadable Version) Updates</u>	DynamoDB local (Downloadable Version) now supports transactional APIs, on-demand read/write capacity, capacity reporting for read and write operations, and 20 global secondary indexes. For more information, see <u>Differences Between Downloadable DynamoDB and the DynamoDB Web Service</u> .	February 4, 2019
<u>Amazon DynamoDB On-Demand</u>	DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use. For more information, see <u>Read/Write Capacity Mode</u> .	November 28, 2018
<u>Amazon DynamoDB Transactions</u>	DynamoDB transactions make coordinated, all-or-nothing changes to multiple items both within and across tables, providing atomicity, consistency, isolation, and durability (ACID) in DynamoDB. For more information, see <u>Amazon DynamoDB Transactions</u> .	November 27, 2018

[Amazon DynamoDB encrypts all customer data at rest](#)

DynamoDB encryption at rest provides an additional layer of data protection by securing your data in the encrypted table, including its primary key, local and global secondary indexes, streams, global tables, backups, and DAX clusters whenever the data is stored in durable media. For more information, see [Amazon DynamoDB Encryption at Rest](#).

[Use Amazon DynamoDB Local More Easily with the New Docker Image](#)

Now, it's easier to use DynamoDB local, the downloadable version of DynamoDB, to help you develop and test your DynamoDB applications by using the new DynamoDB local Docker image. For more information, see [DynamoDB \(Downloadable Version\) and Docker](#).

November 15, 2018

August 22, 2018

<u>DynamoDB Accelerator (DAX)</u>	DynamoDB Accelerator (DAX) now supports encryption at rest for new DAX clusters to help you accelerate reads from Amazon DynamoDB tables in security-sensitive applications that are subject to strict compliance and regulatory requirements. For more information, see <u>DAX Encryption at Rest</u> .	August 9, 2018
<u>DynamoDB point-in-time recovery (PITR) adds support for restoring deleted tables</u>	If you delete a table with point-in-time recovery enabled, a system backup is automatically created and is retained for 35 days (at no additional cost). For more information, see <u>Before You Begin Using Point In Time Recovery</u> .	August 7, 2018
<u>Updates now available over RSS</u>	You can now subscribe to the <u>RSS feed</u> (at the top left corner of this page) to receive notifications about updates to the Amazon DynamoDB Developer Guide.	July 3, 2018

Earlier updates

The following table describes important changes of the *DynamoDB Developer Guide* before July 3, 2018.

Change	Description	Date Changed
Go support for DAX	Now, you can enable microsecond read performance for Amazon DynamoDB tables in your applications written in the Go programming language by using the new DynamoDB Accelerator (DAX) SDK for Go. For more information, see DAX SDK for Go .	June 26, 2018
DynamoDB announces SLA	DynamoDB has released a public availability SLA. For more information, see Amazon DynamoDB Service Level Agreement .	June 19, 2018
DynamoDB continuous backups and Point-In-Time Recovery (PITR)	Point-in-time recovery helps protect your Amazon DynamoDB tables from accidental write or delete operations. With point in time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. DynamoDB maintains incremental backups of your	April 25, 2018

Change	Description	Date Changed
	<p>table. For more information, see Point-in-time recovery for DynamoDB.</p>	
Encryption at rest for DynamoDB	<p>DynamoDB encryption at rest, available for new DynamoDB tables, helps you secure your application data in Amazon DynamoDB tables by using AWS-managed encryption keys stored in AWS Key Management Service. For more information, see DynamoDB encryption at rest.</p>	February 8, 2018
DynamoDB Backup and restore	<p>On-Demand Backup allows you to create full backups of your DynamoDB tables data for data archival, helping you meet your corporate and governmental regulatory requirements. You can backup tables from a few megabytes to hundreds of terabytes of data, with no impact on performance and availability to your production applications. For more information, see Using On-Demand backup and restore for DynamoDB.</p>	November 29, 2017

Change	Description	Date Changed
DynamoDB Global tables	<p>Global Tables builds upon DynamoDB's global footprint to provide you with a fully managed, multi-region, and multi-active database that provides fast, local, read and write performance for massively scaled, global applications. Global Tables replicates your Amazon DynamoDB tables automatically across your choice of AWS regions. For more information, see Global tables - multi-Region replication for DynamoDB.</p>	November 29, 2017
Node.js support for DAX	<p>Node.js developers can leverage DynamoDB Accelerator (DAX), using the DAX client for Node.js. For more information, see In-memory acceleration with DynamoDB Accelerator (DAX).</p>	October 5, 2017

Change	Description	Date Changed
VPC Endpoints for DynamoDB	<p>DynamoDB endpoints allow Amazon EC2 instances in your Amazon VPC to access DynamoDB, without exposure to the public Internet.</p> <p>Network traffic between your VPC and DynamoDB does not leave the Amazon network.</p> <p>For more information, see Using Amazon VPC endpoints to access DynamoDB.</p>	August 16, 2017
Auto Scaling for DynamoDB	<p>DynamoDB auto scaling eliminates the need for manually defining or adjusting provisioned throughput settings. Instead, DynamoDB auto scaling dynamically adjusts read and write capacity in response to actual traffic patterns. This allows a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling.</p> <p>When the workload decreases, DynamoDB auto scaling decreases the provisioned capacity. For more information, see Managing throughput capacity automatically with DynamoDB auto scaling.</p>	June 14, 2017

Change	Description	Date Changed
DynamoDB Accelerator (DAX)	DynamoDB Accelerator (DAX) is a fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement – from milliseconds to microseconds – even at millions of requests per second. For more information, see In-memory acceleration with DynamoDB Accelerator (DAX) .	April 19, 2017
DynamoDB now supports automatic item expiration with Time to Live (TTL)	Amazon DynamoDB Time to Live (TTL) enables you to automatically delete expired items from your tables, at no additional cost. For more information, see Time to Live (TTL) .	Feb 27, 2017
DynamoDB now supports Cost Allocation Tags	You can now add tags to your Amazon DynamoDB tables for improved usage categorization and more granular cost reporting. For more information, see Adding tags and labels to resources .	Jan 19, 2017

Change	Description	Date Changed
New DynamoDB <code>DescribeLimits</code> API	<p>The <code>DescribeLimits</code> API returns the current provisioned capacity limits for your AWS account in a region, both for the region as a whole and for any one DynamoDB table that you create there. It lets you determine what your current account-level limits are so that you can compare them to the provisioned capacity that you are currently using, and have plenty of time to apply for an increase before you hit a limit.</p> <p>For more information, see Service, account, and table quotas in Amazon DynamoDB and the DescribeLimits in the <i>Amazon DynamoDB API Reference</i>.</p>	March 1, 2016

Change	Description	Date Changed
DynamoDB Console Update and New Terminology for Primary Key Attributes	<p>The DynamoDB management console has been redesigned to be more intuitive and easy to use. As part of this update, we are introducing new terminology for primary key attributes:</p> <ul style="list-style-type: none">• Partition Key—also known as a <i>hash attribute</i>.• Sort Key—also known as a <i>range attribute</i>. <p>Only the names have changed; the functionality remains the same.</p> <p>When you create a table or a secondary index, you can choose either a simple primary key (partition key only), or a composite primary key (partition key and sort key). The DynamoDB documentation has been updated to reflect these changes.</p>	November 12, 2015

Change	Description	Date Changed
Amazon DynamoDB Storage Backend for Titan	<p>The DynamoDB Storage Backend for Titan is a storage backend for the Titan graph database implemented on top of Amazon DynamoDB. When using the DynamoDB Storage Backend for Titan, your data benefits from the protection of DynamoDB, which runs across Amazon's high-availability data centers. The plugin is available for Titan version 0.4.4 (primarily for compatibility with existing applications) and Titan version 0.5.4 (recommended for new applications). Like other storage backends for Titan, this plugin supports the Tinkerpop stack (versions 2.4 and 2.5), including the Blueprints API and the Gremlin shell. For more information, see Amazon DynamoDB Storage Backend for Titan.</p>	August 20, 2015

Change	Description	Date Changed
DynamoDB Streams, Cross-Region Replication, and Scan with Strongly Consistent Reads	<p>DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time. For more information, see Change data capture for DynamoDB Streams and the DynamoDB Streams API Reference.</p> <p>DynamoDB cross-region replication is a client-side solution for maintaining identical copies of DynamoDB tables across different AWS regions, in near real time. You can use cross region replication to back up DynamoDB tables, or to provide low-latency access to data where users are geographically distributed.</p> <p>The DynamoDB Scan operation uses eventually consistent reads, by default. You can use strongly consistent reads instead by setting the Consisten</p>	July 16, 2015

Change	Description	Date Changed
	<p>Set the <code>Read</code> parameter to true.</p> <p>For more information, see Read consistency for scan and Scan in the Amazon DynamoDB API Reference.</p>	
AWS CloudTrail support for Amazon DynamoDB	<p>DynamoDB is now integrated with CloudTrail. CloudTrail captures API calls made from the DynamoDB console or from the DynamoDB API and tracks them in log files.</p> <p>For more information, see Logging DynamoDB operations by using AWS CloudTrail and the AWS CloudTrail User Guide.</p>	May 28, 2015

Change	Description	Date Changed
Improved support for Query expressions	<p>This release adds a new <code>KeyConditionExpression</code> parameter to the Query API. A Query reads items from a table or an index using primary key values. The <code>KeyConditionExpression</code> parameter is a string that identifies primary key names, and conditions to be applied to the key values; the Query retrieves only those items that satisfy the expression. The syntax of <code>KeyConditionExpression</code> is similar to that of other expression parameters in DynamoDB, and allows you to define substitution variables for names and values within the expression. For more information, see Query operations in DynamoDB.</p>	April 27, 2015

Change	Description	Date Changed
New comparison functions for conditional writes	<p>In DynamoDB, the <code>ConditionExpression</code> parameter determines whether a <code>PutItem</code>, <code>UpdateItem</code>, or <code>DeleteItem</code> succeeds: The item is written only if the condition evaluates to true. This release adds two new functions, <code>attribute_type</code> and <code>size</code>, for use with <code>ConditionExpression</code>. These functions allow you to perform a conditional writes based on the data type or size of an attribute in a table. For more information, see Condition expressions.</p>	April 27, 2015

Change	Description	Date Changed
Scan API for secondary indexes	<p>In DynamoDB, a Scan operation reads all of the items in a table, applies user-defined filtering criteria, and returns the selected data items to the application. This same capability is now available for secondary indexes too. To scan a local secondary index or a global secondary index, you specify the index name and the name of its parent table. By default, an index Scan returns all of the data in the index; you can use a filter expression to narrow the results that are returned to the application. For more information, see Working with scans in DynamoDB.</p>	February 10, 2015

Change	Description	Date Changed
Online operations for global secondary indexes	<p>Online indexing lets you add or remove global secondary indexes on existing tables. With online indexing, you do not need to define all of a table's indexes when you create a table; instead, you can add a new index at any time. Similarly, if you decide you no longer need an index, you can remove it at any time. Online indexing operations are non-blocking, so that the table remains available for read and write activity while indexes are being added or removed. For more information, see Managing Global Secondary Indexes.</p>	January 27, 2015

Change	Description	Date Changed
Document model support with JSON	<p>DynamoDB allows you to store and retrieve documents with full support for document models. New data types are fully compatible with the JSON standard and allow you to nest document elements within one another. You can use document path dereference operators to read and write individual elements, without having to retrieve the entire document. This release also introduces new expression parameters for specifying projections, conditions and update actions when reading or writing data items. To learn more about document model support with JSON, see Data types and Using expressions in DynamoDB.</p>	October 7, 2014
Flexible scaling	<p>For tables and global secondary indexes, you can increase provisioned read and write throughput capacity by any amount, provided that you stay within your per-table and per-account limits. For more information, see Service, account, and table quotas in Amazon DynamoDB.</p>	October 7, 2014

Change	Description	Date Changed
Larger item sizes	<p>The maximum item size in DynamoDB has increased from 64 KB to 400 KB.</p> <p>For more information, see Service, account, and table quotas in Amazon DynamoDB.</p>	October 7, 2014

Change	Description	Date Changed
Improved conditional expressions	<p>DynamoDB expands the operators that are available for conditional expressions, giving you additional flexibility for conditional puts, updates, and deletes. The newly available operators let you check whether an attribute does or does not exist, is greater than or less than a particular value, is between two values, begins with certain characters, and much more. DynamoDB also provides an optional <i>OR</i> operator for evaluating multiple conditions. By default, multiple conditions in an expression are ANDed together, so the expression is true only if all of its conditions are true. If you specify <i>OR</i> instead, the expression is true if one or more one conditions are true. For more information, see Working with items and attributes.</p>	April 24, 2014

Change	Description	Date Changed
Query filter	<p>The DynamoDB Query API supports a new <code>QueryFilter</code> option. By default, a Query finds items that match a specific partition key value and an optional sort key condition. A Query filter applies conditional expressions to other, non-key attributes; if a Query filter is present, then items that do not match the filter conditions are discarded before the Query results are returned to the application. For more information, see Query operations in DynamoDB.</p>	April 24, 2014

Change	Description	Date Changed
Data export and import using the AWS Management Console	<p>The DynamoDB console has been enhanced to simplify exports and imports of data in DynamoDB tables. With just a few clicks, you can set up an AWS Data Pipeline to orchestrate the workflow, and an Amazon Elastic MapReduce cluster to copy data from DynamoDB tables to an Amazon S3 bucket, or vice-versa. You can perform an export or import one time only, or set up a daily export job. You can even perform cross-region exports and imports, copying DynamoDB data from a table in one AWS region to a table in another AWS region. For more information, see Exporting and importing DynamoDB data using AWS Data Pipeline.</p>	March 6, 2014

Change	Description	Date Changed
Reorganized higher-level API documentation	<p>Information about the following APIs is now easier to find:</p> <ul style="list-style-type: none">• Java: DynamoDBMapper• .NET: Document model and object-persistence model <p>These higher-level APIs are now documented here: <u>Higher-level programming interfaces for DynamoDB.</u></p>	January 20, 2014

Change	Description	Date Changed
Global secondary indexes	DynamoDB adds support for global secondary indexes. As with a local secondary index, you define a global secondary index by using an alternate key from a table and then issuing Query requests on the index. Unlike a local secondary index, the partition key for the global secondary index does not have to be the same as that of the table; it can be any scalar attribute from the table. The sort key is optional and can also be any scalar table attribute. A global secondary index also has its own provisioned throughput settings, which are separate from those of the parent table. For more information, see Improving data access with secondary indexes and Using Global Secondary Indexes in DynamoDB .	December 12, 2013

Change	Description	Date Changed
Fine-grained access control	<p>DynamoDB adds support for fine-grained access control. This feature allows customers to specify which principals (users, groups, or roles) can access individual items and attributes in a DynamoDB table or secondary index. Applications can also leverage web identity federation to offload the task of user authentication to a third-party identity provider, such as Facebook, Google, or Login with Amazon. In this way, applications (including mobile apps) can handle very large numbers of users, while ensuring that no one can access DynamoDB data items unless they are authorized to do so. For more information, see Using IAM policy conditions for fine-grained access control.</p>	October 29, 2013

Change	Description	Date Changed
4 KB read capacity unit size	<p>The capacity unit size for reads has increased from 1 KB to 4 KB. This enhancement can reduce the number of provisioned read capacity units required for many applications. For example, prior to this release, reading a 10 KB item would consume 10 read capacity units; now that same 10 KB read would consume only 3 units (10 KB / 4 KB, rounded up to the next 4 KB boundary). For more information, see Read/write capacity mode.</p>	May 14, 2013
Parallel scans	<p>DynamoDB adds support for parallel Scan operations. Applications can now divide a table into logical segments and scan all of the segments simultaneously. This feature reduces the time required for a Scan to complete, and fully utilizes a table's provisioned read capacity. For more information, see Working with scans in DynamoDB.</p>	May 14, 2013

Change	Description	Date Changed
Local secondary indexes	<p>DynamoDB adds support for local secondary indexes. You can define sort key indexes on non-key attributes, and then use these indexes in Query requests. With local secondary indexes, applications can efficiently retrieve data items across multiple dimensions. For more information, see Local Secondary Indexes.</p>	April 18, 2013
New API version	<p>With this release, DynamoDB introduces a new API version (2012-08-10). The previous API version (2011-12-05) is still supported for backward compatibility with existing applications. New applications should use the new API version 2012-08-10. We recommend that you migrate your existing applications to API version 2012-08-10, since new DynamoDB features (such as local secondary indexes) will not be backported to the previous API version. For more information on API version 2012-08-10, see the Amazon DynamoDB API Reference.</p>	April 18, 2013

Change	Description	Date Changed
IAM policy variable support	<p>The IAM access policy language now supports variables. When a policy is evaluated, any policy variables are replaced with values that are supplied by context-based information from the authenticated user's session. You can use policy variables to define general purpose policies without explicitly listing all the components of the policy.</p> <p>For more information about policy variables, go to Policy Variables in the <i>AWS Identity and Access Management Using IAM</i> guide.</p> <p>For examples of policy variables in DynamoDB, see Identity and Access Management for Amazon DynamoDB.</p>	April 4, 2013
PHP code examples updated for AWS SDK for PHP version 2	<p>Version 2 of the AWS SDK for PHP is now available. The PHP code examples in the Amazon DynamoDB Developer Guide have been updated to use this new SDK.</p> <p>For more information on Version 2 of the SDK, see AWS SDK for PHP.</p>	January 23, 2013

Change	Description	Date Changed
New endpoint	DynamoDB expands to the AWS GovCloud (US-West) region. For the current list of service endpoints and protocols, see Regions and Endpoints .	December 3, 2012
New endpoint	DynamoDB expands to the South America (São Paulo) region. For the current list of supported endpoints, see Regions and Endpoints .	December 3, 2012
New endpoint	DynamoDB expands to the Asia Pacific (Sydney) region. For the current list of supported endpoints, see Regions and Endpoints .	November 13, 2012

Change	Description	Date Changed
DynamoDB implements support for CRC32 checksums, supports strongly consistent batch gets, and removes restrictions on concurrent table updates.	<ul style="list-style-type: none">DynamoDB calculates a CRC32 checksum of the HTTP payload and returns this checksum in a new header, <code>x-amz-crc32</code>. For more information, see DynamoDB low-level API.By default, read operations performed by the <code>BatchGetItem</code> API are eventually consistent. A new <code>ConsistentRead</code> parameter in <code>BatchGetItem</code> lets you choose strong read consistency instead, for any table(s) in the request. For more information, see Description.This release removes some restrictions when updating many tables simultaneously. The total number of tables that can be updated at once is still 10; however, these tables can now be any combination of <code>CREATING</code>, <code>UPDATING</code> or <code>DELETING</code> status. Additionally, there is no longer any minimum amount for increasing or reducing the <code>ReadCapacityUnits</code> or <code>WriteCapacityUnits</code> for a	November 2, 2012

Change	Description	Date Changed
	<p>table. For more information, see Service, account, and table quotas in Amazon DynamoDB.</p>	
Best practices documentation	<p>The Amazon DynamoDB Developer Guide identifies best practices for working with tables and items, along with recommendations for query and scan operations.</p>	September 28, 2012

Change	Description	Date Changed
Support for binary data type	<p>In addition to the Number and String types, DynamoDB now supports Binary data type.</p> <p>Prior to this release, to store binary data, you converted your binary data into string format and stored it in DynamoDB. In addition to the required conversion work on the client-side, the conversion often increased the size of the data item requiring more storage and potentially additional provisioned throughput capacity.</p> <p>With the binary type attribute you can now store any binary data, for example compressed data, encrypted data, and images. For more information see Data types. For working examples of handling binary type data using the AWS SDKs, see the following sections:</p> <ul style="list-style-type: none">• Example: Handling binary type attributes using the AWS SDK for Java document API• Example: Handling binary type attributes using the	August 21, 2012

Change	Description	Date Changed
	<p>AWS SDK for .NET low-level API</p> <p>For the added binary data type support in the AWS SDKs, you will need to download the latest SDKs and you might also need to update any existing applications. For information about downloading the AWS SDKs, see .NET code examples.</p>	
DynamoDB table items can be updated and copied using the DynamoDB console	DynamoDB users can now update and copy table items using the DynamoDB Console, in addition to being able to add and delete items. This new functionality simplifies making changes to individual items through the Console.	August 14, 2012
DynamoDB lowers minimum table throughput requirements	DynamoDB now supports lower minimum table throughput requirements, specifically 1 write capacity unit and 1 read capacity unit. For more information, see the Service, account, and table quotas in Amazon DynamoDB topic in the Amazon DynamoDB Developer Guide.	August 9, 2012

Change	Description	Date Changed
Signature Version 4 support	DynamoDB now supports Signature Version 4 for authenticating requests.	July 5, 2012
Table explorer support in DynamoDB Console	The DynamoDB Console now supports a table explorer that enables you to browse and query the data in your tables. You can also insert new items or delete existing items. The Creating tables and loading data for code examples in DynamoDB and Using the console sections have been updated for these features.	May 22, 2012
New endpoints	<p>DynamoDB availability expands with new endpoints in the US West (N. California) region, US West (Oregon) region, and the Asia Pacific (Singapore) region.</p> <p>For the current list of supported endpoints, go to Regions and Endpoints.</p>	April 24, 2012

Change	Description	Date Changed
BatchWriteItem API support	<p>DynamoDB now supports a batch write API that enables you to put and delete several items from one or more tables in a single API call. For more information about the DynamoDB batch write API, see BatchWriteItem.</p> <p>For information about working with items and using batch write feature using AWS SDKs, see Working with items and attributes and .NET code examples.</p>	April 19, 2012
Documented more error codes	For more information, see Error handling with DynamoDB .	April 5, 2012
New endpoint	DynamoDB expands to the Asia Pacific (Tokyo) region. For the current list of supported endpoints, see Regions and Endpoints .	February 29, 2012

Change	Description	Date Changed
ReturnedItemCount metric added	A new metric, <code>ReturnedItemCount</code> , provides the number of items returned in the response of a Query or Scan operation for DynamoDB is available for monitoring through CloudWatch. For more information, see Logging and monitoring in DynamoDB .	February 24, 2012
Added examples for incrementing values	DynamoDB supports incrementing and decrementing existing numeric values. Examples show adding to existing values in the "Updating an Item" sections at: Working with items: Java. Working with items: .NET.	January 25, 2012
Initial product release	DynamoDB is introduced as a new service in Beta release.	January 18, 2012

Legacy features of DynamoDB

The following topics are legacy features that DynamoDB still supports. No active development is made on these features.

Topics

- [Global tables version 2017.11.29 \(Legacy\)](#)

Global tables version 2017.11.29 (Legacy)

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use [Global Tables version 2019.11.21 \(Current\)](#) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see [Upgrading global tables](#).

Topics

- [Global tables: How it works](#)
- [Best practices and requirements for managing global tables](#)
- [Creating a global table](#)
- [Monitoring global tables](#)
- [Using IAM with global tables](#)

Global tables: How it works

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use [Global Tables version 2019.11.21](#)

[\(Current\)](#) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see [Upgrading global tables](#).

The following sections help you understand the concepts and behavior of global tables in Amazon DynamoDB.

Global table concepts for Version 2017.11.29 (Legacy)

A *global table* is a collection of one or more replica tables, all owned by a single AWS account.

A *replica table* (or *replica*, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per AWS Region.

The following is a conceptual overview of how a global table is created.

1. Create an ordinary DynamoDB table, with DynamoDB Streams enabled, in an AWS Region.
2. Repeat step 1 for every other Region where you want to replicate your data.
3. Define a DynamoDB global table based on the tables that you have created.

The AWS Management Console automates these tasks, so you can create a global table more quickly and easily. For more information, see [Creating a global table](#).

The resulting DynamoDB global table consists of multiple replica tables, one per Region, that DynamoDB treats as a single unit. Every replica has the same table name and the same primary key schema. When an application writes data to a replica table in one Region, DynamoDB automatically propagates the write to the other replica tables in the other AWS Regions.

Important

To keep your table data in sync, global tables automatically create the following attributes for every item:

- `aws:rep:deleting`
- `aws:rep:updatetime`

- `aws:rep:updateregion`

Do not modify these attributes or create attributes with the same name.

You can add replica tables to the global table so that it can be available in additional Regions. (To do this, the global table must be empty. In other words, none of the replica tables can have any data in them.)

You can also remove a replica table from a global table. If you do this, the table is completely disassociated from the global table. This newly independent table no longer interacts with the global table, and data is no longer propagated to or from the global table.

Warning

Be aware that removing a replica is not an atomic process. To ensure consistent behavior and known state, you may want to consider diverting your application write traffic away from the replica to be removed ahead of time. After removing it, wait until all replica region endpoints show the replica as disassociated before making any further writes to it as its own isolated regional table.

Common tasks

Common tasks for global tables work as follows.

You can delete a global table's replica table the same as a regular table. This will stop replication to that Region and delete the table copy kept in that Region. You cannot sever the replication and have copies of the table exist as independent entities.

Note

You won't be able to delete a source table until at least 24 hours after it's used to initiate a new Region. If you try to delete it too soon you will receive an error.

Conflicts can arise if applications update the same item in different Regions at about the same time. To help ensure eventual consistency, DynamoDB global tables use a "last writer wins" method

to reconcile between concurrent updates. All the replicas will agree on the latest update and converge toward a state in which they all have identical data.

Note

There are several ways to avoid conflicts, including:

- Using an IAM policy to only allow writes to the table in one region.
- Using an IAM policy to route users to only one region and keeping the other as an idle standby, or alternately routing odd users to one region and even users to another region.
- Avoiding the use of non-idempotent updates such as `Bookmark = Bookmark + 1`, in favor of static updates such as `Bookmark=25`.

Monitoring global tables

You can use CloudWatch to observe the metric `ReplicationLatency`. This metric tracks the elapsed time between when an updated item appears in the DynamoDB stream for one replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source-Region and destination-Region pair. This is the only CloudWatch metric provided by Global Tables v2.

The latencies you will observe depend on the distance between your chosen Regions, as well as other variables. Latencies in the 0.5 to 2.5 second range for Regions can be common within the same geographic area.

Time To Live (TTL)

You can use Time To Live (TTL) to specify an attribute name whose value indicates the time of expiration for the item. This value is specified as a number in seconds since the start of the Unix epoch.

With global tables legacy version, the TTL deletes are not automatically replicated across other replicas. When an item is deleted via a TTL rule, that work is performed without consuming Write Units.

Be aware that if the source and target table have very low Provisioned write capacity, this may cause throttling as the TTL deletes require write capacity.

Streams and transactions with global tables

Each global table produces an independent stream based on all its writes, regardless of the origination point for those writes. You can choose to consume this DynamoDB stream in one Region or in all Regions independently.

If you want processed local writes but not replicated writes, you can add your own region attribute to each item. Then you can use a Lambda event filter to invoke only the Lambda for writes in the local Region.

Transactional operations provide ACID (Atomicity, Consistency, Isolation, and Durability) guarantees ONLY within the Region where the write is made originally. Transactions are not supported across Regions in global tables.

For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a `TransactWriteItems` operation in the US East (Ohio) Region, you may observe partially completed transactions in US West (Oregon) Region as changes are replicated. Changes will only be replicated to other Regions once they have been committed in the source Region.

Note

- Global tables “write around” DynamoDB Accelerator by updating DynamoDB directly. As a result DAX will not be aware it is holding stale data. The DAX cache will only be refreshed when the cache’s TTL expires.
- Tags on global tables do not automatically propagate.

Read and write throughput

Global tables manage read and write throughput in the following ways.

- The write capacity must be the same on all table instances across Regions.
- With Version 2019.11.21 (Current, if the table is set to support autoscaling or is in on-demand mode then the write capacity is automatically kept in sync. The current amount of write capacity provisioned in each Region will rise and fall independently within those synchronized autoscaling settings. If the table is placed in on-demand mode, that mode will sync to the other replicas.

- Read capacity can differ between Regions because reads may not be equal. When adding a global replica to a table, the capacity of the source Region is propagated. After creation you can adjust the read capacity for one replica, and this new setting is not transferred to the other side.

Consistency and conflict resolution

Any changes made to any item in any replica table are replicated to all the other replicas within the same global table. In a global table, a newly written item is usually propagated to all replica tables within seconds.

With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

An application can read and write data to any replica table. DynamoDB supports eventually consistent reads across Regions, but does not support strongly consistent reads across Regions. If your application only uses eventually consistent reads and only issues reads against one AWS Region, it will work without any modification. However, if your application requires strongly consistent reads then it must perform all the strongly consistent reads and writes in the same Region. Otherwise, if you write to one Region and read from another Region then the read response might include stale data that doesn't reflect the results of recently completed writes in the other Region.

Conflicts can arise if applications update the same item in different Regions at about the same time. To help ensure eventual consistency, DynamoDB global tables use a *last writer wins* reconciliation between concurrent updates, in which DynamoDB makes a best effort to determine the last writer. With this conflict resolution mechanism, all the replicas will agree on the latest update and converge toward a state in which they all have identical data.

Availability and durability

If a single AWS Region becomes isolated or degraded, your application can redirect to a different Region and perform reads and writes against a different replica table. You can apply custom business logic to determine when to redirect requests to other Regions.

If a Region becomes isolated or degraded, DynamoDB keeps track of any writes that have been performed but have not yet been propagated to all of the replica tables. When the Region comes back online, DynamoDB resumes propagating any pending writes from that Region to the replica tables in other Regions. It also resumes propagating writes from other replica tables to the Region

that is now back online. All previously successful writes will be propagated eventually no matter how long the Region is isolated.

Best practices and requirements for managing global tables

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use [Global Tables version 2019.11.21 \(Current\)](#) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see [Upgrading global tables](#).

Using Amazon DynamoDB global tables, you can replicate your table data across AWS Regions. It is important that the replica tables and secondary indexes in your global table have identical write capacity settings to ensure proper replication of data.

Topics

- [Global tables version](#)
- [Requirements for adding a new replica table](#)
- [Best practices and requirements for managing capacity](#)

Global tables version

There are two versions of DynamoDB global tables available: [Global Tables version 2019.11.21 \(Current\)](#) and [Global tables version 2017.11.29 \(Legacy\)](#). Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from Version 2017.11.29 (Legacy) to Version 2019.11.21 (Current), see [Upgrading global tables](#).

Requirements for adding a new replica table

If you want to add a new replica table to a global table, each of the following conditions must be true:

- The table must have the same partition key as all of the other replicas.
- The table must have the same write capacity management settings specified.
- The table must have the same name as all of the other replicas.
- The table must have DynamoDB Streams enabled, with the stream containing both the new and the old images of the item.
- None of the new or existing replica tables in the global table can contain any data.

If global secondary indexes are specified, the following conditions must also be met:

- The global secondary indexes must have the same name.
- The global secondary indexes must have the same partition key and sort key (if present).

Important

Write capacity settings should be set consistently across all of your global tables' replica tables and matching secondary indexes. To update write capacity settings for your global table, we strongly recommend using the DynamoDB console or the `UpdateGlobalTableSettings` API operation. `UpdateGlobalTableSettings` applies changes to write capacity settings to all replica tables and matching secondary indexes in a global table automatically. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` operations, you should apply the change to each replica table and matching secondary index individually. For more information, see [UpdateGlobalTableSettings](#) in the [Amazon DynamoDB API Reference](#).

We strongly recommend that you enable auto scaling to manage provisioned write capacity settings. If you prefer to manage write capacity settings manually, you should provision equal replicated write capacity units to all of your replica tables. Also provision equal replicated write capacity units to matching secondary indexes across your global table. You must also have appropriate AWS Identity and Access Management (IAM) permissions. For more information, see [Using IAM with global tables](#).

Best practices and requirements for managing capacity

Consider the following when managing capacity settings for replica tables in DynamoDB.

Using DynamoDB auto scaling

Using DynamoDB auto scaling is the recommended way to manage throughput capacity settings for replica tables that use the provisioned mode. DynamoDB auto scaling automatically adjusts read capacity units (RCUs) and write capacity units (WCUs) for each replica table based upon your actual application workload. For more information, see [Managing throughput capacity automatically with DynamoDB auto scaling](#).

If you create your replica tables using the AWS Management Console, auto scaling is enabled by default for each replica table, with default auto scaling settings for managing read capacity units and write capacity units.

Changes to auto scaling settings for a replica table or secondary index made through the DynamoDB console or using the `UpdateGlobalTableSettings` call are applied to all of the replica tables and matching secondary indexes in the global table automatically. These changes overwrite any existing auto scaling settings. This ensures that provisioned write capacity settings are consistent across the replica tables and secondary indexes in your global table. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` calls, you should apply the change to each replica table and matching secondary index individually.

Note

If auto scaling doesn't satisfy your application's capacity changes (unpredictable workload), or if you don't want to configure its settings (target settings for minimum, maximum, or utilization threshold), you can use on-demand mode to manage capacity for your global tables. For more information, see [On-demand mode](#).

If you enable on-demand mode on a global table, your consumption of replicated write request units (rWCUs) will be consistent with how rWCUs are provisioned. For example, if you perform 10 writes to a local table that is replicated in two additional Regions, you will consume 60 write request units ($10 + 10 + 10 = 30$; $30 \times 2 = 60$). The consumed 60 write request units include the extra write consumed by global tables Version 2017.11.29 (Legacy) to update the `aws:rep:deleting`, `aws:rep:updatetime`, and `aws:rep:updateregion` attributes.

Managing capacity manually

If you decide not to use DynamoDB auto scaling, you must manually set the read capacity and write capacity settings on each replica table and secondary index.

The provisioned replicated write capacity units (rWCUs) on every replica table should be set to the total number of rWCUs needed for application writes across all Regions multiplied by two. This accommodates application writes that occur in the local Region and replicated application writes coming from other Regions. For example, suppose that you expect 5 writes per second to your replica table in Ohio and 5 writes per second to your replica table in N. Virginia. In this case, you should provision 20 rWCUs to each replica table ($5 + 5 = 10$; $10 \times 2 = 20$).

To update write capacity settings for your global table, we strongly recommend using the DynamoDB console or the `UpdateGlobalTableSettings` API operation. `UpdateGlobalTableSettings` applies changes to write capacity settings to all replica tables and matching secondary indexes in a global table automatically. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` operations, you should apply the change to each replica table and matching secondary index individually. For more information, see [Amazon DynamoDB API Reference](#).

Note

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions. For more information, see [Using IAM with global tables](#).

Creating a global table

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use [Global Tables version 2019.11.21 \(Current\)](#) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see [Upgrading global tables](#).

This section describes how to create a global table using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Topics

- [Creating a global table \(console\)](#)
- [Creating a global table \(AWS CLI\)](#)

Creating a global table (console)

Follow these steps to create a global table using the console. The following example creates a global table with replica tables in United States and Europe.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>. For this example, choose the us-east-2 (US East Ohio) Region.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**.

For **Table name**, enter **Music**.

For **Primary key** enter **Artist**. Choose **Add sort key**, and enter **SongTitle**. (**Artist** and **SongTitle** should both be strings.)

To create the table, choose **Create**. This table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.

4. Choose the **Global Tables** tab, and then choose **Create a Version 2017.11.29 (Legacy) replica**.

The screenshot shows the 'Global tables' tab selected in the navigation bar. Under the 'Replicas' section, it says '(0)' and 'Other AWS Regions to which you have replicated this table.' There are three buttons: a 'C' icon, 'Delete replica', and 'Create replica'. Below this, it says 'No replicas' and has another 'Create replica' button. A note on the left says: 'You are using global tables version 2019.11.21. If you need to use version 2017.11.29 instead, choose "Create version 2017.11.29 replica." You can create a version 2017.11.29 replica only if you have an empty table.' To the right of this note is a button 'Create a version 2017.11.29 replica.', which is circled in red.

5. From the **Available replication Regions** dropdown, choose **US West (Oregon)**.

The console checks to ensure that a table with the same name doesn't exist in the selected Region. If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.

6. Choose **Create Replica**. This starts the table creation process in US West (Oregon);

The **Global Table** tab for the selected table (and for any other replica tables) shows that the table has been replicated in multiple Regions.

7. Now add another Region so that your global table is replicated and synchronized across the United States and Europe. To do this, repeat step 5, but this time, specify **Europe (Frankfurt)** instead of **US West (Oregon)**.
8. You should still be using the AWS Management Console in the US East (Ohio) Region. Select **Items** in the left navigation menu, select the **Music** table, then choose **Create Item**.
 - a. For **Artist**, enter **item_1**.
 - b. For **SongTitle**, enter **Song Value 1**.
 - c. To write the item, choose **Create item**.

9. After a short time, the item is replicated across all three Regions of your global table. To verify this, in the console, on the Region selector in the upper-right corner, choose **Europe (Frankfurt)**. The Music table in Europe (Frankfurt) should contain the new item.
10. Repeat step 9 and choose **US West (Oregon)** to verify replication in that region.

Creating a global table (AWS CLI)

Follow these steps to create a global table Music using the AWS CLI. The following example creates a global table, with replica tables in the United States and in Europe.

1. Create a new table (Music) in US East (Ohio), with DynamoDB Streams enabled (NEW_AND_OLD_IMAGES).

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
   AttributeName=Artist,AttributeType=S \
   AttributeName=SongTitle,AttributeType=S \
  --key-schema \
   AttributeName=Artist,KeyType=HASH \
   AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
  --region us-east-2
```

2. Create an identical Music table in US East (N. Virginia).

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
   AttributeName=Artist,AttributeType=S \
   AttributeName=SongTitle,AttributeType=S \
  --key-schema \
   AttributeName=Artist,KeyType=HASH \
   AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
  --region us-east-1
```

3. Create a global table (`Music`) consisting of replica tables in the `us-east-2` and `us-east-1` Regions.

```
aws dynamodb create-global-table \
--global-table-name Music \
--replication-group RegionName=us-east-2 RegionName=us-east-1 \
--region us-east-2
```

 **Note**

The global table name (`Music`) must match the name of each of the replica tables (`Music`). For more information, see [Best practices and requirements for managing global tables](#).

4. Create another table in Europe (Ireland), with the same settings as those you created in step 1 and step 2.

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
  AttributeName=Artist,AttributeType=S \
  AttributeName=SongTitle,AttributeType=S \
--key-schema \
  AttributeName=Artist,KeyType=HASH \
  AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
  ReadCapacityUnits=10,WriteCapacityUnits=5 \
--stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
--region eu-west-1
```

After doing this step, add the new table to the `Music` global table.

```
aws dynamodb update-global-table \
--global-table-name Music \
--replica-updates 'Create={RegionName=eu-west-1}' \
--region us-east-2
```

5. To verify that replication is working, add a new item to the `Music` table in US East (Ohio).

```
aws dynamodb put-item \
```

```
--table-name Music \
--item '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region us-east-2
```

6. Wait for a few seconds, and then check to see whether the item has been successfully replicated to US East (N. Virginia) and Europe (Ireland).

```
aws dynamodb get-item \
--table-name Music \
--key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region us-east-1
```

```
aws dynamodb get-item \
--table-name Music \
--key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region eu-west-1
```

Monitoring global tables

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use [Global Tables version 2019.11.21 \(Current\)](#) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see [Upgrading global tables](#).

You can use Amazon CloudWatch to monitor the behavior and performance of a global table. Amazon DynamoDB publishes `ReplicationLatency` and `PendingReplicationCount` metrics for each replica in the global table.

- **ReplicationLatency**—The elapsed time between when an updated item appears in the DynamoDB stream for one replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source- and destination-Region pair.

During normal operation, `ReplicationLatency` should be fairly constant. An elevated value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. Over time, this could result in other replica tables *falling behind* because they no longer receive updates consistently. In this case, you should verify that the read capacity units (RCUs) and write capacity units (WCUs) are identical for each of the replica tables. In addition, when choosing WCU settings, follow the recommendations in [Global tables version](#).

`ReplicationLatency` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

- **PendingReplicationCount**—The number of item updates that are written to one replica table, but that have not yet been written to another replica in the global table. `PendingReplicationCount` is expressed in number of items and is emitted for every source-and destination-Region pair.

During normal operation, `PendingReplicationCount` should be very low. If `PendingReplicationCount` increases for extended periods, investigate whether your replica tables' provisioned write capacity settings are sufficient for your current workload.

`PendingReplicationCount` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

For more information, see [DynamoDB Metrics and dimensions](#).

Using IAM with global tables

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use [Global Tables version 2019.11.21 \(Current\)](#) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see [Determining the global table version you are using](#). To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see [Upgrading global tables](#).

When you create a global table for the first time, Amazon DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role is named [AWSServiceRoleForDynamoDBReplication](#), and it allows DynamoDB to manage cross-Region replication for global tables on your behalf. Don't delete this service-linked role. If you do, then all of your global tables will no longer function.

For more information about service-linked roles, see [Using service-linked roles in the IAM User Guide](#).

To create and maintain global tables in DynamoDB, you must have the `dynamodb:CreateGlobalTable` permission to access each of the following:

- The replica table that you want to add.
- Each existing replica that's already part of the global table.
- The global table itself.

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions.

The `application-autoscaling:DeleteScalingPolicy` and `application-autoscaling:DeregisterScalableTarget` permissions are required when updating an existing scaling policy. This is so that the global tables service can remove the old scaling policy before attaching the new policy to the table or secondary index.

If you use an IAM policy to manage access to one replica table, you should apply an identical policy to all other replicas within that global table. This practice helps you maintain a consistent permissions model across all of the replica tables.

By using identical IAM policies on all replicas in a global table, you can also avoid granting unintended read and write access to your global table data. For example, consider a user who

has access to only one replica in a global table. If that user can write to this replica, DynamoDB propagates the write to all of the other replica tables. In effect, the user can (indirectly) write to all of the other replicas in the global table. This scenario can be avoided by using consistent IAM policies on all of the replica tables.

Example: Allow the CreateGlobalTable action

Before you can add a replica to a global table, you must have the `dynamodb:CreateGlobalTable` permission for the global table and for each of its replica tables.

The following IAM policy grants permissions to allow the `CreateGlobalTable` action on all tables.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb:CreateGlobalTable"],  
            "Resource": "*"  
        }  
    ]  
}
```

Example: Allow the UpdateGlobalTable, DescribeLimits, application-autoscaling>DeleteScalingPolicy, and application-autoscaling:DeregisterScalableTarget actions

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling>DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions.

The following IAM policy grants permissions to allow the `UpdateGlobalTableSettings` action on all tables.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb:UpdateGlobalTableSettings"],  
            "Resource": "*"  
        }  
    ]  
}
```

```
        "Effect": "Allow",
        "Action": [
            "dynamodb:UpdateGlobalTable",
            "dynamodb:DescribeLimits",
            "application-autoscaling:DeleteScalingPolicy",
            "application-autoscaling:DeregisterScalableTarget"
        ],
        "Resource": "*"
    }
]
```

Example: Allow the CreateGlobalTable action for a specific global table name with replicas allowed in certain regions only

The following IAM policy grants permissions to allow the CreateGlobalTable action to create a global table named `Customers` with replicas in two Regions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb>CreateGlobalTable",
            "Resource": [
                "arn:aws:dynamodb::123456789012:global-table/Customers",
                "arn:aws:dynamodb:us-east-1:123456789012:table/Customers",
                "arn:aws:dynamodb:us-west-1:123456789012:table/Customers"
            ]
        }
    ]
}
```