
Amazon DynamoDB

Developer Guide

API Version 2012-08-10



Amazon DynamoDB: Developer Guide

Copyright © 2017 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
How It Works	2
Core Components	2
The DynamoDB API	9
Naming Rules and Data Types	11
Read Consistency	15
Throughput Capacity	15
Partitions and Data Distribution	18
From SQL to NoSQL	20
SQL or NoSQL?	21
Accessing the Database	22
Creating a Table	24
Getting Information About a Table	26
Writing Data To a Table	27
Reading Data From a Table	29
Managing Indexes	34
Modifying Data in a Table	37
Deleting Data from a Table	38
Removing a Table	39
Setting Up DynamoDB	41
Setting Up DynamoDB Local (Downloadable Version)	41
Downloading and Running DynamoDB on Your Computer	41
Setting the Local Endpoint	44
Usage Notes	44
Setting Up DynamoDB (Web Service)	46
Signing Up for AWS	46
Getting an AWS Access Key	46
Configuring Your Credentials	47
Accessing DynamoDB	48
Using the Console	48
Using the CLI	49
Downloading and Configuring the AWS CLI	49
Using the AWS CLI with DynamoDB	49
Using the AWS CLI with Downloadable DynamoDB	50
Using the API	51
Getting Started with DynamoDB	52
Java and DynamoDB	52
Prerequisites	53
Step 1: Create a Table	53
Step 2: Load Sample Data	54
Step 3: Create, Read, Update, and Delete an Item	57
Step 4: Query and Scan the Data	64
Step 5: (Optional) Delete the Table	68
Summary	68
JavaScript and DynamoDB	69
Prerequisites	69
Step 1: Create a Table	70
Step 2: Load Sample Data	72
Step 3: Create, Read, Update, and Delete an Item	74
Step 4: Query and Scan the Data	83
Step 5: (Optional): Delete the Table	87
Summary	88
Node.js and DynamoDB	89
Prerequisites	89

Step 1: Create a Table	89
Step 2: Load Sample Data	90
Step 3: Create, Read, Update, and Delete an Item	92
Step 4: Query and Scan the Data	99
Step 5: (Optional): Delete the Table	102
Summary	103
.NET and DynamoDB	103
Prerequisites	104
Step 1: Create a Table	104
Step 2: Load Sample Data	106
Step 3: Create, Read, Update, and Delete an Item	109
Step 4: Query and Scan the Data	122
Step 5: (Optional) Delete the Table	130
Summary	131
PHP and DynamoDB	132
Prerequisites	132
Step 1: Create a Table	132
Step 2: Load Sample Data	134
Step 3: Create, Read, Update, and Delete an Item	136
Step 4: Query and Scan the Data	144
Step 5: (Optional) Delete the Table	148
Summary	149
Python and DynamoDB	149
Prerequisites	150
Step 1: Create a Table	150
Step 2: Load Sample Data	151
Step 3: Create, Read, Update, and Delete an Item	153
Step 4: Query and Scan the Data	160
Step 5: (Optional) Delete the Table	163
Summary	164
Ruby and DynamoDB	164
Prerequisites	164
Step 1: Create a Table	165
Step 2: Load Sample Data	166
Step 3: Create, Read, Update, and Delete an Item	168
Step 4: Query and Scan the Data	174
Step 5: (Optional) Delete the Table	178
Summary	179
Programming with DynamoDB	180
Overview of AWS SDK Support for DynamoDB	180
Programmatic Interfaces	182
Low-Level Interfaces	182
Document Interfaces	183
Object Persistence Interface	184
DynamoDB Low-Level API	185
Request Format	187
Response Format	187
Data Type Descriptors	188
Numeric Data	188
Binary Data	189
Error Handling	189
Error Components	189
Error Messages and Codes	190
Error Handling in Your Application	192
Error Retries and Exponential Backoff	193
Batch Operations and Error Handling	194
Higher-Level Programming Interfaces for DynamoDB	194

Java: DynamoDBMapper	195
.NET: Document Model	232
.NET: Object Persistence Model	253
Running the Code Samples	280
Load Sample Data	280
Java Code Samples	285
.NET Code Samples	287
Working with DynamoDB	290
Working with Tables	290
Basic Operations for Tables	291
Throughput Settings for Reads and Writes	294
Item Sizes and Capacity Unit Consumption	296
Managing Throughput Capacity with Auto Scaling	298
Tagging for DynamoDB	314
Working with Tables: Java	316
Working with Tables: .NET	321
Working with Items	327
Reading an Item	328
Writing an Item	329
Return Values	331
Batch Operations	331
Atomic Counters	333
Conditional Writes	334
Using Expressions in DynamoDB	338
Time To Live	362
Working with Items: Java	368
Working with Items: .NET	387
Working with Queries	410
Key Condition Expression	410
Filter Expressions for <i>Query</i>	412
Limiting the Number of Items in the Result Set	413
Paginating the Results	413
Counting the Items in the Results	414
Capacity Units Consumed by <i>Query</i>	414
Read Consistency for <i>Query</i>	415
Querying: Java	415
Querying: .NET	420
Working with Scans	427
Filter Expressions for <i>Scan</i>	427
Limiting the Number of Items in the Result Set	428
Paginating the Results	428
Counting the Items in the Results	429
Capacity Units Consumed by <i>Scan</i>	429
Read Consistency for <i>Scan</i>	430
Parallel Scan	430
Scanning: Java	431
Scanning: .NET	438
Working with Indexes	446
Global Secondary Indexes	448
Local Secondary Indexes	484
Working with Streams	518
Endpoints for DynamoDB Streams	519
Enabling a Stream	520
Reading and Processing a Stream	521
DynamoDB Streams and Time To Live	523
Using the DynamoDB Streams Kinesis Adapter to Process Stream Records	523
Walkthrough: DynamoDB Streams Low-Level API	534

Cross-Region Replication	540
DynamoDB Streams and AWS Lambda Triggers	541
In-Memory Acceleration with DAX	549
Use Cases for DAX	549
Usage Notes	550
Concepts	550
How DAX Processes Requests	551
Item Cache	552
Query Cache	553
DAX Cluster Components	553
Nodes	553
Clusters	554
Regions and Availability Zones	555
Parameter Groups	555
Security Groups	555
Cluster ARN	555
Cluster Endpoint	555
Node Endpoints	556
Subnet Groups	556
Events	556
Maintenance Window	556
Creating a DAX Cluster	557
Creating a DAX Service Role	557
AWS CLI	558
AWS Management Console	562
DAX and DynamoDB Consistency Models	564
Consistency Among DAX Cluster Nodes	564
DAX Item Cache Behavior	565
DAX Query Cache Behavior	566
Strongly Consistent Reads	567
Negative Caching	567
Strategies for Writes	568
Using the DAX Client in an Application	570
Tutorial: Running a Sample Application	570
Modifying an Existing Application to Use DAX	587
Managing DAX Clusters	590
IAM Permissions for Managing a DAX Cluster	591
Customizing DAX Cluster Settings	592
Configuring TTL Settings	594
Tagging Support for DAX	595
AWS CloudTrail Integration	596
Deleting a DAX Cluster	596
DAX Access Control	596
IAM Service Role for DAX	597
IAM Policy to Allow DAX Cluster Access	597
Case Study: Accessing DynamoDB and DAX	598
Access to DynamoDB, But No Access With DAX	600
Access to DynamoDB and to DAX	602
Access to DynamoDB Via DAX, But No Direct Access to DynamoDB	606
DAX API Reference	608
Authentication and Access Control	609
Authentication	609
Access Control	610
Overview of Managing Access	610
Amazon DynamoDB Resources and Operations	611
Understanding Resource Ownership	611
Managing Access to Resources	612

Specifying Policy Elements: Actions, Effects, and Principals	613
Specifying Conditions in a Policy	613
Using Identity-Based Policies (IAM Policies)	614
Console Permissions	614
AWS Managed (Predefined) Policies for Amazon DynamoDB	615
Customer Managed Policy Examples	615
DynamoDB API Permissions Reference	621
Related Topics	625
Using Conditions	625
Overview	626
Specifying Conditions: Using Condition Keys	628
Related Topics	635
Using Web Identity Federation	635
Additional Resources for Web Identity Federation	635
Example Policy for Web Identity Federation	636
Preparing to Use Web Identity Federation	638
Writing Your App to Use Web Identity Federation	639
Monitoring DynamoDB	642
Monitoring Tools	643
Automated Tools	643
Manual Tools	643
Monitoring with Amazon CloudWatch	644
Metrics and Dimensions	644
Using Metrics	655
Creating Alarms	656
Logging DynamoDB Operations by Using AWS CloudTrail	658
DynamoDB Information in CloudTrail	658
Understanding DynamoDB Log File Entries	660
Best Practices for DynamoDB	665
Table Best Practices	665
Item Best Practices	665
Query and Scan Best Practices	666
Local Secondary Index Best Practices	666
Global Secondary Index Best Practices	666
Best Practices for Tables	666
Design For Uniform Data Access Across Items In Your Tables	667
Understand Partition Behavior	669
Use Burst Capacity Sparingly	673
Distribute Write Activity During Data Upload	673
Understand Access Patterns for Time Series Data	674
Cache Popular Items	674
Consider Workload Uniformity When Adjusting Provisioned Throughput	675
Test Your Application At Scale	676
Best Practices for Items	676
Use One-to-Many Tables Instead Of Large Set Attributes	677
Compress Large Attribute Values	677
Store Large Attribute Values in Amazon S3	677
Break Up Large Attributes Across Multiple Items	678
Best Practices for Querying and Scanning Data	679
Performance Considerations for Scans	679
Avoid Sudden Spikes in Read Activity	679
Take Advantage of Parallel Scans	681
Best Practices for Local Secondary Indexes	682
Use Indexes Sparingly	682
Choose Projections Carefully	682
Optimize Frequent Queries To Avoid Fetches	683
Take Advantage of Sparse Indexes	683

Watch For Expanding Item Collections	683
Best Practices for Global Secondary Indexes	684
Choose a Key That Will Provide Uniform Workloads	684
Take Advantage of Sparse Indexes	684
Use a Global Secondary Index For Quick Lookups	685
Create an Eventually Consistent Read Replica	685
Integrating with Other AWS Services	686
Amazon VPC Endpoints for DynamoDB	686
Tutorial: Using a VPC Endpoint for DynamoDB	688
Integrating with Amazon Cognito	693
Integrating with Amazon Redshift	694
Integrating with Amazon EMR	695
Overview	696
Tutorial: Working with Amazon DynamoDB and Apache Hive	696
Creating an External Table in Hive	703
Processing HiveQL Statements	705
Querying Data in DynamoDB	706
Copying Data to and from Amazon DynamoDB	707
Performance Tuning	718
Integrating with Amazon Data Pipeline	722
Prerequisites to Export and Import Data	724
Exporting Data From DynamoDB to Amazon S3	727
Importing Data From Amazon S3 to DynamoDB	728
Troubleshooting	729
Predefined Templates for AWS Data Pipeline and DynamoDB	730
Limits in DynamoDB	731
Capacity Units and Provisioned Throughput	731
Capacity Unit Sizes	731
Provisioned Throughput Default Limits	731
Increasing Provisioned Throughput	732
Decreasing Provisioned Throughput	732
Tables	733
Table Size	733
Tables Per Account	733
Secondary Indexes	733
Secondary Indexes Per Table	733
Projected Secondary Index Attributes Per Table	733
Partition Keys and Sort Keys	733
Partition Key Length	733
Partition Key Values	733
Sort Key Length	733
Sort Key Values	733
Naming Rules	734
Table Names and Secondary Index Names	734
Attribute Names	734
Data Types	734
String	734
Number	734
Binary	735
Items	735
Item Size	735
Item Size for Tables With Local Secondary Indexes	735
Attributes	735
Attribute Name-Value Pairs Per Item	735
Number of Values in List, Map, or Set	735
Attribute Values	735
Nested Attribute Depth	736

Expression Parameters	736
Lengths	736
Operators and Operands	736
Reserved Words	736
DynamoDB Streams	736
Simultaneous Readers of a Shard in DynamoDB Streams	736
Maximum Write Capacity for a Table With a Stream Enabled	736
DynamoDB Accelerator (DAX)	737
AWS Region Availability	737
Nodes	737
Parameter Groups	737
Subnet Groups	737
API-Specific Limits	737
Appendix	739
Example Tables and Data	739
Sample Data Files	740
Creating Example Tables and Uploading Data	749
Creating Example Tables and Uploading Data - Java	749
Creating Example Tables and Uploading Data - .NET	756
Example Application Using AWS SDK for Python (Boto)	764
Step 1: Deploy and Test Locally	765
Step 2: Examine the Data Model and Implementation Details	768
Step 3: Deploy in Production	774
Step 4: Clean Up Resources	780
Amazon DynamoDB Storage Backend for Titan	780
Reserved Words in DynamoDB	780
Legacy Conditional Parameters	789
AttributesToGet	790
AttributeUpdates	791
ConditionalOperator	793
Expected	793
KeyConditions	796
QueryFilter	798
ScanFilter	799
Writing Conditions With Legacy Parameters	800
Current Low-Level API Version (2012-08-10)	806
Previous Low-Level API Version (2011-12-05)	807
Batch.GetItem	807
Batch.PutItem	812
Create.Table	817
Delete.Item	822
Delete.Table	826
Describe.Tables	829
Get.Item	832
List.Tables	835
Put.Item	837
Query	842
Scan	851
Update.Item	861
Update.Table	867
Document History	871

What Is Amazon DynamoDB?

Welcome to the Amazon DynamoDB Developer Guide.

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.

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, and use the AWS Management Console to monitor resource utilization and performance metrics.

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 \(p. 362\)](#).

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 automatically replicated across multiple Availability Zones in an AWS region, providing built-in high availability and data durability.

We recommend that you begin by reading the following sections:

- [Amazon DynamoDB: How It Works \(p. 2\)](#)—To learn essential DynamoDB concepts.
- [Setting Up DynamoDB \(p. 41\)](#)—To learn how to setup DynamoDB (Downloadable Version or Web Service).
- [Accessing DynamoDB \(p. 48\)](#)—To learn how to access DynamoDB using the console, CLI, or API.

To get started quickly with DynamoDB, see [Getting Started with DynamoDB \(p. 52\)](#).

To learn more about application development see the following:

- [Programming with DynamoDB and the AWS SDKs \(p. 180\)](#)
- [Working with DynamoDB \(p. 290\)](#)

To quickly find recommendations for maximizing performance and minimizing throughput costs see [Best Practices for DynamoDB \(p. 665\)](#). To learn how to tag DynamoDB resources see [Tagging for DynamoDB \(p. 314\)](#).

For best practices, how-to guides and tools, be sure to check the DynamoDB Developer Resources page: <http://aws.amazon.com/dynamodb/developer-resources/>.

You can use AWS Database Migration Service to migrate data from a Relational Database or MongoDB to an Amazon DynamoDB table. For more information, see [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](#).

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 Sample Data \(p. 280\)](#) 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 \(p. 52\)](#).

Topics

- [DynamoDB Core Components \(p. 2\)](#)
- [The DynamoDB API \(p. 9\)](#)
- [Naming Rules and Data Types \(p. 11\)](#)
- [Read Consistency \(p. 15\)](#)
- [Throughput Capacity for Reads and Writes \(p. 15\)](#)
- [Partitions and Data Distribution \(p. 18\)](#)

DynamoDB Core Components

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 [Limits in DynamoDB \(p. 731\)](#).

Topics

- [Tables, Items, and Attributes \(p. 2\)](#)
- [Primary Key \(p. 5\)](#)
- [Secondary Indexes \(p. 5\)](#)
- [DynamoDB Streams \(p. 8\)](#)

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 multiple 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.



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

<pre>{ "Artist": "No One You Know", "SongTitle": "My Dog Spot", "AlbumTitle": "Hey Now", "Price": 1.98, "Genre": "Country", "CriticRating": 8.4 }</pre>
<pre>{ "Artist": "No One You Know", "SongTitle": "Somewhere Down The Road", "AlbumTitle": "Somewhat Famous", "Genre": "Country", "CriticRating": 8.4, "Year": 1984 }</pre>
<pre>{ "Artist": "The Acme Band", "SongTitle": "Still in Love", "AlbumTitle": "The Buck Starts Here", "Price": 2.47, "Genre": "Rock", "PromotionInfo": { "RadioStationsPlaying": ["KRCR", "KQBX", "WTNR", "WJZH"], "TourDates": { "Seattle": "20150625", "Cleveland": "20150630" }, "Rotation": "Heavy" } }</pre>
<pre>{ "Artist": "The Acme Band", "SongTitle": "Look Out, World", "AlbumTitle": "The Buck Starts Here", "Price": 0.99, "Genre": "Rock" }</pre>

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 in DynamoDB \(p. 290\)](#).

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 \(p. 2\)](#) is an example of a table with a simple primary key (*PersonID*). You can access any item in the *People* table immediately 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 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 two items to have the same partition key value. However, those two items must have different sort key values.

The *Music* table described in [Tables, Items, and Attributes \(p. 2\)](#) is an example of a table with a composite primary key (*Artist* and *SongTitle*). You can access any item in the *Music* table immediately, 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. You could even provide a value for *Artist* and a range of *SongTitle* values, to retrieve only a subset of songs by a particular artist.

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.

You can define up to 5 global secondary indexes and 5 local secondary indexes per table.

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.



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 \(p. 446\)](#).

DynamoDB Streams

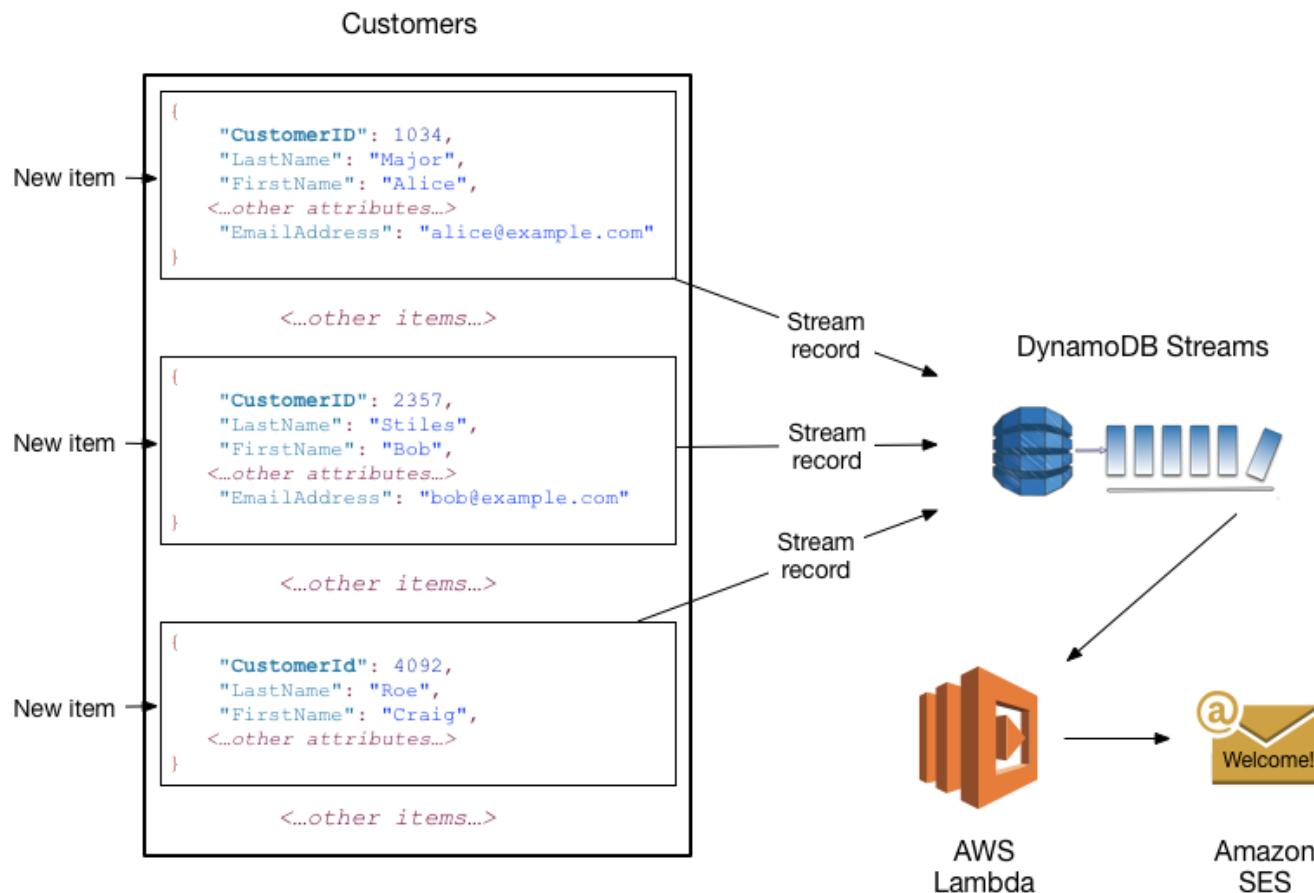
DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near real time, and in the order that the events occurred.

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:

- If a new item is added to the table, the stream captures an image of the entire item, including all of its attributes.
- If an item is updated, the stream captures the "before" and "after" image of any attributes that were modified in the item.
- If 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 executes 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 execute 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 [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#).

The 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.

Topics

- [Control Plane \(p. 9\)](#)
- [Data Plane \(p. 10\)](#)
- [DynamoDB Streams \(p. 11\)](#)

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, index information, and so on.
- `ListTables` – Returns the names of all of your tables in a list.
- `UpdateTable` – Modifies the settings of a table or its indexes, creates or remove 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.

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. You can also use `BatchWriteItem` for deleting multiple items from one or more tables.

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. You can also use `BatchWriteItem` for adding multiple items to one or more tables.

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.

Naming Rules and Data Types

This section describes the DynamoDB naming rules and the various data types that DynamoDB supports. There are limits that apply to datatypes. For more information, see [Data Types \(p. 734\)](#).

Topics

- [Naming Rules \(p. 11\)](#)
- [Data Types \(p. 12\)](#)

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 between 1 and 255 characters long.

Reserved Words and Special Characters

DynamoDB has a list of reserved words and special characters. For a complete list of reserved words in DynamoDB, see [Reserved Words in DynamoDB \(p. 780\)](#). 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 it because you have to define placeholder variables whenever you use these names in an expression. For more information, see [Expression Attribute Names \(p. 342\)](#).

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 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.

String

Strings are Unicode with UTF-8 binary encoding. The length of a string must be greater than zero and is constrained by the maximum DynamoDB item size limit of 400 KB.

If you define a primary key attribute as a string 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.

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 time stamp. 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.

Number

Numbers can be positive, negative, or zero. Numbers can have up to 38 digits precision. Exceeding this results in an exception.

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.

Note

If number precision is important, you should pass numbers to DynamoDB using strings that you convert from number type.

You can use the number data type to represent a date or a time stamp. 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 UTC on 17 July 2015.

For more information, see http://en.wikipedia.org/wiki/Unix_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 must be greater than zero, 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 cannot be an empty String or empty Set (String Set, Number Set, or Binary Set). However, empty Lists and Maps are allowed. For more information, see [Attributes \(p. 735\)](#).

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 \(p. 338\)](#).

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 \(p. 338\)](#).

Sets

DynamoDB supports types that represent sets of Number, String, or Binary values. All of the elements within a set must be of the same type. For example, an attribute of type Number Set can only contain numbers; String Set can only contain strings; and so on.

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. Finally, DynamoDB does not support empty sets.

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=" ]
```

Read Consistency

Amazon DynamoDB is available in multiple AWS regions around the world. Each region is independent and isolated from other AWS regions. For example, if you have a table called *People* in the *us-east-2* region and another table named *People* in the *us-west-2* region, these are considered two entirely separate tables. For a list of all the AWS regions in which DynamoDB is available, see [AWS Regions and Endpoints](#) in the *Amazon Web Services General Reference*.

Every AWS region consists of multiple distinct locations called Availability Zones. Each Availability Zone is isolated from failures in other Availability Zones, and to provide inexpensive, low-latency network connectivity to other Availability Zones in the same region. This allows rapid replication of your data among multiple Availability Zones in a region.

When your application writes data to a DynamoDB table and receives an HTTP 200 response (ok), all copies of the data are updated. The data is eventually consistent across all storage locations, usually within one second or less.

DynamoDB supports *eventually consistent* and *strongly consistent* reads.

Eventually Consistent Reads

When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. If you repeat your read request after a short time, the response should return the latest data.

Strongly Consistent Reads

When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. A strongly consistent read might not be available if there is a network delay or outage.

Note

DynamoDB uses eventually consistent reads, unless you specify otherwise. Read operations (such as `GetItem`, `Query`, and `Scan`) provide a `ConsistentRead` parameter. If you set this parameter to true, DynamoDB uses strongly consistent reads during the operation.

Throughput Capacity for Reads and Writes

Topics

- [DynamoDB Auto Scaling \(p. 16\)](#)
- [Provisioned Throughput \(p. 17\)](#)
- [Reserved Capacity \(p. 17\)](#)

When you create a table or index in Amazon DynamoDB, you must specify your capacity requirements for read and write activity. By defining your throughput capacity in advance, DynamoDB can reserve the necessary resources to meet the read and write activity your application requires, while ensuring consistent, low-latency performance.

You specify throughput capacity in terms of read capacity units and write capacity units:

- 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. If you need to read an item that is larger than 4 KB, DynamoDB will need to 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.
- 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 will need to consume additional write capacity units. The total number of write capacity units required depends on the item size.

For example, suppose that you create a table with 5 read capacity units and 5 write capacity units. With these settings, your application could:

- Perform strongly consistent reads of up to 20 KB per second ($4\text{ KB} \times 5$ read capacity units).
- Perform eventually consistent reads of up to 40 KB per second (twice as much read throughput).
- Write up to 5 KB per second ($1\text{ KB} \times 5$ write capacity units).

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

If your read or write requests exceed the throughput settings for a table, DynamoDB can *throttle* that request. DynamoDB can also throttle read requests exceeds for an index. 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 \(p. 193\)](#)), so you do not need to write this logic yourself.

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

DynamoDB provides the following mechanisms for managing throughput:

- [DynamoDB auto scaling](#)
- [Provisioned throughput](#)
- [Reserved capacity](#)

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 \(p. 298\)](#).

Provisioned Throughput

If you aren't using DynamoDB auto scaling, you have to manually define your throughput requirements. *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 settings, it is subject to request throttling.

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, 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:

- $1 \text{ read capacity unit per item} \times 80 \text{ reads per second} = \mathbf{80}$ 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, 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:

- $1 \text{ write capacity unit per item} \times 100 \text{ writes per second} = \mathbf{100}$ write capacity units

For more information see [Item Sizes and Capacity Unit Consumption \(p. 296\)](#).

Reserved Capacity

As a DynamoDB customer, you can purchase *reserved capacity* in advance, as described at [Amazon DynamoDB Pricing](#). With reserved capacity, you pay a one-time upfront fee and commit to a minimum usage level over a period of time. By reserving your read and write capacity units ahead of time, you realize significant cost savings compared to on-demand provisioned throughput settings.

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 [Authentication and Access Control for Amazon DynamoDB \(p. 609\)](#).

Partitions and Data Distribution

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.

For more details, see [Understand Partition Behavior \(p. 669\)](#).

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.

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

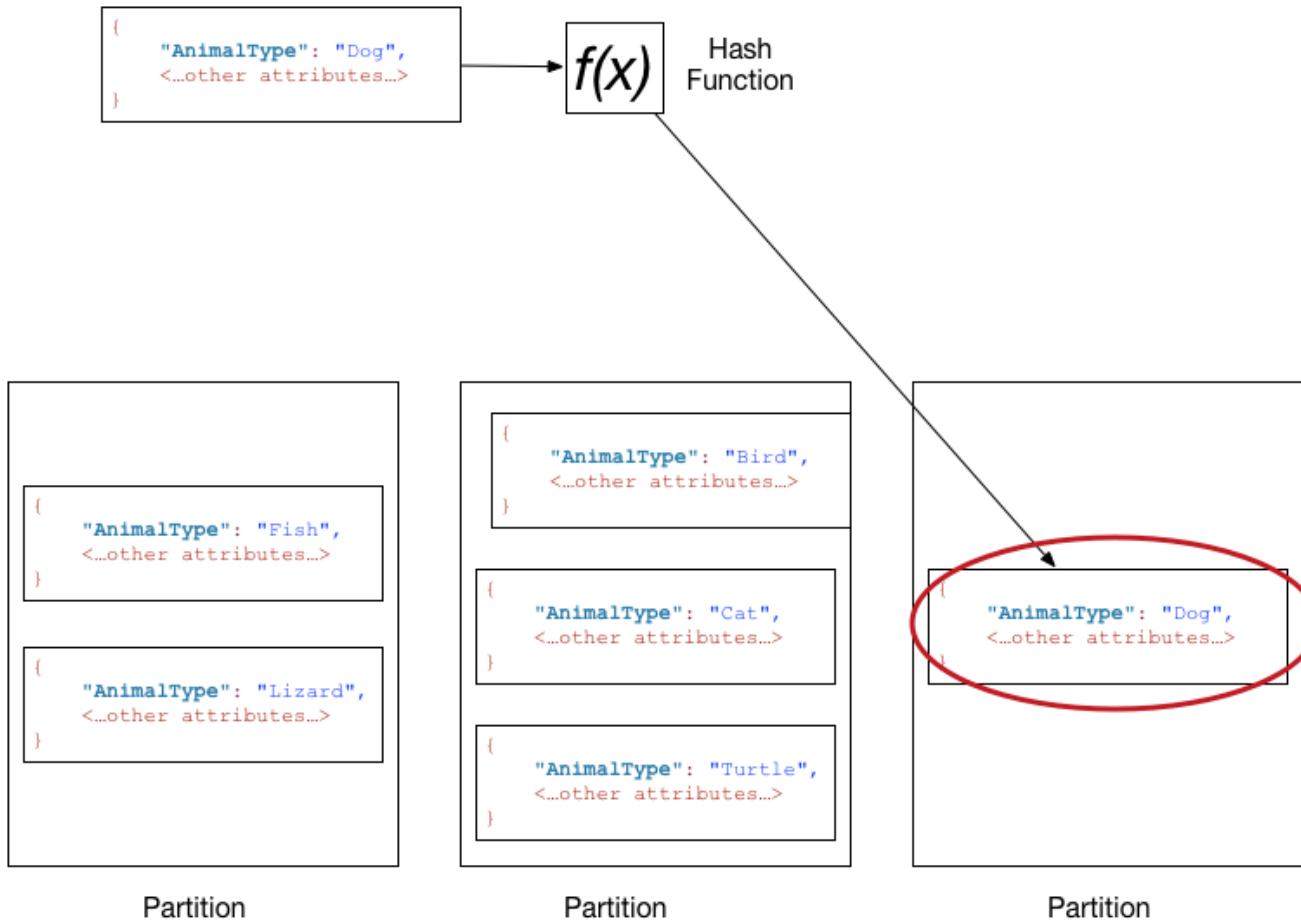
Data Distribution: Partition Key

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. For more information, see [Best Practices for Tables \(p. 666\)](#).

Data Distribution: Partition Key and Sort Key

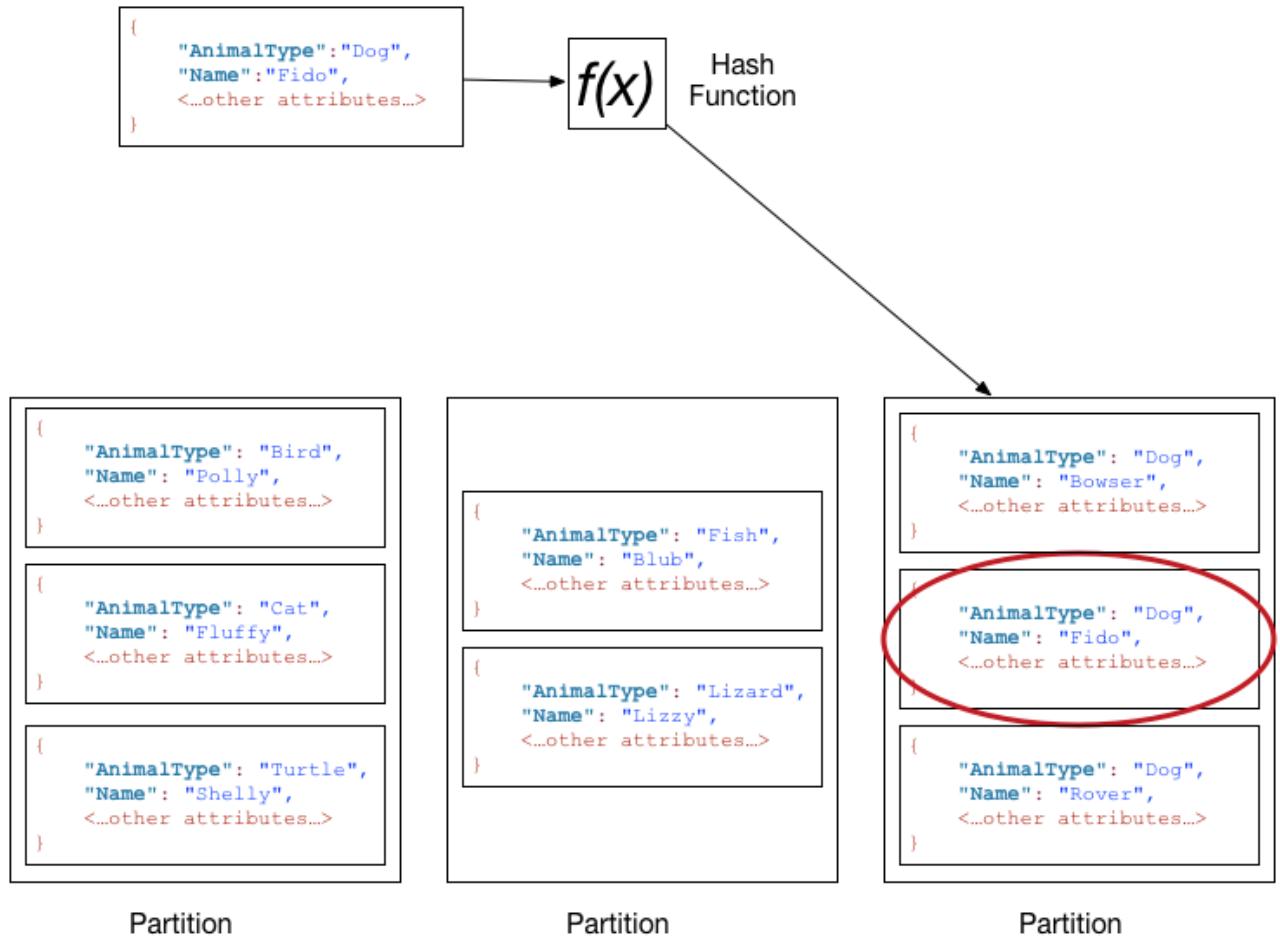
If the table has a composite primary key (partition key and sort key), DynamoDB calculates the hash value of the partition key in the same way as described in [Data Distribution: Partition Key \(p. 18\)](#)—but it stores all of the items with the same partition key value physically close together, ordered by sort key value.

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, there could be several items with 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`), provided that 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* 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 automatically allocates enough storage to handle this requirement.

From SQL to NoSQL

If you are an application developer, you might have some experience using relational database management systems (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. This section describes common database tasks, comparing and contrasting SQL statements with their equivalent DynamoDB operations.

NoSQL is a term used to describe non-relational 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 <http://aws.amazon.com/nosql>.

Note

The SQL examples in this section are compatible with the MySQL relational database management system.

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 samples that use these operations, see [Getting Started with DynamoDB \(p. 52\)](#).

Topics

- [SQL or NoSQL? \(p. 21\)](#)
- [Accessing the Database \(p. 22\)](#)
- [Creating a Table \(p. 24\)](#)
- [Getting Information About a Table \(p. 26\)](#)
- [Writing Data To a Table \(p. 27\)](#)
- [Reading Data From a Table \(p. 29\)](#)
- [Managing Indexes \(p. 34\)](#)
- [Modifying Data in a Table \(p. 37\)](#)
- [Deleting Data from a Table \(p. 38\)](#)
- [Removing a Table \(p. 39\)](#)

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 not uncommon 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 with a small amount of provisioned throughput and gradually increase it as your application becomes more popular. DynamoDB scales seamlessly to handle very large amounts of data and very large numbers of users.

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

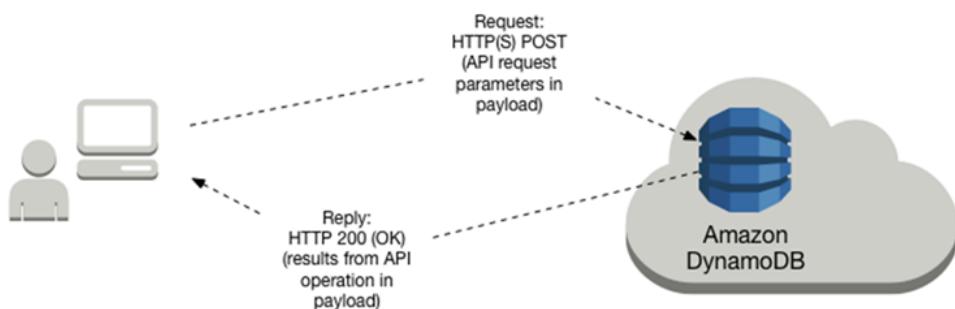
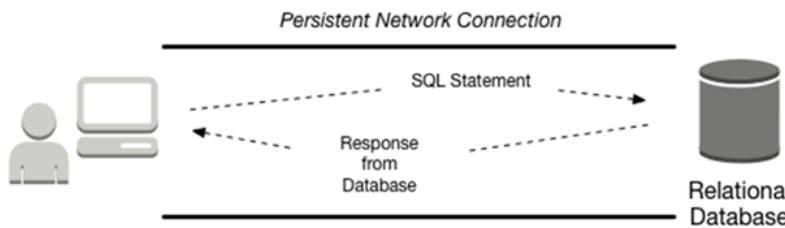
Characteristic	Relational Database Management System (RDBMS)	Amazon DynamoDB
Optimal Workloads	Ad hoc queries; data warehousing; OLAP (online analytical processing).	Web-scale applications, including social networks, gaming, media sharing, and IoT (Internet of Things).

Characteristic	Relational Database Management System (RDBMS)	Amazon DynamoDB
Data Model	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.	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 semi-structured data, including JSON documents.
Data Access	SQL (Structured Query Language) 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.	You can use the AWS Management Console or the AWS CLI to work with DynamoDB and perform ad hoc tasks. Applications can leverage the AWS software development kits (SDKs) to work with DynamoDB using object-based, document-centric, or low-level interfaces.
Performance	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.	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.
Scaling	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.	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.

Accessing the Database

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

The following diagram shows client interaction with a relational database, and with 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 interface (CLI), so that you can enter ad hoc SQL statements and see the results immediately.	In most cases, you write application code. You can also use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to send ad hoc requests to DynamoDB and view the results.
Connecting to the Database	An application program establishes and maintains a network connection with the database. When the application is finished, it terminates the connection.	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.
Authentication	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.	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> .

Characteristic	Relational Database Management System (RDBMS)	Amazon DynamoDB
Authorization	Applications can only perform actions for which they have been authorized. Database administrators or application owners can use the SQL <code>GRANT</code> and <code>REVOKE</code> 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.	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 IAM 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 Authentication and Access Control for Amazon DynamoDB (p. 609) .
Sending a Request	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 executes the plan.	The application sends HTTP(S) requests to DynamoDB. The requests contain the name of the DynamoDB operation to perform, along with parameters. DynamoDB executes the request immediately.
Receiving a Response	The RDBMS returns the results from the SQL statement. If there is an error, the RDBMS returns an error status and message.	DynamoDB returns an HTTP(S) response containing the results of the operation. If there is an error, DynamoDB returns an HTTP error status and message(s).

Creating a Table

Tables are the fundamental data structures in relational databases and in DynamoDB. A relational database management systems (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 attributes or data types at table creation time.

SQL

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.

DynamoDB

Use the `CreateTable` action to create a 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: {  
        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 \(p. 2\)](#) and [Primary Key \(p. 5\)](#).
- `AttributeDefinitions` – Data types for the key schema attributes.
- `ProvisionedThroughput` – 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` action 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 samples using `CreateTable`, see [Getting Started with DynamoDB \(p. 52\)](#).

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. DynamoDB tables are schemaless, so only the primary key attributes are shown.

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. Here 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*.

DynamoDB

DynamoDB has a `DescribeTable` action, which is similar. The only parameter is the table name, as shown following:

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

The reply from `DescribeTable` looks like this:

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

```
        "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*.

DynamoDB tables contain *items*. Items are composed of *attributes*.

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

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 semi-structured data, usually with native JSON or XML data types. However, the implementation details vary among vendors.

In SQL, you 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

In this example, we are using the *Tags* column to store semi-structured data about the songs in the *Music* table. We have defined the *Tags* column as type TEXT, which can store up to 65535 characters in MySQL.

DynamoDB

In Amazon DynamoDB, you use the `PutItem` action 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 provide native support for documents, using JSON. This makes DynamoDB ideal for storing semi-structured 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` action are permanent when DynamoDB replies with an HTTP 200 status code (OK).

Note

For code samples using `PutItem`, see [Getting Started with DynamoDB \(p. 52\)](#).

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",
        "LengthInSeconds": 214
    }
}
```

```
"AlbumTitle": "The Buck Starts Here",
"Price": 2.47,
"Genre": "Rock",
"PromotionInfo": {
    "RadioStationsPlaying": [
        "KHCR", "KBOX", "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`, Amazon DynamoDB supports a `BatchWriteItem` action for writing multiple items at the same time.

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.

DynamoDB provides the following operations for reading data:

- `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 `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 \(p. 18\)](#).)
- `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 execute efficiently, the database and its applications should be performance-tuned on an ongoing basis.

DynamoDB is a non-relational NoSQL database, and 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 \(p. 30\)](#)
- [Querying a Table \(p. 31\)](#)
- [Scanning a Table \(p. 33\)](#)

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.

SQL

In SQL, you 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 row(s) 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*.

DynamoDB

DynamoDB provides the `GetItem` action 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 \(p. 18\)](#).)

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` action is very efficient: It uses the primary key value(s) 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` actions, which are described in [Querying a Table \(p. 31\)](#) and [Scanning a Table \(p. 33\)](#).

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 non-relational database. As such, it does not 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 samples using `GetItem`, see [Getting Started with DynamoDB \(p. 52\)](#).

Querying a Table

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

SQL

The SQL `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*.

DynamoDB

The DynamoDB `Query` action lets you retrieve data in a similar fashion. The `Query` action provides quick, efficient access to the physical locations where the data is stored. (For more information, see [Partitions and Data Distribution \(p. 18\)](#).)

You can use `Query` with any table that has a composite primary key (partition key and sort key). You must specify an equality condition for the partition key, and you can optionally provide another condition for the sort key.

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*.

Here are some `Query` examples in DynamoDB:

```
// 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"
    }
}
```

```
// 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
{
    TableName: "Music",
```

```
KeyConditionExpression: "Artist = :a and contains(SongTitle, :t)",
FilterExpression: "price < :p",
ExpressionAttributeValues: {
    ":a": "No One You Know",
    ":t": "Today",
    ":p": 1.00
}
```

Note

For code samples using `Query`, see [Getting Started with DynamoDB \(p. 52\)](#).

Scanning a Table

In SQL, a `SELECT` statement without a `WHERE` clause will return every row in a table. In 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.

SQL

In 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 (*).

Here are some examples:

```
/* 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;
```

DynamoDB

DynamoDB provides a `Scan` action that works in a similar way. Here are some examples:

```
// 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` action also provides a `FilterExpression` parameter, to discard items that you do not want to appear in the results. A `FilterExpression` is applied after the entire table is scanned, 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 samples that use `scan`, see [Getting Started with DynamoDB \(p. 52\)](#).

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 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 \(p. 34\)](#)
- [Querying and Scanning an Index \(p. 35\)](#)

Creating an Index

SQL

In a relational database, an index is a data structure that let 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 can create an index like this:

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

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` actions on a table or a local secondary index. However, global secondary indexes only support eventual consistency.

You can add a global secondary index to an existing table, using the `UpdateTable` action 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 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` – 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` action to view this attribute.

Note

For code samples that use `updateTable`, see [Getting Started with DynamoDB \(p. 52\)](#).

Querying and Scanning an Index

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 systems (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;
```

DynamoDB

In DynamoDB, you perform `Query` operations directly on the index, in the same way that you would do so on a table. 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*.)

```
// All of the rock songs

{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre",
    ExpressionAttributeValues: {
        ":genre": "Rock"
    },
};
```

```
// 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"
};
```

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

You can also perform `Scan` operations on a secondary index, in the same way that you would do so on a table. The following is a scan on *GenreAndPriceIndex*:

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

{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex"
}
```

Modifying Data in a Table

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

SQL

In SQL, you 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. Here 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.

DynamoDB

In DynamoDB, you use the `UpdateItem` action to modify a single item. (If you want to modify multiple items, you must use multiple `UpdateItem` operations.)

Here is an example:

```
{
    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` replaces the entire item, rather than replacing individual attributes.

`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` action 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 auto-increment fields in SQL databases.

The following is an example of an `UpdateItem` action 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` action 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 samples using `UpdateItem`, see [Getting Started with DynamoDB \(p. 52\)](#).

Deleting Data from a Table

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

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. Here 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 following:

```
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.

DynamoDB

In DynamoDB, you use the `DeleteItem` action to delete data from a table, one item at a time. You must specify the item's primary key values. Here is an example:

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

Note

In addition to `DeleteItem`, Amazon DynamoDB supports a `BatchWriteItem` action 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` action 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 samples using `DeleteItem`, see [Getting Started with DynamoDB \(p. 52\)](#).

Removing a Table

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

SQL

When you no longer need a table and want to discard it permanently, you 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.)

DynamoDB

DynamoDB has a similar action: `DeleteTable`. In the following example, the table is permanently deleted:

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

Note

For code samples using `DeleteTable`, see [Getting Started with DynamoDB \(p. 52\)](#).

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 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 \(Downloadable Version\) \(p. 41\)](#)
- [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#)

Setting Up DynamoDB Local (Downloadable Version)

The downloadable version of DynamoDB lets you write 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 can make a few minor changes to the code so that it uses the DynamoDB web service.

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

Topics

- [Downloading and Running DynamoDB on Your Computer \(p. 41\)](#)
- [Setting the Local Endpoint \(p. 44\)](#)
- [Usage Notes \(p. 44\)](#)

Downloading and Running DynamoDB on Your Computer

The downloadable version of DynamoDB is provided as an executable .jar file. The application runs on Windows, Linux, macOS X, and other platforms that support Java.

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

1. Download DynamoDB for free using one of the following links:

Region	Download Links	Checksums
Asia Pacific (Mumbai) Region	.tar.gz .zip	.tar.gz.sha256 .zip.sha256
Asia Pacific (Singapore) Region	.tar.gz .zip	.tar.gz.sha256 .zip.sha256
Asia Pacific (Tokyo) Region	.tar.gz .zip	.tar.gz.sha256 .zip.sha256
EU (Frankfurt) Region	.tar.gz .zip	.tar.gz.sha256 .zip.sha256
South America (São Paulo) Region	.tar.gz .zip	.tar.gz.sha256 .zip.sha256
US West (Oregon) Region	.tar.gz .zip	.tar.gz.sha256 .zip.sha256

Downloadable DynamoDB is available on Apache Maven. For more information, see [DynamoDB \(Downloadable Version\)](#) and [Apache Maven \(p. 42\)](#), later in this topic. DynamoDB is also available as part of the AWS Toolkit for Eclipse. For more information, see [AWS Toolkit For Eclipse](#).

Important

To run DynamoDB on your computer, you must have the Java Runtime Environment (JRE) version 6.x or newer. The application doesn't run on older 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 type the following command:

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

Note

DynamoDB processes incoming requests until you stop it. To stop DynamoDB, type `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`, type this command:

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

After completing these steps, you can start writing applications.

DynamoDB (Downloadable Version) and Apache Maven

To use DynamoDB 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 DynamoDB Maven repository to your application's Project Object Model (POM) file:

```
<!--Dependency:-->
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>DynamoDBLocal</artifactId>
        <version>[1.11,2.0)</version>
    </dependency>
```

```

</dependencies>
<!--Custom repository:-->
<repositories>
    <repository>
        <id>dynamodb-local-oregon</id>
        <name>DynamoDB Local Release Repository</name>
        <url>https://s3-us-west-2.amazonaws.com/dynamodb-local/release</url>
    </repository>
</repositories>

```

Note

Alternatively, you can use one of the following repository URLs, depending on your region:

id	Repository URL
dynamodb-local-mumbai	https://s3.ap-south-1.amazonaws.com/dynamodb-local-mumbai/release
dynamodb-local-singapore	https://s3-ap-southeast-1.amazonaws.com/dynamodb-local-singapore/release
dynamodb-local-tokyo	https://s3.ap-northeast-1.amazonaws.com/dynamodb-local-tokyo/release
dynamodb-local-frankfurt	https://s3.eu-central-1.amazonaws.com/dynamodb-local-frankfurt/release
dynamodb-local-sao-paulo	https://s3-sa-east-1.amazonaws.com/dynamodb-local-sao-paulo/release

The aws-dynamodb-examples repository in GitHub contains examples for [starting and stopping DynamoDB Local](#) inside a Java program and [using DynamoDB Local in JUnit tests](#).

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` — DynamoDB uses a single database file instead of separate files for each credential and region. If you specify `-sharedDb`, all DynamoDB clients interact with the same set of tables regardless of their region and credential configuration.

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 to interact with downloadable DynamoDB. For example, you can use it to perform all the steps in [Creating Tables and Loading Sample Data \(p. 280\)](#).

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 \(p. 287\)](#)
- [.NET: Setting the AWS Region and Endpoint \(p. 289\)](#)

Note

For examples in other programming languages, see [Getting Started with DynamoDB \(p. 52\)](#).

Usage Notes

Except for the endpoint, applications that run with the downloadable version of 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 you have stored in it.

- If you omit `-sharedDb`, the database file is named `myaccesskeyid_region.db`, with the AWS access key ID and region as they appear in your application configuration. If you delete the file, you lose any data 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 `-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 simply by changing the endpoint you're using.

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:

- 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.
- Consumed capacity units are not tracked. In operation responses, nulls are returned instead of capacity units.
- 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.

Setting Up DynamoDB (Web Service)

To use the Amazon DynamoDB web service:

1. [Sign up for AWS. \(p. 46\)](#)
2. [Get an AWS access key \(p. 46\) \(used to access DynamoDB programmatically\).](#)

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 \(p. 48\)](#).

3. [Configure your credentials \(p. 47\) \(used to access DynamoDB programmatically\).](#)

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://aws.amazon.com/>, and then choose **Create an AWS Account**.

Note

This might be unavailable in your browser if you previously signed into the AWS Management Console. In that case, choose **Sign In to the Console**, and then choose **Create a new AWS account**.

2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a PIN using the phone keypad.

Getting an AWS Access Key

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

To get the access key ID and secret access key for an IAM user

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them from the AWS Management Console. We recommend that you use IAM access keys instead of AWS account root user access keys. IAM lets you securely control access to AWS services and resources in your AWS account.

The only time that you can view or download the secret access keys is when you create the keys. You cannot recover them later. However, you can create new access keys at any time. You must also have permissions to perform the required IAM actions. For more information, see [Delegating Permissions to Administer IAM Users, Groups, and Credentials](#) in the *IAM User Guide*.

1. Open the [IAM console](#).
2. In the navigation pane of the console, choose **Users**.
3. Choose your IAM user name (not the check box).
4. Choose the **Security credentials** tab and then choose **Create access key**.
5. To see the new access key, choose **Show**. Your credentials will look something like this:

- Access key ID: AKIAIOSFODNN7EXAMPLE
 - Secret access key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
6. To download the key pair, choose **Download .csv file**. Store the keys in a secure location.
- Keep the keys confidential in order to protect your account, and never email them. Do not share them outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

Related topics

- [What Is IAM?](#) in the *IAM User Guide*
- [AWS Security Credentials](#) in *AWS General Reference*

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 AWS access key ID and secret access key. You can also use the `aws configure` command of the AWS CLI to automatically create the file. Alternatively, you can use environment variables. For more information on configuring your credentials, see the programming-specific AWS SDK developer guide.

To install and configure the AWS CLI, see [Using the CLI \(p. 49\)](#).

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 \(p. 48\)](#)
- [Using the CLI \(p. 49\)](#)
- [Using the API \(p. 51\)](#)

Using the Console

You can access the AWS Management Console for DynamoDB here: <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.
- 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, from 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 stream and table details, and manage streams and Time To Live (TTL).
- **Items** – Manage items and perform queries and scans.
- **Metrics** – Monitor CloudWatch metrics.
- **Alarms** – Manage CloudWatch alarms.
- **Capacity** – Modify a table's provisioned capacity.
- **Indexes** – Manage global secondary indexes.
- **Triggers** – Manage triggers to connect DynamoDB streams to Lambda functions.
- **Access control** – Set up fine-grained access control with web identity federation.
- **Tags** – Apply tags to your resources to help organize and identify them.

Using the 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 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 [Getting an AWS Access Key \(p. 46\)](#).

For a complete listing of all the commands available for DynamoDB in the AWS CLI, go to <http://docs.aws.amazon.com/cli/latest/reference/dynamodb/index.html>.

Topics

- [Downloading and Configuring the AWS CLI \(p. 49\)](#)
- [Using the AWS CLI with DynamoDB \(p. 49\)](#)
- [Using the AWS CLI with Downloadable DynamoDB \(p. 50\)](#)

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
```

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 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 Downloadable DynamoDB

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

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

Here is an example that 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 downloadable version of DynamoDB 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 \(p. 52\)](#).

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 \(Web Service\) \(p. 46\)](#).

For a high-level overview of DynamoDB application programming with the AWS SDKs, see [Programming with DynamoDB and the AWS SDKs \(p. 180\)](#).

Getting Started with DynamoDB

This section contains hands-on tutorials to help you learn about Amazon DynamoDB. We encourage you to work through one of the language-specific tutorials. The sample code in these tutorials can run against either the downloadable version of DynamoDB or the Amazon DynamoDB web service.

Note

AWS SDKs are available for a wide variety of languages. For a complete list, see [Tools for Amazon Web Services](#).

Topics

- [Java and DynamoDB \(p. 52\)](#)
- [JavaScript and DynamoDB \(p. 69\)](#)
- [Node.js and DynamoDB \(p. 89\)](#)
- [.NET and DynamoDB \(p. 103\)](#)
- [PHP and DynamoDB \(p. 132\)](#)
- [Python and DynamoDB \(p. 149\)](#)
- [Ruby and DynamoDB \(p. 164\)](#)

Java and DynamoDB

In this tutorial, you use the AWS SDK for Java to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

The SDK for Java offers several programming models for different use cases. In this exercise, the Java code uses the document model that provides a level of abstraction that makes it easier for you to work with JSON documents.

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary \(p. 68\)](#).

As you work through this tutorial, you can refer to the [AWS SDK for Java API Reference](#).

Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
DynamoDB (Downloadable Version) is also available as part of the AWS Toolkit for Eclipse. For more information, see [AWS Toolkit For Eclipse](#).
- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Set up the AWS SDK for Java:
 - Install a Java development environment. If you are using the Eclipse IDE, install the AWS Toolkit for Eclipse.
 - Install the AWS SDK for Java.
 - Set up your AWS security credentials for use with the SDK for Java.

For instructions, see [Getting Started](#) in the [AWS SDK for Java Developer Guide](#).

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `ScalarAttributeType` is `N` for number.
- `title` – The sort key. The `ScalarAttributeType` is `S` for string.

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
  
package com.amazonaws.codesamples.gsg;  
  
import java.util.Arrays;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
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.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;  
  
public class MoviesCreateTable {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);
```

```
String tableName = "Movies";

try {
    System.out.println("Attempting to create table; please wait...");
    Table table = dynamoDB.createTable(tableName,
        Arrays.asList(new KeySchemaElement("year", KeyType.HASH), // Partition
                      // key
                      new KeySchemaElement("title", KeyType.RANGE)), // Sort key
        Arrays.asList(new AttributeDefinition("year", ScalarAttributeType.N),
                      new AttributeDefinition("title", ScalarAttributeType.S)),
        new ProvisionedThroughput(10L, 10L));
    table.waitForActive();
    System.out.println("Success. Table status: " +
table.getDescription().getTableStatus());

}
catch (Exception e) {
    System.err.println("Unable to create table: ");
    System.err.println(e.getMessage());
}

}
```

Note

- You set the endpoint to indicate that you are creating the table in DynamoDB on your computer.
 - In the `createTable` call, you specify table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)

2. Compile and run the program.

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- Step 2.1: Download the Sample Data File (p. 55)
 - Step 2.2: Load the Sample Data into the Movies Table (p. 55)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ...
```

```
        "title" : ....,
        "info" : { ... }
    },
...
]
```

In the JSON data, note the following:

- You use the `year` and `title` as the primary key attribute values for the `Movies` table.
- You store the rest of the `info` values in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{
    "year" : 2013,
    "title" : "Turn It Down, Or Else!",
    "info" : {
        "directors" : [
            "Alice Smith",
            "Bob Jones"
        ],
        "release_date" : "2013-01-18T00:00:00Z",
        "rating" : 6.2,
        "genres" : [
            "Comedy",
            "Drama"
        ],
        "image_url" : "http://ia.media-imdb.com/images/N/
O9ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",
        "rank" : 11,
        "running_time_secs" : 5215,
        "actors" : [
            "David Matthewman",
            "Ann Thomas",
            "Jonathan G. Neff"
        ]
    }
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
package com.amazonaws.codesamples.gsg;

import java.io.File;
import java.util.Iterator;

import com.amazonaws.client.builder.AwsClientBuilder;
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.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class MoviesLoadData {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        JsonParser parser = new JsonFactory().createParser(new File("moviedata.json"));

        JsonNode rootNode = new ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();

        ObjectNode currentNode;

        while (iter.hasNext()) {
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();

            try {
                table.putItem(new Item().withPrimaryKey("year", year, "title",
                title).withJSON("info",
                    currentNode.path("info").toString()));
                System.out.println("PutItem succeeded: " + year + " " + title);

            }
            catch (Exception e) {
                System.err.println("Unable to add movie: " + year + " " + title);
                System.err.println(e.getMessage());
                break;
            }
        }
        parser.close();
    }
}
```

This program uses the open source Jackson library to process JSON. Jackson is included in the AWS SDK for Java. You don't have to install it separately.

2. Compile and run the program.

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 57\)](#)
- [Step 3.2: Read an Item \(p. 58\)](#)
- [Step 3.3: Update an Item \(p. 59\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 61\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 62\)](#)
- [Step 3.6: Delete an Item \(p. 63\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy and paste the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.util.HashMap;
import java.util.Map;

import com.amazonaws.client.builder.AwsClientBuilder;
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.PutItemOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class MoviesItemOps01 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        final Map<String, Object> infoMap = new HashMap<String, Object>();
        infoMap.put("plot", "Nothing happens at all.");
        infoMap.put("rating", 0);

        try {
            System.out.println("Adding a new item...");
            PutItemOutcome outcome = table.putItem(infoMap);
            System.out.println("Item added with ID: " + outcome.getItemId());
        }
    }
}
```

```
        PutItemOutcome outcome = table
            .putItem(new Item().withPrimaryKey("year", year, "title",
title).withMap("info", infoMap));

        System.out.println("PutItem succeeded:\n" + outcome.getPutItemResult());

    }
    catch (Exception e) {
        System.err.println("Unable to add item: " + year + " " + title);
        System.err.println(e.getMessage());
    }
}
```

Note

The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.

- ## 2. Compile and run the program.

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{  
  year: 2015,  
  title: "The Big New Movie",  
  info: {  
    plot: "Nothing happens at all.",  
    rating: 0  
  }  
}
```

You can use the `getItem` method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
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 MoviesItemOps02 {

    public static void main(String[] args) throws Exception {
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();
    }
}
```

```

DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("Movies");

int year = 2015;
String title = "The Big New Movie";

GetItemSpec spec = new GetItemSpec().withPrimaryKey("year", year, "title",
    title);

try {
    System.out.println("Attempting to read the item...");
    Item outcome = table.getItem(spec);
    System.out.println("GetItem succeeded: " + outcome);

}
catch (Exception e) {
    System.err.println("Unable to read item: " + year + " " + title);
    System.err.println(e.getMessage());
}

}
}

```

2. Compile and run the program.

Step 3.3: Update an Item

You can use the `updateItem` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

To the following:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
    }
}
```

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
package com.amazonaws.codesamples.gsg;

import java.util.Arrays;

import com.amazonaws.client.builder.AwsClientBuilder;
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.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class MoviesItemOps03 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("year",
year, "title", title)
            .withUpdateExpression("set info.rating = :r, info.plot=:p, info.actors=:a")
            .WithValueMap(new ValueMap().withNumber(":r", 5.5).withString(":p",
"Everything happens all at once."))
            .withList(":a", Arrays.asList("Larry", "Moe", "Curly")))
            .withReturnValues(ReturnValue.UPDATED_NEW);

        try {
            System.out.println("Updating the item...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
outcome.getItem().toJSONPretty());
        }
        catch (Exception e) {
            System.err.println("Unable to update item: " + year + " " + title);
            System.err.println(e.getMessage());
        }
    }
}
```

Note

This program uses an `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. Compile and run the program.

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `updateItem` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
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.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class MoviesItemOps04 {

    public static void main(String[] args) throws Exception {
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("year",
year, "title", title)
            .withUpdateExpression("set info.rating = info.rating + :val")
            .WithValueMap(new ValueMap().withNumber(":val",
1)).withReturnValues(ReturnValue.UPDATED_NEW);

        try {
            System.out.println("Incrementing an atomic counter...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
outcome.getItem().toJSONPretty());
        }
        catch (Exception e) {
            System.err.println("Unable to update item: " + year + " " + title);
            System.err.println(e.getMessage());
        }
    }
}
```

2. Compile and run the program.

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the movie item is updated only if there are more than three actors.

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
  
package com.amazonaws.codesamples.gsg;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
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.PrimaryKey;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;  
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;  
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;  
import com.amazonaws.services.dynamodbv2.model.ReturnValue;  
  
public class MoviesItemOps05 {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        Table table = dynamoDB.getTable("Movies");  
  
        int year = 2015;  
        String title = "The Big New Movie";  
  
        UpdateItemSpec updateItemSpec = new UpdateItemSpec()  
            .withPrimaryKey(new PrimaryKey("year", year, "title",  
                title)).withUpdateExpression("remove info.actors[0]")  
            .withConditionExpression("size(info.actors) > :num").withValueMap(new  
                ValueMap().withNumber(":num", 3))  
            .withReturnValues(ReturnValue.UPDATED_NEW);  
  
        // Conditional update (we expect this to fail)  
        try {  
            System.out.println("Attempting a conditional update...");  
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);  
            System.out.println("UpdateItem succeeded:\n" +  
                outcome.getItem().toJSONPretty());  
  
        }  
        catch (Exception e) {  
            System.err.println("Unable to update item: " + year + " " + title);  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

2. Compile and run the program.

The program should fail with the following message:

```
The conditional request failed
```

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the `ConditionExpression` looks like this:

```
.withConditionExpression("size(info.actors) >= :num")
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Compile and run the program. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item

You can use the `deleteItem` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
  
package com.amazonaws.codesamples.gsg;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
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.PrimaryKey;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;  
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;  
  
public class MoviesItemOps06 {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        Table table = dynamoDB.getTable("Movies");  
  
        int year = 2015;  
        String title = "The Big New Movie";  
  
        DeleteItemSpec deleteItemSpec = new DeleteItemSpec()  
            .withPrimaryKey(new PrimaryKey("year", year, "title",  
                title)).withConditionExpression("info.rating <= :val")  
            .WithValueMap(new ValueMap().withNumber(":val", 5.0));  
  
        // Conditional delete (we expect this to fail)
```

```
try {
    System.out.println("Attempting a conditional delete...");
    table.deleteItem(deleteItemSpec);
    System.out.println("DeleteItem succeeded");
}
catch (Exception e) {
    System.err.println("Unable to delete item: " + year + " " + title);
    System.err.println(e.getMessage());
}
}
```

2. Compile and run the program.

The program should fail with the following message:

```
The conditional request failed
```

This is because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition in DeleteItemSpec.

```
DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
    .withPrimaryKey(new PrimaryKey("year", 2015, "title", "The Big New Movie"));
```

4. Compile and run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key); for example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to `query`, there is also a `scan` method that can retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query \(p. 64\)](#)
- [Step 4.2: Scan \(p. 66\)](#)

Step 4.1: Query

The code included in this step performs the following queries:

- Retrieve all movies released in the `year` 1985.
- Retrieve all movies released in the `year` 1992, with a `title` beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
  
package com.amazonaws.codesamples.gsg;  
  
import java.util.HashMap;  
import java.util.Iterator;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
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.QueryOutcome;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;  
  
public class MoviesQuery {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        Table table = dynamoDB.getTable("Movies");  
  
        HashMap<String, String> nameMap = new HashMap<String, String>();  
        nameMap.put("#yr", "year");  
  
        HashMap<String, Object> valueMap = new HashMap<String, Object>();  
        valueMap.put(":yyyy", 1985);  
  
        QuerySpec querySpec = new QuerySpec().withKeyConditionExpression("#yr  
= :yyyy").withNameMap(nameMap)  
            .withValueMap(valueMap);  
  
        ItemCollection<QueryOutcome> items = null;  
        Iterator<Item> iterator = null;  
        Item item = null;  
  
        try {  
            System.out.println("Movies from 1985");  
            items = table.query(querySpec);  
  
            iterator = items.iterator();  
            while (iterator.hasNext()) {  
                item = iterator.next();  
                System.out.println(item.getNumber("year") + ": " +  
item.getString("title"));  
            }  
  
        }  
        catch (Exception e) {  
            System.err.println("Unable to query movies from 1985");  
            System.err.println(e.getMessage());  
        }  
  
        valueMap.put(":yyyy", 1992);  
    }  
}
```

```
valueMap.put(":letter1", "A");
valueMap.put(":letter2", "L");

querySpec.withProjectionExpression("#yr, title, info.genres, info.actors[0]")
    .withKeyConditionExpression("#yr = :yyyy and title between :letter1
and :letter2").withNameMap(nameMap)
    .withValueMap(valueMap);

try {
    System.out.println("Movies from 1992 - titles A-L, with genres and lead
actor");
    items = table.query(querySpec);

    iterator = items.iterator();
    while (iterator.hasNext()) {
        item = iterator.next();
        System.out.println(item.getNumber("year") + ": " +
item.getString("title") + " " + item.getMap("info"));
    }
}

catch (Exception e) {
    System.err.println("Unable to query movies from 1992:");
    System.err.println(e.getMessage());
}
}
```

Note

- `nameMap` provides name substitution. This is used because `year` is a reserved word in DynamoDB—you cannot use it directly in any expression, including `KeyConditionExpression`. You use the expression attribute name `#yr` to address this.
 - `valueMap` provides value substitution. This is used because you can't use literals in any expression, including `KeyConditionExpression`. You use the expression attribute value `:yyyy` to address this.

First, you create the `queryspec` object, which describes the query parameters, and then you pass the object to the `query` method.

- ## 2. Compile and run the program.

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Scan

The `scan` method reads every item in the entire table, and returns all the data in the table. You can provide an optional `filter_expression` so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all the others.

1. Copy and paste the following program into your Java development environment:

```

// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.util.Iterator;

import com.amazonaws.client.builder.AwsClientBuilder;
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.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesScan {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        ScanSpec scanSpec = new ScanSpec().withProjectionExpression("#yr, title,
info.rating")
            .withFilterExpression("#yr between :start_yr and :end_yr").withNameMap(new
NameMap().with("#yr", "year"))
            .WithValueMap(new ValueMap().withNumber(":start_yr",
1950).withNumber(":end_yr", 1959));

        try {
            ItemCollection<ScanOutcome> items = table.scan(scanSpec);

            Iterator<Item> iter = items.iterator();
            while (iter.hasNext()) {
                Item item = iter.next();
                System.out.println(item.toString());
            }

        }
        catch (Exception e) {
            System.err.println("Unable to scan the table:");
            System.err.println(e.getMessage());
        }
    }
}

```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.

2. Compile and run the program.

Note

You can also use the `Scan` operation with any secondary indexes that you created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into your Java development environment:

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
  
package com.amazonaws.codesamples.gsg;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
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;  
  
public class MoviesDeleteTable {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        Table table = dynamoDB.getTable("Movies");  
  
        try {  
            System.out.println("Attempting to delete table; please wait...");  
            table.delete();  
            table.waitForDelete();  
            System.out.print("Success.");  
  
        }  
        catch (Exception e) {  
            System.err.println("Unable to delete table: ");  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

2. Compile and run the program.

Summary

In this tutorial, you created the `Movies` table in DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the Amazon DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB service, you must change the endpoint in your application.

1. Remove the following import:

```
import com.amazonaws.client.builder.AwsClientBuilder;
```

2. Next, go to `AmazonDynamoDB` in the code:

```
AmazonDynamoDB client =
    AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
    .build();
```

3. Now modify the client so that it accesses an AWS region instead of a specific endpoint:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.REGION)
    .build();
```

For example, if you want to access the `us-west-2` region, you would do this:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.US_WEST_2)
    .build();
```

Instead of using DynamoDB on your computer, the program now uses the Amazon DynamoDB web service endpoint in US West (Oregon).

Amazon DynamoDB is available in several regions worldwide. For the complete list, see [Regions and Endpoints](#) in the *AWS General Reference*. For more information about setting regions and endpoints in your code, see [AWS Region Selection](#) in the *AWS SDK for Java Developer Guide*.

JavaScript and DynamoDB

In this tutorial, you use JavaScript to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary \(p. 88\)](#).

As you work through this tutorial, you can refer to the [AWS SDK for JavaScript API Reference](#).

Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
- Set up an AWS access key to use AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Set up the AWS SDK for JavaScript. To do this, add or modify the following script tag to your HTML pages:

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>
```

Note

The version of AWS SDK for JavaScript might have been updated. For the latest version, see the [AWS SDK for JavaScript API Reference](#).

- Enable [cross-origin resource sharing \(CORS\)](#) so your computer's browser can communicate with the downloadable version of DynamoDB.

To enable CORS

1. Download the free ModHeader Chrome browser extension (or any other browser extension that allows you to modify HTTP response headers).
2. Run the ModHeader Chrome browser extension, and add an HTTP response header with the name set to "Access-Control-Allow-Origin" and a value of "null" or "*".

Important

This configuration is required only while running this tutorial program for JavaScript on your computer. After you finish the tutorial, you should disable or remove this configuration.

3. You can now run the JavaScript tutorial program files.

If you prefer to run a complete version of the JavaScript tutorial program instead of performing step-by-step instructions, do the following:

1. Download the following file: [MoviesJavaScript.zip](#).
2. Extract the file `MoviesJavaScript.html` from the archive.
3. Modify the `MoviesJavaScript.html` file to use your endpoint.
4. Run the `MoviesJavaScript.html` file.

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy and paste the following program into a file named `MoviesCreateTable.html`:

```
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    // Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    // DynamoDB.
```

```

// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var dynamodb = new AWS.DynamoDB();

function createMovies() {
    var params = {
        TableName : "Movies",
        KeySchema: [
            { AttributeName: "year", KeyType: "HASH" },
            { AttributeName: "title", KeyType: "RANGE" }
        ],
        AttributeDefinitions: [
            { AttributeName: "year", AttributeType: "N" },
            { AttributeName: "title", AttributeType: "S" }
        ],
        ProvisionedThroughput: {
            ReadCapacityUnits: 5,
            WriteCapacityUnits: 5
        }
    };
    dynamodb.createTable(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to create table: " +
                "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "Created table: " + "\n" +
                JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="createTableButton" type="button" value="Create Table"
    onclick="createMovies();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

Note

- You set the endpoint to indicate that you are creating the table in DynamoDB on your computer.
 - In the `createMovies` function, you specify the table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this tutorial.)
2. Open the `MoviesCreateTable.html` file in your browser.
 3. Choose **Create Table**.

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 73\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 73\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ... ,  
    "title" : ....,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{  
  "year" : 2013,  
  "title" : "Turn It Down, Or Else!",  
  "info" : {  
    "directors" : [  
      "Alice Smith",  
      "Bob Jones"  
    ],  
    "release_date" : "2013-01-18T00:00:00Z",  
    "rating" : 6.2,  
    "genres" : [  
      "Comedy",  
      "Drama"  
    ],  
    "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
    "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
    "rank" : 11,  
    "running_time_secs" : 5215,  
    "actors" : [  
      "David Matthewman",  
      "John Doe",  
      "Jane Smith"  
    ]  
  }  
}
```

```
        "Ann Thomas",
        "Jonathan G. Neff"
    ]
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy and paste the following program into a file named `MoviesLoadData.html`:

```
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script type="text/javascript">
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function processFile(evt) {
    document.getElementById('textarea').innerHTML = "";
    document.getElementById('textarea').innerHTML += "Importing movies into DynamoDB.
Please wait..." + "\n";
    var file = evt.target.files[0];
    if (file) {
        var r = new FileReader();
        r.onload = function(e) {
            var contents = e.target.result;
            var allMovies = JSON.parse(contents);

            allMovies.forEach(function (movie) {
                document.getElementById('textarea').innerHTML += "Processing: " +
movie.title + "\n";
                var params = {
                    TableName: "Movies",
                    Item: {
                        "year": movie.year,
                        "title": movie.title,
                        "info": movie.info
                    }
                }
            })
        }
    }
}
```

```
        };
        docClient.put(params, function (err, data) {
            if (err) {
                document.getElementById('textarea').innerHTML += "Unable to add movie: " + count + movie.title + "\n";
                document.getElementById('textarea').innerHTML += "Error JSON: " + JSON.stringify(err) + "\n";
            } else {
                document.getElementById('textarea').innerHTML += "PutItem succeeded: " + movie.title + "\n";
                textarea.scrollTop = textarea.scrollHeight;
            }
        });
    };
    r.readAsText(file);
} else {
    alert("Could not read movie data file");
}
}

</script>
</head>

<body>
<input type="file" id="fileinput" accept='application/json' />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

<script>
    document.getElementById('fileinput').addEventListener('change', processFile,
    false);
</script>
</body>
</html>
```

2. Open the `MoviesLoadData.html` file in your browser.
3. Choose **Browse**, and load the `moviedata.json` file.

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 74\)](#)
- [Step 3.2: Read an Item \(p. 76\)](#)
- [Step 3.3: Update an Item \(p. 77\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 79\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 80\)](#)
- [Step 3.6: Delete an Item \(p. 81\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy and paste the following program into a file named `MoviesItemOps01.html`:

```

<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function createItem() {
    var params = {
        TableName :"Movies",
        Item:{ 
            "year": 2015,
            "title": "The Big New Movie",
            "info":{ 
                "plot": "Nothing happens at all.",
                "rating": 0
            }
        }
    };
    docClient.put(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to add item: " +
            "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "PutItem succeeded: " +
            "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="createItem" type="button" value="Create Item" onclick="createItem();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

Note

The primary key is required. This code adds an item that has a primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.

2. Open the `MoviesItemOps01.html` file in your browser.
3. Choose **Create Item**.

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Nothing happens at all.",  
        rating: 0  
    }  
}
```

You can use the `get` method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into a file named `MoviesItemOps02.html`:

```
<html>  
<head>  
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>  
  
<script>  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: 'http://localhost:8000',  
    // accessKeyId default can be used while using the downloadable version of DynamoDB.  
  
    // For security reasons, do not store AWS Credentials in your files. Use Amazon  
    Cognito instead.  
    accessKeyId: "fakeMyKeyId",  
    // secretAccessKey default can be used while using the downloadable version of  
    // DynamoDB.  
    // For security reasons, do not store AWS Credentials in your files. Use Amazon  
    Cognito instead.  
    secretAccessKey: "fakeSecretAccessKey"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient();  
  
function readItem() {  
    var table = "Movies";  
    var year = 2015;  
    var title = "The Big New Movie";  
  
    var params = {  
        TableName: table,  
        Key:{  
            "year": year,  
            "title": title  
        }  
    };  
    docClient.get(params, function(err, data) {  
        if (err) {  
            document.getElementById('textarea').innerHTML = "Unable to read item: " +  
"\n" + JSON.stringify(err, undefined, 2);  
        } else {  
            document.getElementById('textarea').innerHTML = "GetItem succeeded: " +  
"\n" + JSON.stringify(data, undefined, 2);  
        }  
    });  
}
```

```

</script>
</head>

<body>
<input id="readItem" type="button" value="Read Item" onclick="readItem();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

2. Open the `MoviesItemOps02.html` file in your browser.
3. Choose **Read Item**.

Step 3.3: Update an Item

You can use the `update` method to modify an item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}
```

To the following:

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Everything happens all at once.",
    rating: 5.5,
    actors: ["Larry", "Moe", "Curly"]
  }
}
```

1. Copy and paste the following program into a file named `MoviesItemOps03.html`:

```

<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
  region: "us-west-2",
  endpoint: 'http://localhost:8000',
  // accessKeyId default can be used while using the downloadable version of DynamoDB.

```

```

// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
accessKeyId: "fakeMyKeyId",
// secretAccessKey default can be used while using the downloadable version of
DynamoDB.
// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function updateItem() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    var params = {
        TableName:table,
        Key:{
            "year": year,
            "title": title
        },
        UpdateExpression: "set info.rating = :r, info.plot=:p, info.actors=:a",
        ExpressionAttributeValues:{
            ":r":5.5,
            ":p":"Everything happens all at once.",
            ":a":["Larry", "Moe", "Curly"]
        },
        ReturnValue:"UPDATED_NEW"
    };

    docClient.update(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to update item: " +
"\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "UpdateItem succeeded: " +
"\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="updateItem" type="button" value="Update Item" onclick="updateItem();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValue` parameter instructs DynamoDB to return only the updated attributes ("`UPDATED_NEW`").

2. Open the `MoviesItemOps03.html` file in your browser.
3. Choose **Update Item**.

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `update` method to increment or decrement the value of an attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy and paste the following program into a file named `MoviesItemOps04.html`:

```
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function increaseRating() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    var params = {
        TableName:table,
        Key:{
            "year": year,
            "title": title
        },
        UpdateExpression: "set info.rating = info.rating + :val",
        ExpressionAttributeValues:{
            ":val":1
        },
        ReturnValues:"UPDATED_NEW"
    };

    docClient.update(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to update rating: " +
            "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "Increase Rating succeeded:
" + "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>
```

```
<body>
<input id="increaseRating" type="button" value="Increase Rating"
    onclick="increaseRating();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>

</body>
</html>
```

2. Open the `MoviesItemOps04.html` file in your browser.
3. Choose **Increase Rating**.

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors in the movie.

1. Copy and paste the following program into a file named `MoviesItemOps05.html`:

```
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function conditionalUpdate() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    // Conditional update (will fail)
    var params = {
        TableName:table,
        Key:{
            "year": year,
            "title": title
        },
        UpdateExpression: "remove info.actors[0]",
        ConditionExpression: "size(info.actors) > :num",
        ExpressionAttributeValues:{
            ":num":3
        },
        ReturnValues:"UPDATED_NEW"
    }
}
```

```

};

docClient.update(params, function(err, data) {
    if (err) {
        document.getElementById('textarea').innerHTML = "The conditional update
failed: " + "\n" + JSON.stringify(err, undefined, 2);
    } else {
        document.getElementById('textarea').innerHTML = "The conditional update
succeeded: " + "\n" + JSON.stringify(data, undefined, 2);
    }
});

</script>
</head>

<body>
<input id="conditionalUpdate" type="button" value="Conditional Update"
onclick="conditionalUpdate();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

2. Open the `MoviesItemOps05.html` file in your browser.
3. Choose **Conditional Update**.

The program should fail with the following message:

`The conditional update failed`

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

4. Modify the program so that the `ConditionExpression` looks like this:

`ConditionExpression: "size(info.actors) >= :num",`

The condition is now *greater than or equal to 3* instead of *greater than 3*.

5. Run the program again. The `updateItem` operation should now succeed.

Step 3.6: Delete an Item

You can use the `delete` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into a file named `MoviesItemOps06.html`:

```

<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

```

```

// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
accessKeyId: "fakeMyKeyId",
// secretAccessKey default can be used while using the downloadable version of
DynamoDB.
// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function conditionalDelete() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    var params = {
        TableName:table,
        Key:{
            "year":year,
            "title":title
        },
        ConditionExpression:"info.rating <= :val",
        ExpressionAttributeValues: {
            ":val": 5.0
        }
    };

    docClient.delete(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "The conditional delete
failed: " + "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "The conditional delete
succeeded: " + "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="conditionalDelete" type="button" value="Conditional Delete"
onclick="conditionalDelete();;" />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

2. Open the `MoviesItemOps06.html` file on your browser.
3. Choose **Conditional Delete**.

The program should fail with the following message:

The conditional delete failed

This is because the rating for this particular movie is greater than 5.

4. Modify the program to remove the condition from `params`.

```

var params = {
    TableName:table,

```

```
    Key:{  
        "title":title,  
        "year":year  
    }  
};
```

5. Run the program again. The delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value; the sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key); for example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to `query`, there is also a `scan` method that can retrieve all the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 83\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 85\)](#)
- [Step 4.3: Scan \(p. 86\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the `year` 1985.

1. Copy and paste the following program into a file named `MoviesQuery01.html`:

```
<html>  
<head>  
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>  
  
<script>  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: 'http://localhost:8000',  
    // accessKeyId default can be used while using the downloadable version of DynamoDB.  
  
    // For security reasons, do not store AWS Credentials in your files. Use Amazon  
    // Cognito instead.  
    accessKeyId: "fakeMyKeyId",  
    // secretAccessKey default can be used while using the downloadable version of  
    // DynamoDB.  
    // For security reasons, do not store AWS Credentials in your files. Use Amazon  
    // Cognito instead.  
    secretAccessKey: "fakeSecretAccessKey"
```

```

});
```

```

var docClient = new AWS.DynamoDB.DocumentClient();
```

```

function queryData() {
    document.getElementById('textarea').innerHTML += "Querying for movies from 1985.";
```

```

    var params = {
        TableName : "Movies",
        KeyConditionExpression: "#yr = :yyyy",
        ExpressionAttributeNames:{
            "#yr": "year"
        },
        ExpressionAttributeValues: {
            ":yyyy":1985
        }
    };

```

```

    docClient.query(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML += "Unable to query. Error: " +
+ "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML += "Querying for movies from
1985: " + "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}
```

```

</script>
</head>
```

```

<body>
<input id="queryData" type="button" value="Query" onclick="queryData();" />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
```

```

</body>
</html>

```

Note

`ExpressionAttributeNames` provides name substitution. This is used because `year` is a reserved word in DynamoDB—you can't use it directly in any expression, including `KeyConditionExpression`. For this reason, you use the expression attribute name `#yr`. `ExpressionAttributeValues` provides value substitution. This is used because you can't use literals in any expression, including `KeyConditionExpression`. For this reason, you use the expression attribute value `:yyyy`.

2. Open the `MoviesQuery01.html` file in your browser.
3. Choose **Query**.

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in year 1992, with title beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into a file named `MoviesQuery02.html`:

```

<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function queryData() {
    document.getElementById('textarea').innerHTML += "Querying for movies from 1985.";

    var params = {
        TableName : "Movies",
        ProjectionExpression:"#yr, title, info.genres, info.actors[0]",
        KeyConditionExpression: "#yr = :yyyy and title between :letter1 and :letter2",
        ExpressionAttributeNames:{
            "#yr": "year"
        },
        ExpressionAttributeValues: {
            ":yyyy":1992,
            ":letter1": "A",
            ":letter2": "L"
        }
    };

    docClient.query(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML += "Unable to query. Error: " +
                "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML += "Querying for movies
from 1992 - titles A-L, with genres and lead actor: " + "\n" + JSON.stringify(data,
undefined, 2);
        }
    });
}

</script>
</head>

<body>

```

```
<input id="queryData" type="button" value="Query" onclick="queryData();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>
</body>
</html>
```

2. Open the `MoviesQuery02.html` file on your browser.
3. Choose **Query**.

Step 4.3: Scan

The `scan` method reads every item in the entire table, and returns all the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The `scan` specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all the others.

1. Copy and paste the following program into a file named `MoviesScan.html`:

```
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.

    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function scanData() {
    document.getElementById('textarea').innerHTML += "Scanning Movies table." + "\n";

    var params = {
        TableName: "Movies",
        ProjectionExpression: "#yr, title, info.rating",
        FilterExpression: "#yr between :start_yr and :end_yr",
        ExpressionAttributeNames: {
            "#yr": "year",
        },
        ExpressionAttributeValues: {
            ":start_yr": 1950,
            ":end_yr": 1959
        }
    };

    docClient.scan(params, onScan);
}
```

```

        function onScan(err, data) {
            if (err) {
                document.getElementById('textarea').innerHTML += "Unable to scan the table:
" + "\n" + JSON.stringify(err, undefined, 2);
            } else {
                // Print all the movies
                document.getElementById('textarea').innerHTML += "Scan succeeded. " +
"\n";
                data.Items.forEach(function(movie) {
                    document.getElementById('textarea').innerHTML += movie.year + ": " +
movie.title + " - rating: " + movie.info.rating + "\n";
                });

                // Continue scanning if we have more movies (per scan 1MB limitation)
                document.getElementById('textarea').innerHTML += "Scanning for more..." +
"\n";
                params.ExclusiveStartKey = data.LastEvaluatedKey;
                docClient.scan(params, onScan);
            }
        }
    }

</script>
</head>

<body>
<input id="scanData" type="button" value="Scan" onclick="scanData();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
 - `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
2. Open the `MoviesScan.html` file in your browser.
 3. Choose **Scan**.

Note

You can also use the `Scan` operation with any secondary indexes that you create on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional): Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into a file named `MoviesDeleteTable.html`:

```

<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',

```

```

// accessKeyId default can be used while using the downloadable version of DynamoDB.

// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
accessKeyId: "fakeMyKeyId",
// secretAccessKey default can be used while using the downloadable version of
DynamoDB.
// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var dynamodb = new AWS.DynamoDB();

function deleteMovies() {
    var params = {
        TableName : "Movies"
    };

    dynamodb.deleteTable(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to delete table: "
+ "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "Table deleted.";
        }
    });
}

</script>
</head>

<body>
<input id="deleteTableButton" type="button" value="Delete Table"
onclick="deleteMovies();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

2. Open the `MoviesDeleteTable.html` file in your browser.
3. Choose **Delete Table**.

Summary

In this tutorial, you created the `Movies` table in DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the Amazon DynamoDB web service.

Updating the AWS Configuration Region to Use the DynamoDB Web Service

To use the DynamoDB web service, you must update the Region in your application. You also have to make sure that Amazon Cognito is available in that same Region so that your browser scripts can authenticate successfully.

```
AWS.config.update({region: "aws-region"});
```

For example, if you want to use the us-west-2 region, set the following region:

```
AWS.config.update({region: "us-west-2"});
```

The program now uses the Amazon DynamoDB web service region in US West (Oregon).

DynamoDB is available in several regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information about setting regions and endpoints in your code, see [Setting the Region](#) in the [AWS SDK for JavaScript Getting Started Guide](#).

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. Amazon Cognito uses AWS Identity and Access Management (IAM) roles to generate temporary credentials for your application's authenticated and unauthenticated users.

For more information, see [Configure AWS Credentials in Your Files Using Amazon Cognito \(p. 693\)](#).

Node.js and DynamoDB

In this tutorial, you use the AWS SDK for JavaScript to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary \(p. 103\)](#).

As you work through this tutorial, you can refer to the [AWS SDK for JavaScript API Reference](#).

Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Set up the AWS SDK for JavaScript:
 - Go to <http://nodejs.org> and install Node.js.
 - Go to <https://aws.amazon.com/sdk-for-node-js> and install the AWS SDK for JavaScript.

For more information, see the [AWS SDK for JavaScript Getting Started Guide](#).

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.

- **title** – The sort key. The `AttributeType` is `S` for string.

1. Copy and paste the following program into a file named `MoviesCreateTable.js`.

```
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var dynamodb = new AWS.DynamoDB();

var params = {
    TableName : "Movies",
    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
    }
};

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));
    }
});
```

Note

- You set the endpoint to indicate that you are creating the table in DynamoDB on your computer.
- In the `createTable` call, you specify table name, primary key attributes, and its data types.
- The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)

2. To run the program, type the following command:

```
node MoviesCreateTable.js
```

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 92\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 92\)](#)

We use a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{  
  "year" : 2013,  
  "title" : "Turn It Down, Or Else!",  
  "info" : {  
    "directors" : [  
      "Alice Smith",  
      "Bob Jones"  
    ],  
    "release_date" : "2013-01-18T00:00:00Z",  
    "rating" : 6.2,  
    "genres" : [  
      "Comedy",  
      "Drama"  
    ],  
    "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
    "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
    "rank" : 11,  
    "running_time_secs" : 5215,  
    "actors" : [  
      "David Matthewman",  
      "Ann Thomas",  
      "Jonathan G. Neff"  
    ]  
  }  
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy and paste the following program into a file named `MoviesLoadData.js`:

```
var AWS = require("aws-sdk");
var fs = require('fs');

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

console.log("Importing movies into DynamoDB. Please wait.");

var allMovies = JSON.parse(fs.readFileSync('moviedata.json', 'utf8'));
allMovies.forEach(function(movie) {
    var params = {
        TableName: "Movies",
        Item: {
            "year": movie.year,
            "title": movie.title,
            "info": movie.info
        }
    };

    docClient.put(params, function(err, data) {
        if (err) {
            console.error("Unable to add movie", movie.title, ". Error JSON:", JSON.stringify(err, null, 2));
        } else {
            console.log("PutItem succeeded:", movie.title);
        }
    });
});
```

2. To run the program, type the following command:

```
node MoviesLoadData.js
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 93\)](#)
- [Step 3.2: Read an Item \(p. 93\)](#)

- [Step 3.3: Update an Item \(p. 94\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 96\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 97\)](#)
- [Step 3.6: Delete an Item \(p. 98\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy and paste the following program into a file named `MoviesItemOps01.js`:

```
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

var params = {
    TableName:table,
    Item:{
        "year": year,
        "title": title,
        "info":{
            "plot": "Nothing happens at all.",
            "rating": 0
        }
    }
};

console.log("Adding a new item...");
docClient.put(params, function(err, data) {
    if (err) {
        console.error("Unable to add item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("Added item:", JSON.stringify(data, null, 2));
    }
});
```

Note

The primary key is required. This code adds an item that has a primary key (`year, title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.

2. To run the program, type the following command:

```
node MoviesItemOps01.js
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Nothing happens at all.",  
        rating: 0  
    }  
}
```

You can use the `get` method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into a file named `MoviesItemOps02.js`:

```
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient()  
  
var table = "Movies";  
  
var year = 2015;  
var title = "The Big New Movie";  
  
var params = {  
    TableName: table,  
    Key:{  
        "year": year,  
        "title": title  
    }  
};  
  
docClient.get(params, function(err, data) {  
    if (err) {  
        console.error("Unable to read item. Error JSON:", JSON.stringify(err, null, 2));  
    } else {  
        console.log("GetItem succeeded:", JSON.stringify(data, null, 2));  
    }  
});
```

2. To run the program, type the following command:

```
node MoviesItemOps02.js
```

Step 3.3: Update an Item

You can use the `update` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

To the following:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
    }
}
```

1. Copy and paste the following program into a file named `MoviesItemOps03.js`:

```
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

// Update the item, unconditionally,

var params = {
    TableName:table,
    Key:{
        "year": year,
        "title": title
    },
    UpdateExpression: "set info.rating = :r, info.plot=:p, info.actors=:a",
    ExpressionAttributeValues:{
        ":r":5.5,
        ":p":"Everything happens all at once.",
        ":a":["Larry", "Moe", "Curly"]
    },
    ReturnValues:"UPDATED_NEW"
};

console.log("Updating the item...");
docClient.update(params, function(err, data) {
    if (err) {
        console.error("Unable to update item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2));
    }
});
```

```
});
```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes ("UPDATED_NEW").

2. To run the program, type the following command:

```
node MoviesItemOps03.js
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `update` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy and paste the following program into a file named `MoviesItemOps04.js`:

```
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

// Increment an atomic counter

var params = {
    TableName:table,
    Key:{
        "year": year,
        "title": title
    },
    UpdateExpression: "set info.rating = info.rating + :val",
    ExpressionAttributeValues:{
        ":val":1
    },
    ReturnValues:"UPDATED_NEW"
};

console.log("Updating the item...");
docClient.update(params, function(err, data) {
    if (err) {
        console.error("Unable to update item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2));
    }
});
```

2. To run the program, type the following command:

```
node MoviesItemOps04.js
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors in the movie.

1. Copy and paste the following program into a file named `MoviesItemOps05.js`:

```
var AWS = require("aws-sdk");

AWS.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

// Conditional update (will fail)

var params = {
  TableName:table,
  Key:{
    "year": year,
    "title": title
  },
  UpdateExpression: "remove info.actors[0]",
  ConditionExpression: "size(info.actors) > :num",
  ExpressionAttributeValues:{
    ":num":3
  },
  ReturnValues:"UPDATED_NEW"
};

console.log("Attempting a conditional update...");
docClient.update(params, function(err, data) {
  if (err) {
    console.error("Unable to update item. Error JSON:", JSON.stringify(err, null, 2));
  } else {
    console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2));
  }
});
```

2. To run the program, type the following command:

```
node MoviesItemOps05.js
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the `ConditionExpression` looks like this:

```
ConditionExpression: "size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Run the program again. The `updateItem` operation should now succeed.

Step 3.6: Delete an Item

You can use the `delete` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into a file named `MoviesItemOps06.js`:

```
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

var params = {
    TableName:table,
    Key:{
        "year":year,
        "title":title
    },
    ConditionExpression:"info.rating <= :val",
    ExpressionAttributeValues: {
        ":val": 5.0
    }
};

console.log("Attempting a conditional delete...");
docClient.delete(params, function(err, data) {
    if (err) {
        console.error("Unable to delete item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("DeleteItem succeeded:", JSON.stringify(data, null, 2));
    }
});
```

2. To run the program, type the following command:

```
node MoviesItemOps06.js
```

The program should fail with the following message:

The conditional request failed

This is because the rating for this particular movie is greater than 5.

3. Modify the program to remove the condition from `params`.

```
var params = {  
    TableName:table,  
    Key:{  
        "title":title,  
        "year":year  
    }  
};
```

4. Run the program again. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value; the sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key); for example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to `query`, there is also a `scan` method that can retrieve all the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 99\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 100\)](#)
- [Step 4.3: Scan \(p. 101\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the `year` 1985.

1. Copy and paste the following program into a file named `MoviesQuery01.js`:

```
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient();
```

```

console.log("Querying for movies from 1985.");

var params = {
    TableName : "Movies",
    KeyConditionExpression: "#yr = :yyyy",
    ExpressionAttributeNames:{
        "#yr": "year"
    },
    ExpressionAttributeValues: {
        ":yyyy":1985
    }
};

docClient.query(params, function(err, data) {
    if (err) {
        console.error("Unable to query. Error:", JSON.stringify(err, null, 2));
    } else {
        console.log("Query succeeded.");
        data.Items.forEach(function(item) {
            console.log(" - ", item.year + ": " + item.title);
        });
    }
});

```

Note

`ExpressionAttributeNames` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you cannot use it directly in any expression, including `KeyConditionExpression`. We use the expression attribute name `#yr` to address this. `ExpressionAttributeValues` provides value substitution. We use this because you cannot use literals in any expression, including `KeyConditionExpression`. We use the expression attribute value `:yyyy` to address this.

2. To run the program, type the following command:

```
node MoviesQuery01.js
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in `year` 1992, with `title` beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into a file named `MoviesQuery02.js`:

```

var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

```

```

console.log("Querying for movies from 1992 - titles A-L, with genres and lead actor");

var params = {
    TableName : "Movies",
    ProjectionExpression:"#yr, title, info.genres, info.actors[0]",
    KeyConditionExpression: "#yr = :yyyy and title between :letter1 and :letter2",
    ExpressionAttributeNames:{ "#yr": "year"
    },
    ExpressionAttributeValues: {
        ":yyyy":1992,
        ":letter1": "A",
        ":letter2": "L"
    }
};

docClient.query(params, function(err, data) {
    if (err) {
        console.log("Unable to query. Error:", JSON.stringify(err, null, 2));
    } else {
        console.log("Query succeeded.");
        data.Items.forEach(function(item) {
            console.log(" -", item.year + ":" + item.title
            + " ... " + item.info.genres
            + " ... " + item.info.actors[0]);
        });
    }
});

```

2. To run the program, type the following command:

```
node MoviesQuery02.js
```

Step 4.3: Scan

The `scan` method reads every item in the table, and returns all the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, the filter is only applied after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The `scan` specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all of the others.

1. Copy and paste the following program into a file named `MoviesScan.js`:

```

var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

var params = {
    TableName: "Movies",
    ProjectionExpression: "#yr, title, info.rating",
    FilterExpression: "#yr between :start_yr and :end_yr",
    ExpressionAttributeNames: {
        "#yr": "year",

```

```

        },
        ExpressionAttributeValues: {
            ":start_yr": 1950,
            ":end_yr": 1959
        }
    );
}

console.log("Scanning Movies table.");
docClient.scan(params, onScan);

function onScan(err, data) {
    if (err) {
        console.error("Unable to scan the table. Error JSON:", JSON.stringify(err,
null, 2));
    } else {
        // print all the movies
        console.log("Scan succeeded.");
        data.Items.forEach(function(movie) {
            console.log(
                movie.year + ":",
                movie.title, "- rating:", movie.info.rating);
        });

        // continue scanning if we have more movies, because
        // scan can retrieve a maximum of 1MB of data
        if (typeof data.LastEvaluatedKey != "undefined") {
            console.log("Scanning for more...");
            params.ExclusiveStartKey = data.LastEvaluatedKey;
            docClient.scan(params, onScan);
        }
    }
}
}

```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
 - `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
2. To run the program, type the following command:

```
node MoviesScan.js
```

Note

You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional): Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into a file named `MoviesDeleteTable.js`:

```

var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var dynamodb = new AWS.DynamoDB();

```

```
var params = {
    TableName : "Movies"
};

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));
    }
});
```

2. To run the program, type the following command:

```
node MoviesDeleteTable.js
```

Summary

In this tutorial, you created the `Movies` table in DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the Amazon DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB service, you must change the endpoint in your application. To do this, modify the code as follows:

```
AWS.config.update({endpoint: "https://dynamodb.aws-region.amazonaws.com"});
```

For example, if you want to use the us-west-2 Region, you set the following endpoint:

```
AWS.config.update({endpoint: "https://dynamodb.us-west-2.amazonaws.com"});
```

Instead of using DynamoDB on your computer, the program now uses the DynamoDB web service endpoint in US West (Oregon).

Amazon DynamoDB is available in several Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the *AWS General Reference*. For more information about setting Regions and endpoints in your code, see [Setting the Region](#) in the *AWS SDK for JavaScript Developer Guide*.

.NET and DynamoDB

In this tutorial, you use the AWS SDK for .NET to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` using a utility program written in C#, and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

The DynamoDB module of the AWS SDK for .NET offers several programming models for different use cases. In this exercise, the C# code uses both the document model, which provides a level of abstraction that is often convenient, and also the low-level API, which handles nested attributes more effectively. For information about the document model API, see [.NET: Document Model \(p. 232\)](#). For information about the low-level API, see [Working with Tables: .NET \(p. 321\)](#).

You use the downloadable version of DynamoDB in this tutorial. For more information about how to run the same code against the DynamoDB service in the cloud, see the [Summary \(p. 131\)](#).

Prerequisites

- Use a computer that is running a recent version of Windows and a current version of Microsoft Visual Studio. If you don't have Visual Studio installed, you can download a free copy of the Community edition from the [Microsoft Visual Studio website](#).
- Download and run DynamoDB (Downloadable Version). For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Set up a security profile for DynamoDB in Visual Studio. For step-by-step instructions, see [.NET Code Samples \(p. 287\)](#).
- In Visual Studio, create a new project called `DynamoDB_intro` using the **Console Application** template in the **Installed/Templates/Visual C#/** node. This is the project you use throughout this tutorial.

Note

The following tutorial does not work with .NET core because it does not support synchronous methods. For more information, see [AWS Asynchronous APIs for .NET](#).

- Install the NuGet package for the DynamoDB module of the AWS SDK for .NET, version 3, into your `DynamoDB_intro` project. To do this, in Visual Studio, open the **NuGet Package Manager Console** from the **Tools** menu. Then type the following command at the `PM>` prompt:

```
PM> Install-Package AWSSDK.DynamoDBv2
```

Step 1: Create a Table

The document model in the AWS SDK for .NET doesn't provide for creating tables, so you have to use the low-level APIs. For more information, see [Working with Tables: .NET \(p. 321\)](#).

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;
```

```

namespace DynamoDB_intro
{
    class Program
    {
        public static void Main(string[] args)
        {
            // First, set up a DynamoDB client for DynamoDB Local
            AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
            ddbConfig.ServiceURL = "http://localhost:8000";
            AmazonDynamoDBClient client;
            try
            {
                client = new AmazonDynamoDBClient(ddbConfig);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
                PauseForDebugWindow();
                return;
            }

            // Build a 'CreateTableRequest' for the new table
            CreateTableRequest createRequest = new CreateTableRequest
            {
                TableName = "Movies",
                AttributeDefinitions = new List<AttributeDefinition>()
            {
                new AttributeDefinition
                {
                    AttributeName = "year",
                    AttributeType = "N"
                },
                new AttributeDefinition
                {
                    AttributeName = "title",
                    AttributeType = "S"
                }
            },
                KeySchema = new List<KeySchemaElement>()
            {
                new KeySchemaElement
                {
                    AttributeName = "year",
                    KeyType = "HASH"
                },
                new KeySchemaElement
                {
                    AttributeName = "title",
                    KeyType = "RANGE"
                }
            },
            };

            // Provisioned-throughput settings are required even though
            // the local test version of DynamoDB ignores them
            createRequest.ProvisionedThroughput = new ProvisionedThroughput(1, 1);

            // Using the DynamoDB client, make a synchronous CreateTable request
            CreateTableResponse createResponse;
            try
            {
                createResponse = client.CreateTable(createRequest);
            }
            catch (Exception ex)

```

```
        {
            Console.WriteLine("\n Error: failed to create the new table; " +
                ex.Message);
            PauseForDebugWindow();
            return;
        }

        // Report the status of the new table...
        Console.WriteLine("\n\n Created the \"Movies\" table successfully!\n
    Status of the new table: '{0}'", createResponse.TableDescription.TableStatus);
    }

    public static void PauseForDebugWindow()
    {
        // Keep the console open if in Debug mode...
        Console.Write("\n\n ...Press any key to continue");
        Console.ReadKey();
        Console.WriteLine();
    }
}
```

Note

- In the `AmazonDynamoDBConfig`, you set the `ServiceURL` to "http://localhost:8000" to create the table in the downloadable version of DynamoDB running on your computer.
- In the `CreateTableRequest`, you specify the table name, along with primary key attributes and their data types.
- The partition-key portion of the primary key, which determines the logical partition where DynamoDB stores an item, is identified in its `KeySchemaElement` in the `CreateTableRequest` as having `KeyType` "HASH".
- The sort-key portion of the primary key, which determines the ordering of items having the same partition-key value, is identified in its `KeySchemaElement` in the `CreateTableRequest` as having `KeyType` "RANGE".
- The `ProvisionedThroughput` field is required, even though the downloadable version of DynamoDB ignores it.

2. Compile and run the program.

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 107\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 107\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb).

The movie data is encoded as JSON. For each movie, the JSON defines a `year` name-value pair, a `title` name-value pair, and a complex `info` object, as shown in the following example:

```
{  
    "year" : 2013,
```

```
        "title" : "Turn It Down, Or Else!",
        "info" : {
            "directors" : [
                "Alice Smith",
                "Bob Jones"
            ],
            "release_date" : "2013-01-18T00:00:00Z",
            "rating" : 6.2,
            "genres" : [
                "Comedy",
                "Drama"
            ],
            "image_url" : "http://ia.media-imdb.com/images/N/
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",
            "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",
            "rank" : 11,
            "running_time_secs" : 5215,
            "actors" : [
                "David Matthewman",
                "Ann Thomas",
                "Jonathan G. Neff"
            ]
        }
    }
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into the `bin/Debug` folder of your `DynamoDB_intro` Visual Studio project.

Step 2.2: Load the Sample Data into the Movies Table

Build a program that loads movie data into the table you created in Step 1.

1. This program uses the open source `Newtonsoft.Json` library for deserializing JSON data, licensed under the MIT License (MIT) (see <https://github.com/JamesNK/Newtonsoft.Json/blob/master/LICENSE.md>).

To load the `Json.NET` library into your project, in Visual Studio, open the **NuGet Package Manager Console** from the **Tools** menu. Then type the following command at the `PM>` prompt:

```
PM> Install-Package Newtonsoft.Json
```

2. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

using Newtonsoft;
using Newtonsoft.Json;
```

```
using Newtonsoft.Json.Linq;

namespace DynamoDB_intro
{
    class Program
    {
        public static Table GetTableObject(string tableName)
        {
            // First, set up a DynamoDB client for DynamoDB Local
            AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
            ddbConfig.ServiceURL = "http://localhost:8000";
            AmazonDynamoDBClient client;
            try
            {
                client = new AmazonDynamoDBClient(ddbConfig);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
                return (null);
            }

            // Now, create a Table object for the specified table
            Table table;
            try
            {
                table = Table.LoadTable(client, tableName);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
                return (null);
            }
            return (table);
        }

        public static void Main(string[] args)
        {
            // First, read in the JSON data from the moviedate.json file
            StreamReader sr = null;
            JsonTextReader jtr = null;
            JArray movieArray = null;
            try
            {
                sr = new StreamReader("moviedata.json");
                jtr = new JsonTextReader(sr);
                movieArray = (JArray)JToken.ReadFrom(jtr);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: could not read from the 'moviedata.json' file, because: " + ex.Message);
                PauseForDebugWindow();
                return;
            }
            finally
            {
                if (jtr != null)
                    jtr.Close();
                if (sr != null)
                    sr.Close();
            }
        }

        // Get a Table object for the table that you created in Step 1
    }
}
```

```

Table table = GetTableObject("Movies");
if (table == null)
{
    PauseForDebugWindow();
    return;
}

// Load the movie data into the table (this could take some time)
Console.WriteLine("\n    Now writing {0:#,##0} movie records from moviedata.json
(might take 15 minutes)...\\n    ...completed: ", movieArray.Count);
for (int i = 0, j = 99; i < movieArray.Count; i++)
{
    try
    {
        string itemJson = movieArray[i].ToString();
        Document doc = Document.FromJson(itemJson);
        table.PutItem(doc);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\nError: Could not write the movie record
#{0:#,##0}, because {1}", i, ex.Message);
        PauseForDebugWindow();
        return;
    }
    if (i >= j)
    {
        j++;
        Console.Write("{0,5:#,##0}, ", j);
        if (j % 1000 == 0)
            Console.WriteLine("\\n");
        j += 99;
    }
}
Console.WriteLine("\n    Finished writing all movie records to DynamoDB!");
PauseForDebugWindow();
}

public static void PauseForDebugWindow()
{
    // Keep the console open if in Debug mode...
    Console.Write("\n\\n ...Press any key to continue");
    Console.ReadKey();
    Console.WriteLine();
}
}

```

3. Now compile the project, leaving it in Debug mode, and run it.

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- Step 3.1: Create a New Item (p. 110)
 - Step 3.2: Read an Item (p. 111)
 - Step 3.3: Update an Item (p. 113)
 - Step 3.4: Increment an Atomic Counter (p. 116)

- [Step 3.5: Update an Item \(Conditionally\) \(p. 118\)](#)
- [Step 3.6: Delete an Item \(p. 120\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject("Movies");
            if (table == null)
            {
                PauseForDebugWindow();
                return;
            }

            // Create a Document representing the movie item to be written to the
            table
            Document document = new Document();
            document["year"] = 2015;
            document["title"] = "The Big New Movie";
            document["info"] = Document.FromJson("{\"plot\" : \"Nothing happens at all.\",
\", \"rating\" : 0}");

            // Use Table.PutItem to write the document item to the table
            try
            {
                table.PutItem(document);
                Console.WriteLine("\nPutItem succeeded.\n");
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: Table.PutItem failed because: " +
ex.Message);
                PauseForDebugWindow();
                return;
            }
        }

        public static Table GetTableObject(string tableName)
        {
            // First, set up a DynamoDB client for DynamoDB Local
            AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
            ddbConfig.ServiceURL = "http://localhost:8000";
            AmazonDynamoDBClient client;
            try
```

```
        {
            client = new AmazonDynamoDBClient(ddbConfig);
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
            return (null);
        }

        // Now, create a Table object for the specified table
        Table table = null;
        try
        {
            table = Table.LoadTable(client, tableName);
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
            return (null);
        }
        return (table);
    }

    public static void PauseForDebugWindow()
    {
        // Keep the console open if in Debug mode...
        Console.Write("\n\n ...Press any key to continue");
        Console.ReadKey();
        Console.WriteLine();
    }
}
```

Note

The primary key is required. In this table, the primary key is a composite of both a partition-key attribute (`year`) and a sort-key attribute (`title`).

This code writes an item to the table that has the two primary key attributes (year + title) and a complex `info` attribute that stores more information about the movie.

- ## 2. Compile and run the program.

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{  
  year: 2015,  
  title: "The Big New Movie",  
  info: {  
    plot: "Nothing happens at all.",  
    rating: 0  
  }  
}
```

You can use the `GetItem` method to read the item from the `Movies` table. You must specify the primary key values so that you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject("Movies");
            if (table == null)
            {
                PauseForDebugWindow();
                return;
            }

            try
            {
                Document document = table.GetItem(2015, "The Big New Movie");
                if (document != null)
                    Console.WriteLine("\nGetItem succeeded: \n" +
document.ToJsonPretty());
                else
                    Console.WriteLine("\nGetItem succeeded, but the item was not
found");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }

        public static Table GetTableObject(string tableName)
        {
            // First, set up a DynamoDB client for DynamoDB Local
            AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
            ddbConfig.ServiceURL = "http://localhost:8000";
            AmazonDynamoDBClient client;
            try
            {
                client = new AmazonDynamoDBClient(ddbConfig);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
                return (null);
            }

            // Now, create a Table object for the specified table
            Table table = null;
            try
            {
                table = Table.LoadTable(client, tableName);
            }
            catch (Exception ex)
```

```
        {
            Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
            return (null);
        }
        return (table);
    }

    public static void PauseForDebugWindow()
    {
        // Keep the console open if in Debug mode...
        Console.Write("\n\n ...Press any key to continue");
        Console.ReadKey();
        Console.WriteLine();
    }
}
```

- ## 2. Compile and run the program.

Step 3.3: Update an Item

You can use the `UpdateItem` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
 - Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Nothing happens at all.",  
        rating: 0  
    }  
}
```

To the following:

```
{  
  year: 2015,  
  title: "The Big New Movie",  
  info: {  
    plot: "Everything happens all at once.",  
    rating: 5.5,  
    actors: ["Larry", "Moe", "Curly"]  
  }  
}
```

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

using System.Threading.Tasks;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local database
            AmazonDynamoDBClient client = GetLocalClient();
            if (client == null)
            {
                PauseForDebugWindow();
                return;
            }

            // Create an UpdateItemRequest to modify two existing nested attributes
            // and add a new one
            UpdateItemRequest updateRequest = new UpdateItemRequest()
            {
                TableName = "Movies",
                Key = new Dictionary<string, AttributeValue>
                {
                    { "year", new AttributeValue {
                        N = "2015"
                    } },
                    { "title", new AttributeValue {
                        S = "The Big New Movie"
                    } }
                },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>
                {
                    { ":r", new AttributeValue {
                        N = "5.5"
                    } },
                    { ":p", new AttributeValue {
                        S = "Everything happens all at once!"
                    } },
                    { ":a", new AttributeValue {
                        SS = { "Larry", "Moe", "Curly" }
                    } }
                },
                UpdateExpression = "SET info.rating = :r, info.plot = :p, info.actors
= :a",
                ReturnValue = "UPDATED_NEW"
            };

            // Use AmazonDynamoDBClient.UpdateItem to update the specified attributes
            UpdateItemResponse uir = null;
            try
            {
                uir = client.UpdateItem(updateRequest);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\nError: UpdateItem failed, because: " +
ex.Message);
                if (uir != null)
                    Console.WriteLine("    Status code was " +
uir.HttpStatusCode.ToString());
                PauseForDebugWindow();
            }
        }
    }
}

```

```

        return;
    }

    // Get the item from the table and display it to validate that the update
succeeded
    DisplayMovieItem(client, "2015", "The Big New Movie");
}

public static AmazonDynamoDBClient GetLocalClient()
{
    // First, set up a DynamoDB client for DynamoDB Local
    AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
    ddbConfig.ServiceURL = "http://localhost:8000";
    AmazonDynamoDBClient client;
    try
    {
        client = new AmazonDynamoDBClient(ddbConfig);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
        return (null);
    }
    return (client);
}

public static void DisplayMovieItem(AmazonDynamoDBClient client, string year,
string title)
{
    // Create Primitives for the HASH and RANGE portions of the primary key
    Primitive hash = new Primitive(year, true);
    Primitive range = new Primitive(title, false);

    Table table = null;
    try
    {
        table = Table.LoadTable(client, "Movies");
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
        return;
    }
    Document document = table.GetItem(hash, range);
    Console.WriteLine("\n The movie record looks like this: \n" +
document.ToString());
}

public static void PauseForDebugWindow()
{
    // Keep the console open if in Debug mode...
    Console.Write("\n\n ...Press any key to continue");
    Console.ReadKey();
    Console.WriteLine();
}
}
```

Note

Because the document model in the AWS SDK for .NET doesn't support updating nested attributes, you must use the `AmazonDynamoDBClient.UpdateItem` API instead of `Table.UpdateItem` to update attributes under the top-level `info` attribute.

To do this, create an `UpdateItemRequest` that specifies the item you want to update and the new values you want to set.

- The `UpdateExpression` is what defines all the updates to be performed on the specified item.
- By setting the `ReturnValues` field to `"UPDATED_NEW"`, you are requesting that DynamoDB return only the updated attributes in the response.

2. Compile and run the program.

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `UpdateItem` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program increments the `rating` for a movie. Each time you run it, the program increments this attribute by one. Again, it is the `UpdateExpression` that determines what happens:

```
UpdateExpression = "SET info.rating = info.rating + :inc",
```

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local database
            AmazonDynamoDBClient client = GetLocalClient();
            if (client == null)
            {
                PauseForDebugWindow();
                return;
            }

            // Create an UpdateItemRequest to modify two existing nested attributes
            // and add a new one
            UpdateItemRequest updateRequest = new UpdateItemRequest()
            {
                TableName = "Movies",
                Key = new Dictionary<string, AttributeValue>
                {
                    { "year", new AttributeValue {
                        N = "2015"
                    } },
                    { "title", new AttributeValue {
                        S = "The Big New Movie"
                    } }
                },
                UpdateExpression = "SET info.rating = info.rating + :inc",
                ConditionExpression = "info.year < 2015",
                ExpressionAttributeNames = new Dictionary<string, string>(),
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            };
        }
    }
}
```

```

        },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>
    {
        { ":inc", new AttributeValue {
            N = "1"
        } }
    },
    UpdateExpression = "SET info.rating = info.rating + :inc",
    ReturnValue = "UPDATED_NEW"
};

// Use AmazonDynamoDBClient.UpdateItem to update the specified attributes
UpdateItemResponse uir = null;
try
{
    uir = client.UpdateItem(updateRequest);
}
catch (Exception ex)
{
    Console.WriteLine("\nError: UpdateItem failed, because " +
ex.Message);
    if (uir != null)
        Console.WriteLine("    Status code was: " +
uir.HttpStatusCode.ToString());
    PauseForDebugWindow();
    return;
}

// Get the item from the table and display it to validate that the update
succeeded
DisplayMovieItem(client, "2015", "The Big New Movie");
}

public static AmazonDynamoDBClient GetLocalClient()
{
    // First, set up a DynamoDB client for DynamoDB Local
    AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
    ddbConfig.ServiceURL = "http://localhost:8000";
    AmazonDynamoDBClient client;
    try
    {
        client = new AmazonDynamoDBClient(ddbConfig);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
        return (null);
    }
    return (client);
}

public static void DisplayMovieItem(AmazonDynamoDBClient client, string year,
string title)
{
    // Create Primitives for the HASH and RANGE portions of the primary key
    Primitive hash = new Primitive(year, true);
    Primitive range = new Primitive(title, false);

    Table table = null;
    try
    {
        table = Table.LoadTable(client, "Movies");
    }
    catch (Exception ex)
    {
}

```

```
        Console.WriteLine("\n Error: failed to load the 'Movies' table; " +  
ex.Message);  
        return;  
    }  
    Document document = table.GetItem(hash, range);  
    Console.WriteLine("\n The movie record looks like this: \n" +  
document.ToString());  
}  
  
public static void PauseForDebugWindow()  
{  
    // Keep the console open if in Debug mode...  
    Console.Write("\n\n ...Press any key to continue");  
    Console.ReadKey();  
    Console.WriteLine();  
}  
}  
}
```

- ## 2. Compile and run the program.

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is only updated if there are more than three actors in the movie.

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local database
            AmazonDynamoDBClient client = GetLocalClient();
            if (client == null)
            {
                PauseForDebugWindow();
                return;
            }

            // Create an UpdateItemRequest to modify two existing nested attributes
            // and add a new one
            UpdateItemRequest updateRequest = new UpdateItemRequest()
            {
                TableName = "Movies",
                Key = new Dictionary<string, AttributeValue>
                {
                    { "year", new AttributeValue {
```

```

        N = "2015"
    } },
    { "title", new AttributeValue {
        S = "The Big New Movie"
    } }
},
ExpressionAttributeValues = new Dictionary<string, AttributeValue>
{
    { ":n", new AttributeValue {
        N = "3"
    } }
},
ConditionExpression = "size(info.actors) > :n",
UpdateExpression = "REMOVE info.actors",
ReturnValues = "UPDATED_NEW"
};

// Use AmazonDynamoDBClient.UpdateItem to update the specified attributes
UpdateItemResponse uir = null;
try
{
    uir = client.UpdateItem(updateRequest);
}
catch (Exception ex)
{
    Console.WriteLine("\nError: UpdateItem failed, because:\n" + ex.Message);
    if (uir != null)
        Console.WriteLine("    Status code was " +
uir.HttpStatusCode.ToString());
    PauseForDebugWindow();
    return;
}
if (uir.HttpStatusCode != System.Net HttpStatusCode.OK)
{
    PauseForDebugWindow();
    return;
}

// Get the item from the table and display it to validate that the update
succeeded
DisplayMovieItem(client, "2015", "The Big New Movie");
}

public static AmazonDynamoDBClient GetLocalClient()
{
    // First, set up a DynamoDB client for DynamoDB Local
    AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
    ddbConfig.ServiceURL = "http://localhost:8000";
    AmazonDynamoDBClient client;
    try
    {
        client = new AmazonDynamoDBClient(ddbConfig);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
        return (null);
    }
    return (client);
}

public static void DisplayMovieItem(AmazonDynamoDBClient client, string year,
string title)
{

```

```
// Create Primitives for the HASH and RANGE portions of the primary key
Primitive hash = new Primitive(year, true);
Primitive range = new Primitive(title, false);

Table table = null;
try
{
    table = Table.LoadTable(client, "Movies");
}
catch (Exception ex)
{
    Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
    return;
}
Document document = table.GetItem(hash, range);
Console.WriteLine("\n The movie record looks like this: \n" +
document.ToStringPretty());
}

public static void PauseForDebugWindow()
{
    // Keep the console open if in Debug mode...
    Console.Write("\n\n ...Press any key to continue");
    Console.ReadKey();
    Console.WriteLine();
}
}
```

- ## 2. Compile and run the program.

The program should fail with the following message:

The conditional request failed

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the number of actors that the ConditionExpression uses is 2 instead of 3:

```
{ ":n", new AttributeValue { N = "2" } }
```

The condition now specifies that the number of actors must be greater than 2.

4. When you compile and run the program now, the `UpdateItem` operation should succeed.

Step 3.6: Delete an Item

You can use the `Table.DeleteItem` operation to delete an item by specifying its primary key. You can optionally provide a condition in the `DeleteItemOperationConfig` parameter, to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

using System.Threading.Tasks;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject("Movies");
            if (table == null)
                return;

            // Create the condition
            DeleteItemOperationConfig opConfig = new DeleteItemOperationConfig();
            opConfig.ConditionalExpression = new Expression();
            opConfig.ConditionalExpression.ExpressionAttributeValues[":val"] = "5.0";
            opConfig.ConditionalExpression.ExpressionStatement = "info.rating
<= :val";

            // Delete this item
            try
            {
                table.DeleteItem(2015, "The Big New Movie", opConfig);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: Could not delete the movie item with
year={0}, title=\"{1}\"\n    Reason: {2}.",
                                2015, "The Big New Movie", ex.Message);
            }

            // Try to retrieve it, to see if it has been deleted
            Document document = table.GetItem(2015, "The Big New Movie");
            if (document == null)
                Console.WriteLine("\n The movie item with year={0}, title=\"{1}\" has
been deleted.",
                                2015, "The Big New Movie");
            else
                Console.WriteLine("\nRead back the item: \n" +
document.ToJsonPretty());

            // Keep the console open if in Debug mode...
            Console.Write("\n\n ...Press any key to continue");
            Console.ReadKey();
            Console.WriteLine();
        }

        public static Table GetTableObject(string tableName)
        {
            // First, set up a DynamoDB client for DynamoDB Local
            AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
            ddbConfig.ServiceURL = "http://localhost:8000";
            AmazonDynamoDBClient client;
            try
            {
                client = new AmazonDynamoDBClient(ddbConfig);
            }
            catch (Exception ex)
            {

```

```
        Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
    ex.Message);
        return (null);
    }

    // Now, create a Table object for the specified table
    Table table = null;
    try
    {
        table = Table.LoadTable(client, tableName);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
    ex.Message);
        return (null);
    }
    return (table);
}
}
```

2. Compile and run the program.

The program should fail with the following message:

```
The conditional request failed
```

This is because the rating for this particular move is greater than 5.

3. Modify the program to remove the `DeleteItemOperationConfig` named `opConfig` from the call to `table.DeleteItem`:

```
try { table.DeleteItem( 2015, "The Big New Movie" ); }
```

4. Compile and run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `Query` method to retrieve data from a table. You must specify a partition key value; the sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year` partition-key attribute. You can add the `title` sort-key attribute to retrieve a subset of movies based on some condition (on the sort-key attribute), such as finding movies released in 2014 that have a title starting with the letter "A".

In addition to `Query`, there is also a `Scan` method that can retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query \(p. 123\)](#)
- [Step 4.2: Scan \(p. 127\)](#)

Step 4.1: Query

The C# code included in this step performs the following queries:

- Retrieves all movies release in year 1985.
- Retrieves all movies released in year 1992, with title beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static string commaSep = ", ";
        static string movieFormatString = "{0} {1}, lead actor: {2}, genres: {3}";

        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local DynamoDB database
            AmazonDynamoDBClient client = GetLocalClient();

            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject(client, "Movies");
            if (table == null)
            {
                PauseForDebugWindow();
                return;
            }

            /-----
            * 4.1.1: Call Table.Query to initiate a query for all movies with
            *         year == 1985, using an empty filter expression.
            -----
        }

        Search search;
        try
        {
            search = table.Query(1985, new Expression());
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n Error: 1985 query failed because: " +
                ex.Message);
            PauseForDebugWindow();
            return;
        }

        // Display the titles of the movies returned by this query
        List<Document> docList = new List<Document>();
        Console.WriteLine("\n All movies released in 1985: " +
```

```

        "\n-----");
do
{
    try { docList = search.GetNextSet(); }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: Search.GetNextStep failed because: " +
ex.Message);
        break;
    }
    foreach (var doc in docList)
        Console.WriteLine("    " + doc["title"]);
} while (!search.IsDone);

/*
* 4.1.2a: Call Table.Query to initiate a query for all movies where
*          year equals 1992 AND title is between "B" and "Hzzz",
*          returning the lead actor and genres of each.
*/
Primitive y_1992 = new Primitive("1992", true);
QueryOperationConfig config = new QueryOperationConfig();
config.Filter = new QueryFilter();
config.Filter.AddCondition("year", QueryOperator.Equal, new DynamoDBEntry[]
{ 1992 });
config.Filter.AddCondition("title", QueryOperator.Between, new
DynamoDBEntry[] { "B", "Hzz" });
config.AttributesToGet = new List<string> { "title", "info" };
config.Select = SelectValues.SpecificAttributes;

try
{
    search = table.Query(config);
}
catch (Exception ex)
{
    Console.WriteLine("\n Error: 1992 query failed because: " +
ex.Message);
    PauseForDebugWindow();
    return;
}

// Display the movie information returned by this query
Console.WriteLine("\n\n Movies released in 1992 with titles between \"B\""
and "\"Hzz\" (Document Model):" +
"\n-----");
docList = new List<Document>();
Document infoDoc;
do
{
    try
    {
        docList = search.GetNextSet();
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: Search.GetNextStep failed because: " +
ex.Message);
        break;
    }
    foreach (var doc in docList)
    {
        infoDoc = doc["info"].AsDocument();
        Console.WriteLine(movieFormatString,

```

```

                doc["title"],
                infoDoc["actors"].AsArrayOfString()[0],
                string.Join(commaSep,
infoDoc["genres"].AsArrayOfString()));
            }
        } while (!search.IsDone);

        //---------------------------------------------------------------------
        * 4.1.2b: Call AmazonDynamoDBClient.Query to initiate a query for all
        * movies where year equals 1992 AND title is between M and Tzz,
        * returning the genres and the lead actor of each.
        *-----
    */
    QueryRequest qRequest = new QueryRequest
    {
        TableName = "Movies",
        ExpressionAttributeNames = new Dictionary<string, string>
        {
            { "#yr", "year" }
        },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>
        {
            { ":y_1992", new AttributeValue {
                N = "1992"
            } },
            { ":M", new AttributeValue {
                S = "M"
            } },
            { ":Tzz", new AttributeValue {
                S = "Tzz"
            } }
        },
        KeyConditionExpression = "#yr = :y_1992 and title between :M
and :Tzz",
        ProjectionExpression = "title, info.actors[0], info.genres"
    };

    QueryResponse qResponse;
    try
    {
        qResponse = client.Query(qRequest);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: Low-level query failed, because: " +
ex.Message);
        PauseForDebugWindow();
        return;
    }

    // Display the movie information returned by this query
    Console.WriteLine("\n\n Movies released in 1992 with titles between \"M\""
and "\"Tzz\" (low-level):" +
"\n-----");
    foreach (Dictionary<string, AttributeValue> item in qResponse.Items)
    {
        Dictionary<string, AttributeValue> info = item["info"].M;
        Console.WriteLine(movieFormatString,
                        item["title"].S,
                        info["actors"].L[0].S,
                        GetDdbListAsString(info["genres"].L));
    }
}

```

```

public static string GetDdbListAsString(List<AttributeValue> strList)
{
    StringBuilder sb = new StringBuilder();
    string str = null;
    AttributeValue av;
    for (int i = 0; i < strList.Count; i++)
    {
        av = strList[i];
        if (av.S != null)
            str = av.S;
        else if (av.N != null)
            str = av.N;
        else if (av.SS != null)
            str = string.Join(commaSep, av.SS.ToArray());
        else if (av.NS != null)
            str = string.Join(commaSep, av.NS.ToArray());
        if (str != null)
        {
            if (i > 0)
                sb.Append(commaSep);
            sb.Append(str);
        }
    }
    return (sb.ToString());
}

public static AmazonDynamoDBClient GetLocalClient()
{
    // First, set up a DynamoDB client for DynamoDB Local
    AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
    ddbConfig.ServiceURL = "http://localhost:8000";
    AmazonDynamoDBClient client;
    try
    {
        client = new AmazonDynamoDBClient(ddbConfig);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
        return (null);
    }
    return (client);
}

public static Table GetTableObject(AmazonDynamoDBClient client, string
tableName)
{
    Table table = null;
    try
    {
        table = Table.LoadTable(client, tableName);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
        return (null);
    }
    return (table);
}

public static void PauseForDebugWindow()
{
    // Keep the console open if in Debug mode...
}

```

```
        Console.WriteLine("\n\n ...Press any key to continue");
        Console.ReadKey();
        Console.WriteLine();
    }
}
```

Note

- In the first query, for all movies released in 1985, an empty expression indicates that filtering on the sort-key part of the primary key is not wanted.
 - In the second query, which uses the AWS SDK for .NET document model to query for all movies released in 1992 with titles starting with the letters A through L, you can query only for top-level attributes. So you must retrieve the entire `info` attribute. The display code then accesses the nested attributes you're interested in.
 - In the third query, you use the low-level AWS SDK for .NET API, which gives more control over what is returned. Here, you can retrieve only those nested attributes within the `info` attribute that you're interested in, namely `info.genres` and `info.actors[0]`.
2. Compile and run the program.

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can also optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Scan

The `Scan` method reads every item in the table and returns all the data in the table. You can provide an optional `filter_expression` so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items) and discard all the others.

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Amazon;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2.DocumentModel;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local DynamoDB database
            AmazonDynamoDBClient client = GetLocalClient();
```

```

// Get a Table object for the table that you created in Step 1
Table table = GetTableObject(client, "Movies");
if (table == null)
{
    PauseForDebugWindow();
    return;
}

/*
 * 4.2a: Call Table.Scan to return the movies released in the 1950's,
 *        displaying title, year, lead actor and lead director.
*/
ScanFilter filter = new ScanFilter();
filter.AddCondition("year", ScanOperator.Between, new DynamoDBEntry[]
{ 1950, 1959 });
ScanOperationConfig config = new ScanOperationConfig
{
    AttributesToGet = new List<string> { "year", "title", "info" },
    Filter = filter
};
Search search = table.Scan(filter);

// Display the movie information returned by this query
Console.WriteLine("\n\n Movies released in the 1950's (Document Model):" +
    "\n-----");
List<Document> docList = new List<Document>();
Document infoDoc;
string movieFormatString = "      \"{0}\" ({1})-- lead actor: {2}, lead
director: {3}";
do
{
    try
    {
        docList = search.GetNextSet();
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: Search.GetNextStep failed because: " +
ex.Message);
        break;
    }
    foreach (var doc in docList)
    {
        infoDoc = doc["info"].AsDocument();
        Console.WriteLine(movieFormatString,
            doc["title"],
            doc["year"],
            infoDoc["actors"].AsArrayList()[0],
            infoDoc["directors"].AsArrayList()[0]);
    }
} while (!search.IsDone);

/*
 * 4.2b: Call AmazonDynamoDBClient.Scan to return all movies released
 *        in the 1960's, only downloading the title, year, lead
 *        actor and lead director attributes.
*/
ScanRequest sRequest = new ScanRequest
{
    TableName = "Movies",
    ExpressionAttributeNames = new Dictionary<string, string>
{

```

```

        { "#yr", "year" }
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>
{
    { ":y_a", new AttributeValue {
        N = "1960"
    } },
    { ":y_z", new AttributeValue {
        N = "1969"
    } },
},
FilterExpression = "#yr between :y_a and :y_z",
ProjectionExpression = "#yr, title, info.actors[0], info.directors[0]"
};

ScanResponse sResponse;
try
{
    sResponse = client.Scan(sRequest);
}
catch (Exception ex)
{
    Console.WriteLine("\n Error: Low-level scan failed, because: " +
ex.Message);
    PauseForDebugWindow();
    return;
}

// Display the movie information returned by this scan
Console.WriteLine("\n\n Movies released in the 1960's (low-level):" +
"\n-----");
foreach (Dictionary<string, AttributeValue> item in sResponse.Items)
{
    Dictionary<string, AttributeValue> info = item["info"].M;
    Console.WriteLine(movieFormatString,
        item["title"].S,
        item["year"].N,
        info["actors"].L[0].S,
        info["directors"].L[0].S);
}
}

public static AmazonDynamoDBClient GetLocalClient()
{
    // First, set up a DynamoDB client for DynamoDB Local
    AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
    ddbConfig.ServiceURL = "http://localhost:8000";
    AmazonDynamoDBClient client;
    try
    {
        client = new AmazonDynamoDBClient(ddbConfig);
    }
    catch (Exception ex)
    {
        Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
        return (null);
    }
    return (client);
}

public static Table GetTableObject(AmazonDynamoDBClient client, string
tableName)
{
    Table table = null;
}

```

```

        try
        {
            table = Table.LoadTable(client, tableName);
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
ex.Message);
            return (null);
        }
        return (table);
    }

    public static void PauseForDebugWindow()
    {
        // Keep the console open if in Debug mode...
        Console.Write("\n\n ...Press any key to continue");
        Console.ReadKey();
        Console.WriteLine();
    }
}
}
}

```

In the code, note the following:

- The first scan uses the AWS SDK for .NET document model to scan the `Movies` table and return movies released in the 1950s. Because the document model doesn't support nested attributes in the `AttributesToGet` field, you must download the entire `info` attribute to have access to the lead actor and director.
 - The second scan uses the AWS SDK for .NET low-level API to scan the `Movies` table and return movies released in the 1960s. In this case, you can download only those attribute values in `info` that you're interested in, namely `info.actors[0]` and `info.directors[0]`.
2. Compile and run the program.

Note

You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into the `Program.cs` file, replacing its current contents:

```

using System;
using System.Text;

using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace DynamoDB_intro
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local DynamoDB database
            AmazonDynamoDBClient client = GetLocalClient();

            try

```

```
        {
            client.DeleteTable("Movies");
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n Error: the \'Movies\' table could not be deleted!
\n Reason: " + ex.Message);
            Console.WriteLine("\n\n ...Press any key to continue");
            Console.ReadKey();
            Console.WriteLine();
            return;
        }
        Console.WriteLine("\n Deleted the \'Movies\' table successfully!");
    }

    public static AmazonDynamoDBClient GetLocalClient()
    {
        // First, set up a DynamoDB client for DynamoDB Local
        AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
        ddbConfig.ServiceURL = "http://localhost:8000";
        AmazonDynamoDBClient client;
        try
        {
            client = new AmazonDynamoDBClient(ddbConfig);
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n Error: failed to create a DynamoDB client; " +
ex.Message);
            return (null);
        }
        return (client);
    }
}
```

- ## 2. Compile and run the program.

Summary

In this tutorial, you created the `Movies` table in DynamoDB on your computer and performed basic operations. DynamoDB (Downloadable Version) is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the Amazon DynamoDB service.

Modifying the Code to Use the DynamoDB Service

To use the Amazon DynamoDB service, you must change the endpoint in your application.

1. To do this, first remove the following line:

```
ddbConfig.ServiceURL = "http://localhost:8000";
```

2. Add a new line that specifies the AWS Region you want to access:

```
ddbConfig.RegionEndpoint = RegionEndpoint.REGION;
```

For example, if you want to access the us-west-2 region, you would do this:

```
ddbConfig.RegionEndpoint = RegionEndpoint.USWest2;
```

Instead of using the downloadable version of DynamoDB, the program now uses the DynamoDB service endpoint in US West (Oregon).

Amazon DynamoDB is available in several Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the *AWS General Reference*. For more information about setting Regions and endpoints in your code, see [AWS Region Selection](#) in the *AWS SDK for .NET Developer Guide*.

PHP and DynamoDB

In this tutorial, you use the AWS SDK for PHP to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB service, see the [Summary \(p. 149\)](#).

As you work through this tutorial, you can refer to the [AWS SDK for PHP Developer Guide](#). The [Amazon DynamoDB section](#) in the *AWS SDK for PHP API Reference* describes the parameters and results for DynamoDB operations.

Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Set up the AWS SDK for PHP:
 - Go to <http://php.net> and install PHP.
 - Go to <https://aws.amazon.com/sdk-for-php> and install the SDK for PHP.

For more information, see [Getting Started](#) in the *AWS SDK for PHP Getting Started Guide*.

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following two attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy and paste the following program into a file named `MoviesCreateTable.php`:

```
<?php
require 'vendor/autoload.php';
```

```

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();

$params = [
    'TableName' => 'Movies',
    '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
    ]
];
try {
    $result = $dynamodb->createTable($params);
    echo 'Created table. Status: ' .
        $result['TableDescription']['TableStatus'] . "\n";
} catch (DynamoDbException $e) {
    echo "Unable to create table:\n";
    echo $e->getMessage() . "\n";
}
?>

```

Note

- You set the endpoint to indicate that you are creating the table in DynamoDB on your computer.
- In the `createTable` call, you specify table name, primary key attributes, and its data types.
- The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)

2. To run the program, type the following command:

```
php MoviesCreateTable.php
```

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 135\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 135\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{  
  "year" : 2013,  
  "title" : "Turn It Down, Or Else!",  
  "info" : {  
    "directors" : [  
      "Alice Smith",  
      "Bob Jones"  
    ],  
    "release_date" : "2013-01-18T00:00:00Z",  
    "rating" : 6.2,  
    "genres" : [  
      "Comedy",  
      "Drama"  
    ],  
    "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
}
```

```
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",
        "rank" : 11,
        "running_time_secs" : 5215,
        "actors" : [
            "David Matthewman",
            "Ann Thomas",
            "Jonathan G. Neff"
        ]
    }
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy and paste the following program into a file named `MoviesLoadData.php`:

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region' => 'us-west-2',
    'version' => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$movies = json_decode(file_get_contents('moviedata.json'), true);

foreach ($movies as $movie) {

    $year = $movie['year'];
    $title = $movie['title'];
    $info = $movie['info'];

    $json = json_encode([
        'year' => $year,
        'title' => $title,
        'info' => $info
    ]);

    $params = [
        'TableName' => $tableName,
        'Item' => $marshaler->marshalJson($json)
    ];
}
```

```
try {
    $result = $dynamodb->putItem($params);
    echo "Added movie: " . $movie['year'] . " " . $movie['title'] . "\n";
} catch (DynamoDbException $e) {
    echo "Unable to add movie:\n";
    echo $e->getMessage() . "\n";
    break;
}
?>
```

Note

The [DynamoDB Marshaler class](#) has methods for converting JSON documents and PHP arrays to the DynamoDB format. In this program, `$marshaler->marshalJson($json)` takes a JSON document and converts it into a DynamoDB item.

2. To run the program, type the following command:

```
php MoviesLoadData.php
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 136\)](#)
- [Step 3.2: Read an Item \(p. 137\)](#)
- [Step 3.3: Update an Item \(p. 138\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 140\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 141\)](#)
- [Step 3.6: Delete an Item \(p. 142\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy and paste the following program into a file named `MoviesItemOps01.php`:

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'  => 'http://localhost:8000',
    'region'    => 'us-west-2',
    'version'   => 'latest'
]);
```

```

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$item = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . ''",
    "info": {
        "plot": "Nothing happens at all.",
        "rating": 0
    }
});
');

$params = [
    'TableName' => 'Movies',
    'Item' => $item
];

try {
    $result = $dynamodb->putItem($params);
    echo "Added item: $year - $title\n";

} catch (DynamoDbException $e) {
    echo "Unable to add item:\n";
    echo $e->getMessage() . "\n";
}
?>
```

Note

The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores a map that provides more information about the movie.

2. To run the program, type the following command:

```
php MoviesItemOps01.php
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

You can use the `getItem` method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into a file named `MoviesItemOps02.php`:

```

<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region' => 'us-west-2',
    'version' => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . '"
}
');

$params = [
    'TableName' => $tableName,
    'Key' => $key
];

try {
    $result = $dynamodb->getItem($params);
    print_r($result["Item"]);
} catch (DynamoDbException $e) {
    echo "Unable to get item:\n";
    echo $e->getMessage() . "\n";
}

?>

```

2. To run the program, type the following command:

```
php MoviesItemOps02.php
```

Step 3.3: Update an Item

You can use the `updateItem` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Nothing happens at all.",  
        rating: 0  
    }  
}
```

To the following:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Everything happens all at once.",  
        rating: 5.5,  
        actors: ["Larry", "Moe", "Curly"]  
    }  
}
```

1. Copy and paste the following program into a file named `MoviesItemOps03.php`:

```
<?php  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
    'version' => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
$marshaler = new Marshaler();  
  
$tableName = 'Movies';  
  
$year = 2015;  
$title = 'The Big New Movie';  
  
$key = $marshaler->marshalJson(''  
{  
    "year": ' . $year . ',  
    "title": "' . $title . '"  
}  
' );  
  
$eav = $marshaler->marshalJson(''  
{  
    ":r": 5.5 ,  
    ":p": "Everything happens all at once.",  
    ":a": [ "Larry", "Moe", "Curly" ]  
}  
' );  
  
$params = [  
    'TableName' => $tableName,
```

```

'Key' => $key,
'UpdateExpression' =>
    'set info.rating = :r, info.plot=:p, info.actors=:a',
'ExpressionAttributeValues'=> $eav,
'ReturnValues' => 'UPDATED_NEW'
];

try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item.\n";
    print_r($result['Attributes']);
}

} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}

?>

```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. To run the program, type the following command:

```
php MoviesItemOps03.php
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `updateItem` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy and paste the following program into a file named `MoviesItemOps04.php`:

```

<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint'    => 'http://localhost:8000',
    'region'      => 'us-west-2',
    'version'     => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaller = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

```

```

$key = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . '"
}
');

$eav = $marshaler->marshalJson(
{
    ":val": 1
}
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' => 'set info.rating = info.rating + :val',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];
try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item. ReturnValues are:\n";
    print_r($result['Attributes']);
} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}
?>

```

2. To run the program, type the following command:

```
php MoviesItemOps04.php
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors in the movie.

1. Copy and paste the following program into a file named `MoviesItemOps05.php`.

```

<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

```

```

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . '"
}
');

$eav = $marshaler->marshalJson(
{
    ":num": 3
}
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' => 'remove info.actors[0]',
    'ConditionExpression' => 'size(info.actors) > :num',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];
try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item. ReturnValues are:\n";
    print_r($result['Attributes']);
} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}
?>

```

- To run the program, type the following command:

```
php MoviesItemOps05.php
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

- Modify the program so that the `ConditionExpression` looks like this:

```
ConditionExpression="size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

- Run the program again. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item

You can use the `deleteItem` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into a file named `MoviesItemOps06.php`:

```

<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . ''"
}
');

$eav = $marshaler->marshalJson(
{
    ":val": 5
}
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'ConditionExpression' => 'info.rating <= :val',
    'ExpressionAttributeValues'=> $eav
];

try {
    $result = $dynamodb->deleteItem($params);
    echo "Deleted item.\n";
} catch (DynamoDbException $e) {
    echo "Unable to delete item:\n";
    echo $e->getMessage() . "\n";
}

?>
```

2. To run the program, type the following command:

```
php MoviesItemOps06.php
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition:

```
$params = [  
    'TableName' => $tableName,  
    'Key' => $key  
];
```

4. Run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key). For example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to `query`, there is also a `scan` method that can retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 144\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 145\)](#)
- [Step 4.3: Scan \(p. 147\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the `year` 1985.

1. Copy and paste the following program into a file named `MoviesQuery01.php`:

```
<?php  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
    'version' => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
$marshaller = new Marshaler();
```

```
$tableName = 'Movies';

$eav = $marshaler->marshalJson(
{
    ':yyyy': 1985
});
');

$params = [
    'TableName' => $tableName,
    'KeyConditionExpression' => '#yr = :yyyy',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];

echo "Querying for movies from 1985.\n";

try {
    $result = $dynamodb->query($params);

    echo "Query succeeded.\n";

    foreach ($result['Items'] as $movie) {
        echo $marshaler->unmarshalValue($movie['year']) . ': ' .
            $marshaler->unmarshalValue($movie['title']) . "\n";
    }
}

} catch (DynamoDbException $e) {
    echo "Unable to query:\n";
    echo $e->getMessage() . "\n";
}

?>
```

Note

- `ExpressionAttributeNames` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you can't use it directly in any expression, including `KeyConditionExpression`. You can use the expression attribute name `#yr` to address this.
- `ExpressionAttributeValues` provides value substitution. You use this because you can't use literals in any expression, including `KeyConditionExpression`. You can use the expression attribute value `:yyyy` to address this.

2. To run the program, type the following command:

```
php MoviesItemQuery01.php
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in `year` 1992, with `title` beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into a file named `MoviesQuery02.php`:

```

<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$eav = $marshaler->marshalJson(
{
    ":yyyy":1992,
    ":letter1": "A",
    ":letter2": "L"
}
');

$params = [
    'TableName' => $tableName,
    'ProjectionExpression' => '#yr, title, info.genres, info.actors[0]',
    'KeyConditionExpression' =>
        '#yr = :yyyy and title between :letter1 and :letter2',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];
echo "Querying for movies from 1992 - titles A-L, with genres and lead actor\n";

try {
    $result = $dynamodb->query($params);

    echo "Query succeeded.\n";

    foreach ($result['Items'] as $i) {
        $movie = $marshaler->unmarshalItem($i);
        print $movie['year'] . ':' . $movie['title'] . ' ... ';

        foreach ($movie['info']['genres'] as $gen) {
            print $gen . ' ';
        }

        echo ' ... ' . $movie['info']['actors'][0] . "\n";
    }
} catch (DynamoDbException $e) {
    echo "Unable to query:\n";
    echo $e->getMessage() . "\n";
}

?>

```

2. To run the program, type the following command:

php MoviesQuery02.php

Step 4.3: Scan

The `scan` method reads every item in the entire table, and returns all of the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all of the others.

1. Copy and paste the following program into a file named `MoviesScan.php`:

```

<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region' => 'us-west-2',
    'version' => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();

$marshaler = new Marshaler();

//Expression attribute values
$eav = $marshaler->marshalJson(
{
    ":start_yr": 1950,
    ":end_yr": 1959
}
');

$params = [
    'TableName' => 'Movies',
    'ProjectionExpression' => '#yr, title, info.rating',
    'FilterExpression' => '#yr between :start_yr and :end_yr',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];
echo "Scanning Movies table.\n";

try {
    while (true) {
        $result = $dynamodb->scan($params);

        foreach ($result['Items'] as $i) {
            $movie = $marshaler->unmarshalItem($i);
            echo $movie['year'] . ':' . $movie['title'];
            echo ' ... ' . $movie['info']['rating']
                . "\n";
        }

        if (isset($result['LastEvaluatedKey'])) {

```

```
        $params['ExclusiveStartKey'] = $result['LastEvaluatedKey'];
    } else {
        break;
    }
}

} catch (DynamoDbException $e) {
    echo "Unable to scan:\n";
    echo $e->getMessage() . "\n";
}

?>
```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
 - `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
2. To run the program, type the following command:

```
php MoviesScan.php
```

Note

You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into a file named `MoviesDeleteTable.php`:

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\Aws\Sdk\Exception\DynamoDbException;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();

$params = [
    'TableName' => 'Movies'
];

try {
    $result = $dynamodb->deleteTable($params);
    echo "Deleted table.\n";

} catch (DynamoDbException $e) {
    echo "Unable to delete table:\n";
    echo $e->getMessage() . "\n";
}
```

?>

2. To run the program, type the following command:

```
php MoviesDeleteTable.php
```

Summary

In this tutorial, you created the `Movies` table in DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you need to modify your code so that it uses the Amazon DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB service, you must change the endpoint in your application. To do this, find the following lines in the code:

```
$sdk = new Aws\Sdk([
    'endpoint'  => 'http://localhost:8000',
    'region'    => 'us-west-2',
    'version'   => 'latest'
]);
```

Remove the `endpoint` parameter so that the code looks like this:

```
$sdk = new Aws\Sdk([
    'region'    => 'us-west-2',
    'version'   => 'latest'
]);
```

After you remove this line, the code can access the DynamoDB service in the Region specified by the `region` config value. For example, the following line specifies that you want to use the US West (Oregon) Region:

```
'region'    => 'us-west-2',
```

Instead of using the downloadable version of DynamoDB on your computer, the program now uses the DynamoDB service endpoint in US West (Oregon).

DynamoDB is available in several Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the *AWS General Reference*. For more information about setting Regions and endpoints in your code, see the [boto: A Python interface to Amazon Web Services](#).

Python and DynamoDB

In this tutorial, you use the AWS SDK for Python (Boto 3) to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

You use the downloadable version of DynamoDB in this tutorial. In the [Summary \(p. 164\)](#), we explain how to run the same code against the DynamoDB web service.

As you work through this tutorial, you can refer to the AWS SDK for Python (Boto) documentation at <http://boto.readthedocs.org/en/latest/>. The following sections are specific to DynamoDB:

- [DynamoDB tutorial](#)
- [DynamoDB low-level client](#)

Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Install Python 2.6 or later. For more information, see <https://www.python.org/downloads>. For instructions, see [Quickstart](#) in the Boto 3 documentation.

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy and paste the following program into a file named `MoviesCreateTable.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.create_table(
    TableName='Movies',
    KeySchema=[
        {
            'AttributeName': 'year',
            'KeyType': 'HASH'    #Partition key
        },
        {
            'AttributeName': 'title',
            'KeyType': 'RANGE'  #Sort key
        }
    ],
    AttributeDefinitions=[
        {
            'AttributeName': 'year',
            'AttributeType': 'N'
        },
        {
            'AttributeName': 'title',
            'AttributeType': 'S'
        },
    ],
    BillingMode='PAY_PER_REQUEST'
)

print("Table created successfully")
```

```
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    )

print("Table status:", table.table_status)
```

Note

- You set the endpoint to indicate that you are creating the table in the downloadable version of DynamoDB on your computer.
 - In the `create_table` call, you specify table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required. However, the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)
 - These examples use the Python 3 style `print` function. The line `from __future__ import print_function` enables Python 3 printing in Python 2.6 and later.
2. To run the program, type the following command:

```
python MoviesCreateTable.py
```

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 152\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 152\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
    {  
        "year" : ... ,  
        "title" : ... ,  
        "info" : { ... }  
    },  
    {  
        "year" : ... ,  
        "title" : ... ,  
        "info" : { ... }  
    },  
    ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.

- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{
    "year" : 2013,
    "title" : "Turn It Down, Or Else!",
    "info" : {
        "directors" : [
            "Alice Smith",
            "Bob Jones"
        ],
        "release_date" : "2013-01-18T00:00:00Z",
        "rating" : 6.2,
        "genres" : [
            "Comedy",
            "Drama"
        ],
        "image_url" : "http://ia.media-imdb.com/images/N/
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",
        "rank" : 11,
        "running_time_secs" : 5215,
        "actors" : [
            "David Matthewman",
            "Ann Thomas",
            "Jonathan G. Neff"
        ]
    }
}
```

Step 2.1: Download the Sample Data File

- Download the sample data archive: [moviedata.zip](#)
- Extract the data file (`moviedata.json`) from the archive.
- Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

- Copy and paste the following program into a file named `MoviesLoadData.py`:

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

with open("moviedata.json") as json_file:
    movies = json.load(json_file, parse_float = decimal.Decimal)
    for movie in movies:
        year = int(movie['year'])
```

```
title = movie['title']
info = movie['info']

print("Adding movie:", year, title)

table.put_item(
    Item={
        'year': year,
        'title': title,
        'info': info,
    }
)
```

2. To run the program, type the following command:

```
python MoviesLoadData.py
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 153\)](#)
- [Step 3.2: Read an Item \(p. 154\)](#)
- [Step 3.3: Update an Item \(p. 155\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 156\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 157\)](#)
- [Step 3.6: Delete an Item \(p. 158\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy and paste the following program into a file named `MoviesItemOps01.py`:

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
```

```

year = 2015

response = table.put_item(
    Item={
        'year': year,
        'title': title,
        'info': {
            'plot':"Nothing happens at all.",
            'rating': decimal.Decimal(0)
        }
    }
)

print("PutItem succeeded:")
print(json.dumps(response, indent=4, cls=DecimalEncoder))

```

Note

- The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.
- The `DecimalEncoder` class is used to print out numbers stored using the `Decimal` class. The Boto SDK uses the `Decimal` class to hold DynamoDB number values.

2. To run the program, type the following command:

```
python MoviesItemOps01.py
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

You can use the `get_item` method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into a file named `MoviesItemOps02.py`.

```

from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr
from botocore.exceptions import ClientError

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)

```

```

        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource("dynamodb", region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

try:
    response = table.get_item(
        Key={
            'year': year,
            'title': title
        }
)
except ClientError as e:
    print(e.response['Error']['Message'])
else:
    item = response['Item']
    print("GetItem succeeded:")
    print(json.dumps(item, indent=4, cls=DecimalEncoder))

```

2. To run the program, type the following command:

```
python MoviesItemOps02.py
```

Step 3.3: Update an Item

You can use the `update_item` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

To the following:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
    }
}
```

1. Copy and paste the following program into a file named `MoviesItemOps03.py`:

```

from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

response = table.update_item(
    Key={
        'year': year,
        'title': title
    },
    UpdateExpression="set info.rating = :r, info.plot=:p, info.actors=:a",
    ExpressionAttributeValues={
        ':r': decimal.Decimal(5.5),
        ':p': "Everything happens all at once.",
        ':a': ["Larry", "Moe", "Curly"]
    },
    ReturnValues="UPDATED_NEW"
)

print("UpdateItem succeeded:")
print(json.dumps(response, indent=4, cls=DecimalEncoder))

```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. To run the program, type the following command:

```
python MoviesItemOps03.py
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `update_item` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy and paste the following program into a file named `MoviesItemOps04.py`:

```

from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

response = table.update_item(
    Key={
        'year': year,
        'title': title
    },
    UpdateExpression="set info.rating = info.rating + :val",
    ExpressionAttributeValues={
        ':val': decimal.Decimal(1)
    },
    ReturnValues="UPDATED_NEW"
)

print("UpdateItem succeeded:")
print(json.dumps(response, indent=4, cls=DecimalEncoder))

```

2. To run the program, type the following command:

```
python MoviesItemOps04.py
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors.

1. Copy and paste the following program into a file named `MoviesItemOps05.py`:

```

from __future__ import print_function # Python 2/3 compatibility
import boto3
from botocore.exceptions import ClientError
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):

```

```

if isinstance(o, decimal.Decimal):
    if o % 1 > 0:
        return float(o)
    else:
        return int(o)
    return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

# Conditional update (will fail)
print("Attempting conditional update...")

try:
    response = table.update_item(
        Key={
            'year': year,
            'title': title
        },
        UpdateExpression="remove info.actors[0]",
        ConditionExpression="size(info.actors) > :num",
        ExpressionAttributeValues={
            ':num': 3
        },
        ReturnValues="UPDATED_NEW"
    )
except ClientError as e:
    if e.response['Error']['Code'] == "ConditionalCheckFailedException":
        print(e.response['Error']['Message'])
    else:
        raise
else:
    print("UpdateItem succeeded:")
    print(json.dumps(response, indent=4, cls=DecimalEncoder))

```

- To run the program, type the following command:

```
python MoviesItemOps05.py
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

- Modify the program so that the `ConditionExpression` looks like this:

```
ConditionExpression="size(info.actors) >= :num",
```

The condition is now *greater than or equal to* 3 instead of *greater than* 3.

- Run the program again. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item

You can use the `delete_item` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into a file named `MoviesItemOps06.py`:

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
from botocore.exceptions import ClientError
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

print("Attempting a conditional delete...")

try:
    response = table.delete_item(
        Key={
            'year': year,
            'title': title
        },
        ConditionExpression="info.rating <= :val",
        ExpressionAttributeValues= {
            ":val": decimal.Decimal(5)
        }
    )
except ClientError as e:
    if e.response['Error']['Code'] == "ConditionalCheckFailedException":
        print(e.response['Error']['Message'])
    else:
        raise
else:
    print("DeleteItem succeeded:")
    print(json.dumps(response, indent=4, cls=DecimalEncoder))
```

2. To run the program, type the following command:

```
python MoviesItemOps06.py
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition in `table.delete_item`:

```
response = table.delete_item(
    Key={
```

```
        'year': year,
        'title': title
    )
)
```

- Run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key); for example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to `query`, there is also a `scan` method that can retrieve all the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 160\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 161\)](#)
- [Step 4.3: Scan \(p. 162\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the `year` 1985.

1. Copy and paste the following program into a file named `MoviesQuery01.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')
```

```

print("Movies from 1985")

response = table.query(
    KeyConditionExpression=Key('year').eq(1985)
)

for i in response['Items']:
    print(i['year'], ":", i['title'])

```

Note

The Boto 3 SDK constructs a `ConditionExpression` for you when you use the `Key` and `Attr` functions imported from `boto3.dynamodb.conditions`. You can also specify a `ConditionExpression` as a string. For a list of available conditions for DynamoDB, see the [DynamoDB Conditions in AWS SDK for Python \(Boto 3\) Getting Started](#). For more information, see [Condition Expressions \(p. 346\)](#).

2. To run the program, type the following command:

```
python MoviesQuery01.py
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in `year` 1992, with `title` beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into a file named `MoviesQuery02.py`:

```

from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            return str(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

print("Movies from 1992 - titles A-L, with genres and lead actor")

response = table.query(
    ProjectionExpression="#yr, title, info.genres, info.actors[0]",
    ExpressionAttributeNames={ "#yr": "year" }, # Expression Attribute Names for Projection Expression only.

```

```

        KeyConditionExpression=Key('year').eq(1992) & Key('title').between('A', 'L')
    )

    for i in response[u'Items']:
        print(json.dumps(i, cls=DecimalEncoder))

```

2. To run the program, type the following command:

```
python MoviesQuery02.py
```

Step 4.3: Scan

The `scan` method reads every item in the entire table, and returns all the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The `scan` specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all the others.

1. Copy and paste the following program into a file named `MoviesScan.py`:

```

from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

fe = Key('year').between(1950, 1959);
pe = "#yr, title, info.rating"
# Expression Attribute Names for Projection Expression only.
ean = { "#yr": "year", }
esk = None

response = table.scan(
    FilterExpression=fe,
    ProjectionExpression=pe,
    ExpressionAttributeNames=ean
)

for i in response['Items']:
    print(json.dumps(i, cls=DecimalEncoder))

while 'LastEvaluatedKey' in response:
    response = table.scan(
        ProjectionExpression=pe,

```

```
    FilterExpression=fe,
    ExpressionAttributeNames= ean,
    ExclusiveStartKey=response['LastEvaluatedKey']
    )

for i in response['Items']:
    print(json.dumps(i, cls=DecimalEncoder))
```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
- The `scan` method returns a subset of the items each time, called a page. The `LastEvaluatedKey` value in the response is then passed to the `scan` method via the `ExclusiveStartKey` parameter. When the last page is returned, `LastEvaluatedKey` is not part of the response.

Note

- `ExpressionAttributeNames` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you can't use it directly in any expression, including `KeyConditionExpression`. You can use the expression attribute name `#yr` to address this.
- `ExpressionAttributeValues` provides value substitution. You use this because you can't use literals in any expression, including `KeyConditionExpression`. You can use the expression attribute value `:yyyy` to address this.

2. To run the program, type the following command:

```
python MoviesScan.py
```

Note

You can also use the `Scan` operation with any secondary indexes that you create on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into a file named `MoviesDeleteTable.py`:

```
from __future__ import print_function # Python 2/3 compatibility
import boto3

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

table.delete()
```

2. To run the program, type the following command:

```
python MoviesDeleteTable.py
```

Summary

In this tutorial, you created the `Movies` table in the downloadable version of DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the Amazon DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the Amazon DynamoDB service, you must change the endpoint in your application. To do this, modify the following line:

```
dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")
```

For example, if you want to use the `us-west-2` Region:

```
dynamodb = boto3.resource('dynamodb', region_name='us-west-2')
```

Instead of using the downloadable version of DynamoDB on your computer, the program now uses the DynamoDB service in US West (Oregon).

DynamoDB is available in several Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information about setting Regions and endpoints in your code, see [AWS Region Selection](#) in the [AWS SDK for Java Developer Guide](#).

Ruby and DynamoDB

In this tutorial, you use the AWS SDK for Ruby to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB service, see the [Summary \(p. 179\)](#).

As you work through this tutorial, you can refer to the [AWS SDK for Ruby API Reference](#). The [DynamoDB section](#) describes the parameters and results for DynamoDB operations.

Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 41\)](#).
- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).
- Set up the AWS SDK for Ruby:
 - Go to <https://www.ruby-lang.org/en/documentation/installation/> and install Ruby.
 - Go to <https://aws.amazon.com/sdk-for-ruby> and install the AWS SDK for Ruby.

For more information, see [Installation](#) in the *AWS SDK for Ruby API Reference*.

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following two attributes:

- `year` – The partition key. The `attribute_type` is `N` for number.
- `title` – The sort key. The `attribute_type` is `S` for string.

1. Copy and paste the following program into a file named `MoviesCreateTable.rb`:

```
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

params = {
  table_name: "Movies",
  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
  }
}

begin
  result = dynamodb.create_table(params)
  puts "Created table. Status: " +
    result.table_description.table_status;

rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to create table:"
  puts "#{error.message}"
end
```

Note

- You set the endpoint to indicate that you are creating the table in the downloadable version of DynamoDB on your computer.
 - In the `create_table` call, you specify table name, primary key attributes, and its data types.
 - The `provisioned_throughput` parameter is required. However, the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)
2. To run the program, type the following command:

```
ruby MoviesCreateTable.rb
```

To learn more about managing tables, see [Working with Tables in DynamoDB \(p. 290\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 167\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 167\)](#)

You use a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ...,  
    "title" : ...,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- You store the rest of the `info` values in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{  
  "year" : 2013,
```

```
        "title" : "Turn It Down, Or Else!",
        "info" : {
            "directors" : [
                "Alice Smith",
                "Bob Jones"
            ],
            "release_date" : "2013-01-18T00:00:00Z",
            "rating" : 6.2,
            "genres" : [
                "Comedy",
                "Drama"
            ],
            "image_url" : "http://ia.media-imdb.com/images/N/
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",
            "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",
            "rank" : 11,
            "running_time_secs" : 5215,
            "actors" : [
                "David Matthewman",
                "Ann Thomas",
                "Jonathan G. Neff"
            ]
        }
    }
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy and paste the following program into a file named `MoviesLoadData.rb`:

```
require "aws-sdk"
require "json"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

file = File.read('moviedata.json')
movies = JSON.parse(file)
movies.each{|movie| 

  params = {
    table_name: tableName,
    item: movie
  }

  begin
    result = dynamodb.put_item(params)
    puts "Added movie: #{movie["year"]} #{movie["title"]}"
  end
}
```

```
rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to add movie:"
    puts "#{error.message}"
end
}
```

2. To run the program, type the following command:

```
ruby MoviesLoadData.rb
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items in DynamoDB \(p. 327\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 168\)](#)
- [Step 3.2: Read an Item \(p. 169\)](#)
- [Step 3.3: Update an Item \(p. 170\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 171\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 172\)](#)
- [Step 3.6: Delete an Item \(p. 173\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the table.

1. Copy and paste the following program into a file named `MoviesItemOps01.rb`:

```
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

item = {
  year: year,
  title: title,
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}

params = {
  table_name: "Movies",
```

```

    item: item
}

begin
    result = dynamodb.put_item(params)
    puts "Added item: #{year} - #{title}"

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to add item:"
    puts "#{error.message}"
end

```

Note

The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores a map that provides more information about the movie.

2. To run the program, type the following command:

```
ruby MoviesItemOps01.rb
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

You can use the `get_item` method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its `year` and `title`.

1. Copy and paste the following program into a file named `MoviesItemOps02.rb`:

```

require "aws-sdk"

Aws.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

key = {
    year: year,
    title: title
}

params = {
    table_name: "Movies",
    key: {

```

```

        year: year,
        title: title
    }
}

begin
    result = dynamodb.get_item(params)
    printf "%i - %s\n%s\n%d\n",
        result.item["year"],
        result.item["title"],
        result.item["info"]["plot"],
        result.item["info"]["rating"]

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to read item:"
    puts "#{$error.message}"
end

```

2. To run the program, type the following command:

```
ruby MoviesItemOps02.rb
```

Step 3.3: Update an Item

You can use the `update_item` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from this:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

To the following:

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
    }
}
```

1. Copy and paste the following program into a file named `MoviesItemOps03.rb`:

```
require "aws-sdk"
```

```

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
  table_name: "Movies",
  key: {
    year: year,
    title: title
  },
  update_expression: "set info.rating = :r, info.plot=:p, info.actors=:a",
  expression_attribute_values: {
    ":r" => 5.5,
    ":p" => "Everything happens all at once.", # value
<Hash,Array,String,Numeric,Boolean,IO,Set,nil>
    ":a" => ["Larry", "Moe", "Curly"]
  },
  return_values: "UPDATED_NEW"
}

begin
  result = dynamodb.update_item(params)
  puts "Added item: #{year} - #{title}"

rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to add item:"
  puts "#{error.message}"
end

```

Note

This program uses `update_expression` to describe all updates you want to perform on the specified item. The `return_values` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. To run the program, type the following command:

```
ruby MoviesItemOps03.rb
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `update_item` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy and paste the following program into a file named `MoviesItemOps04.rb`:

```

require "aws-sdk"

Aws.config.update({
```

```

    region: "us-west-2",
    endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
  table_name: "Movies",
  key: {
    year: year,
    title: title
  },
  update_expression: "set info.rating = info.rating + :val",
  expression_attribute_values: {
    ":val" => 1
  },
  return_values: "UPDATED_NEW"
}

begin
  result = dynamodb.update_item(params)
  puts "Updated item. ReturnValues are:"
  result.attributes["info"].each do |key, value|
    if key == "rating"
      puts "#{key}: #{value.to_f}"
    else
      puts "#{key}: #{value}"
    end
  end
rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to update item:"
  puts "#{error.message}"
end

```

2. To run the program, type the following command:

```
ruby MoviesItemOps04.rb
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `update_item` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors.

1. Copy and paste the following program into a file named `MoviesItemOps05.rb`:

```

require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

```

```

year = 2015
title = "The Big New Movie"

params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    },
    update_expression: "remove info.actors[0]",
    condition_expression: "size(info.actors) > :num",
    expression_attribute_values: {
        ":num" => 3
    },
    return_values: "UPDATED_NEW"
}

begin
    result = dynamodb.update_item(params)
    puts "Updated item. ReturnValues are:"
    result.attributes["info"].each do |key, value|
        if key == "rating"
            puts "#{key}: #{value.to_f}"
        else
            puts "#{key}: #{value}"
        end
    end

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to update item:"
    puts "#{error.message}"
end

```

2. To run the program, type the following command:

```
ruby MoviesItemOps05.rb
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the `ConditionExpression` looks like this:

```
condition_expression: "size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Run the program again. The `update_item` method should now succeed.

Step 3.6: Delete an Item

You can use the `delete_item` method to delete one item by specifying its primary key. You can optionally provide a `condition_expression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy and paste the following program into a file named `MoviesItemOps06.rb`:

```
require "aws-sdk"
```

```

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
  table_name: "Movies",
  key: {
    year: year,
    title: title
  },
  condition_expression: "info.rating <= :val",
  expression_attribute_values: {
    ":val" => 5
  }
}

begin
  result = dynamodb.delete_item(params)
  puts "Deleted item."
rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to update item:"
  puts "#{error.message}"
end

```

2. To run the program, type the following command:

```
ruby MoviesItemOps06.rb
```

The program should fail with the following message:

```
The conditional request failed
```

This is because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition:

```

params = {
  table_name: "Movies",
  key: {
    year: year,
    title: title
  }
}

```

4. Run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- **year** – The partition key. The attribute type is number.
- **title** – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key); for example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to `query`, there is also a `scan` method that can retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries \(p. 410\)](#) and [Working with Scans \(p. 427\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 175\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 176\)](#)
- [Step 4.3: Scan \(p. 177\)](#)

Step 4.1: Query - All Movies Released in a Year

The following program retrieves all movies released in the `year` 1985.

1. Copy and paste the following program into a file named `MoviesQuery01.rb`:

```
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = "Movies"

params = {
  table_name: tableName,
  key_condition_expression: "#yr = :yyyy",
  expression_attribute_names: {
    "#yr" => "year"
  },
  expression_attribute_values: {
    ":yyyy" => 1985
  }
}

puts "Querying for movies from 1985.";

begin
  result = dynamodb.query(params)
  puts "Query succeeded."

  result.items.each{|movie|
    puts "#{movie["year"].to_i} #{movie["title"]}"
  }

rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to query table:"
  puts "#{error.message}"
end
```

Note

- `expression_attribute_names` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you can't use it directly in any expression, including `KeyConditionExpression`. You can use the expression attribute name `#yr` to address this.
- `expression_attribute_values` provides value substitution. You use this because you can't use literals in any expression, including `key_condition_expression`. You can use the expression attribute value `:yyyy` to address this.

2. To run the program, type the following command:

```
ruby MoviesItemQuery01.rb
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The following program retrieves all movies released in `year` 1992, with a `title` beginning with the letter "A" through the letter "L".

1. Copy and paste the following program into a file named `MoviesQuery02.rb`:

```
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = "Movies"

params = {
  table_name: tableName,
  projection_expression: "#yr, title, info.genres, info.actors[0]",
  key_condition_expression:
    "#yr = :yyyy and title between :letter1 and :letter2",
  expression_attribute_names: {
    "#yr" => "year"
  },
  expression_attribute_values: {
    ":yyyy" => 1992,
    ":letter1" => "A",
    ":letter2" => "L"
  }
}

puts "Querying for movies from 1992 - titles A-L, with genres and lead actor";

begin
  result = dynamodb.query(params)
  puts "Query succeeded."
```

```

result.items.each{|movie|
    print "#{movie["year"].to_i}: #{movie["title"]} ... "

    movie['info']['genres'].each{|gen|
        print gen + " "
    }

    print "... #{movie["info"]["actors"][0]}\n"
}

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to query table:"
    puts "#{@error.message}"
end

```

2. To run the program, type the following command:

```
ruby MoviesQuery02.rb
```

Step 4.3: Scan

The `scan` method reads every item in the entire table, and returns all the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, note that the filter is only applied after the entire table has been scanned.

The following program scans the `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all the others.

1. Copy and paste the following program into a file named `MoviesScan.rb`:

```

require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

tableName = "Movies"

params = {
  table_name: tableName,
  projection_expression: "#yr, title, info.rating",
  filter_expression: "#yr between :start_yr and :end_yr",
  expression_attribute_names: {"#yr"=> "year"},
  expression_attribute_values: {
    ":start_yr" => 1950,
    ":end_yr" => 1959
  }
}

puts "Scanning Movies table."

begin
  loop do
    result = dynamodb.scan(params)

    result.items.each{|movie|
      puts "#{movie["year"].to_i}: " +
        "#{movie["title"]} ... "
    }
  end
end

```

```
        "#{movie["info"]["rating"].to_f}"
    }

    break if result.last_evaluated_key.nil?

    puts "Scanning for more..."
    params[:exclusive_start_key] = result.last_evaluated_key
end

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to scan:"
    puts "#{error.message}"
end
```

In the code, note the following:

- `projection_expression` specifies the attributes you want in the scan result.
- `filter_expression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.

2. To run the program, type the following command:

```
ruby MoviesScan.rb
```

Note

You can also use the `scan` method with any secondary indexes that you create on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy and paste the following program into a file named `MoviesDeleteTable.rb`:

```
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

params = {
  table_name: "Movies"
}

begin
  result = dynamodb.delete_table(params)
  puts "Deleted table."
end

rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to delete table:"
  puts "#{error.message}"
end
```

2. To run the program, type the following command:

```
ruby MoviesDeleteTable.rb
```

Summary

In this tutorial, you created the `Movies` table in the downloadable version of DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the Amazon DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the Amazon DynamoDB service, you must change the endpoint in your application in order. To do this, find the following lines in the code:

```
Aws.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
})
```

Remove the `endpoint` parameter so that the code looks like this:

```
Aws.config.update({  
    region: "us-west-2"  
});
```

After you remove this line, the code can access the DynamoDB service in the Region specified by the `region` config value.

Instead of using the version of DynamoDB on your computer, the program uses the DynamoDB service endpoint in US West (Oregon).

DynamoDB is available in several Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information, see the [AWS SDK for Ruby Getting Started Guide](#).

Programming with DynamoDB and the AWS SDKs

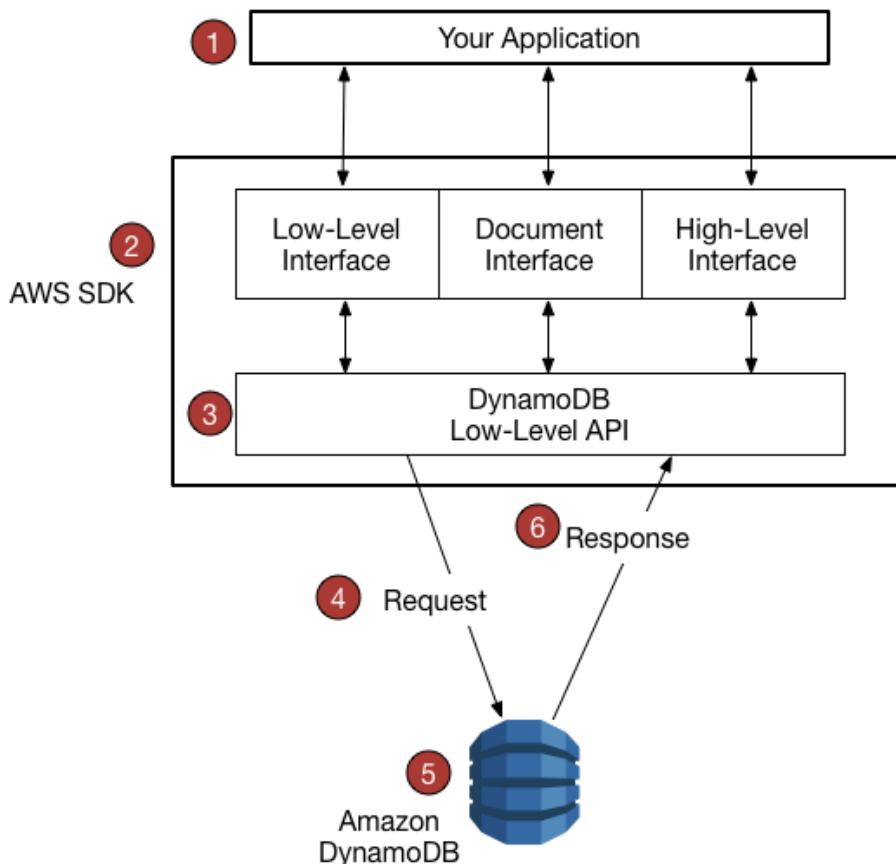
This chapter covers developer-related topics. If you want to run code samples instead, see [Running the Code Samples In This Developer Guide \(p. 280\)](#).

Topics

- [Overview of AWS SDK Support for DynamoDB \(p. 180\)](#)
- [Programmatic Interfaces \(p. 182\)](#)
- [DynamoDB Low-Level API \(p. 185\)](#)
- [Error Handling \(p. 189\)](#)
- [Higher-Level Programming Interfaces for DynamoDB \(p. 194\)](#)
- [Running the Code Samples In This Developer Guide \(p. 280\)](#)

Overview of AWS SDK Support for DynamoDB

The following diagram provides a high-level overview of DynamoDB application programming with 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.
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 executes 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:

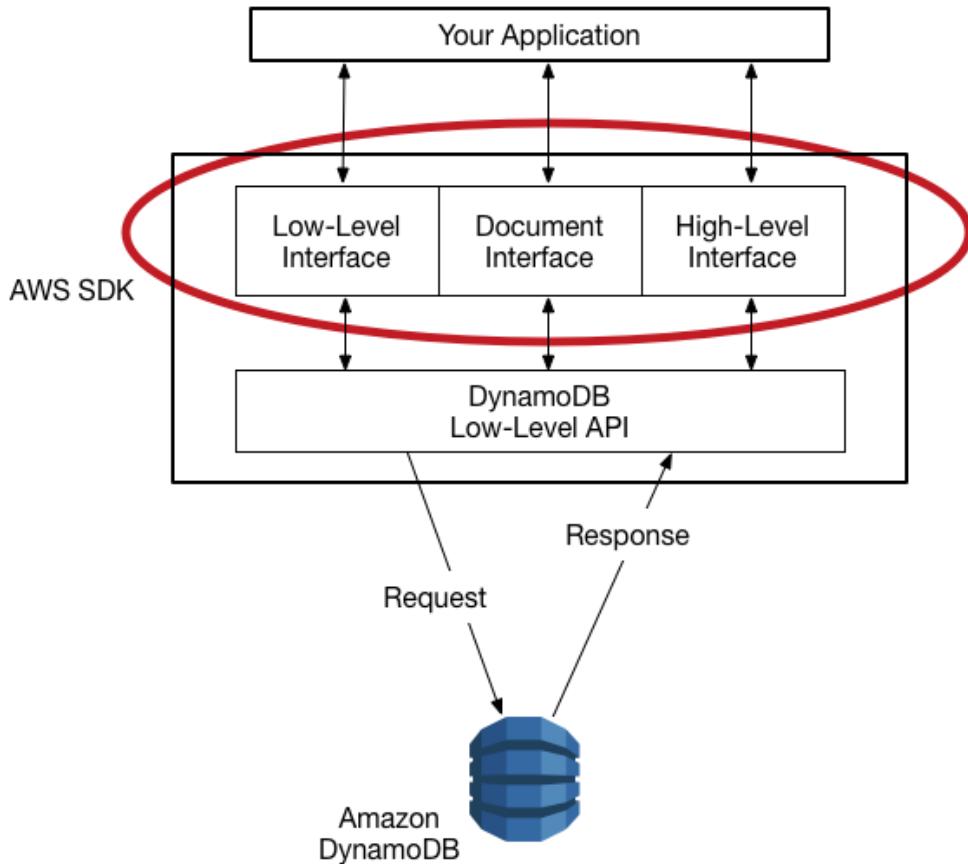
- Formatting HTTP(S) requests and serializing request parameters.
- Generating a cryptographic signature for each request.
- Forwarding request to a DynamoDB endpoint and receiving responses from DynamoDB.
- Extracting the results from those responses.
- Implementing basic retry logic in case of errors.

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 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 \(p. 182\)](#)
- [Document Interfaces \(p. 183\)](#)
- [Object Persistence Interface \(p. 184\)](#)

Low-Level Interfaces

Every language-specific AWS SDK provides a low-level interface for 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 \(p. 188\)](#), 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. The program issues a `GetItem` request for a song in the `Music` table, and prints the year that the song was released.

The `com.amazonaws.services.dynamodbv2.AmazonDynamoDB` class implements the DynamoDB low-level interface.

```
package com.amazonaws.codesamples;

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 MusicLowLevelDemo {

    public static void main(String[] args) {

        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("Call Me Today"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music")
            .withKey(key);

        try {
            GetItemResult result = client.getItem(request);
            if (result && result.getItem() != null) {
                AttributeValue year = result.getItem().get("Year");
                System.out.println("The song was released in " + year.getN());
            } else {
                System.out.println("No matching song was found");
            }
        } catch (Exception e) {
            System.err.println("Unable to retrieve data: ");
            System.err.println(e.getMessage());
        }
    }
}
```

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 \(p. 188\)](#); 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 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 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 \(p. 194\)](#).

The following Java program uses `DynamoDBMapper`, the object persistence interface of the AWS SDK for Java. The `MusicItem` class represents an item in the `Music` table.

```

package com.amazonaws.codesamples;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="Music")
public class MusicItem {
    private String artist;
    private String songTitle;
    private String albumTitle;
    private int year;

    @DynamoDBHashKey(attributeName="Artist")
    public String getArtist() { return artist; }
    public void setArtist(String artist) {this.artist = artist; }

    @DynamoDBRangeKey(attributeName="SongTitle")
    public String getSongTitle() { return songTitle; }
    public void setSongTitle(String songTitle) {this.songTitle = songTitle; }

    @DynamoDBAttribute(attributeName = "AlbumTitle")

```

```
public String getAlbumTitle() { return albumTitle; }
public void setAlbumTitle(String albumTitle) {this.albumTitle = albumTitle; }

@DynamoDBAttribute(attributeName = "Year")
public int getYear() { return year; }
public void setYear(int year) { this.year = year; }
}
```

You can then instantiate a `MusicItem` object, and retrieve a song using the `load()` method of `DynamoDBMapper`. The program then prints the year that the song was released.

The `com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper` class implements the DynamoDB object persistence interface. Note how `DynamoDBMapper` acts as a wrapper around the low-level client (`AmazonDynamoDB`).

```
package com.amazonaws.codesamples;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;

public class MusicMapperDemo {

    public static void main(String[] args) {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

        DynamoDBMapper mapper = new DynamoDBMapper(client);

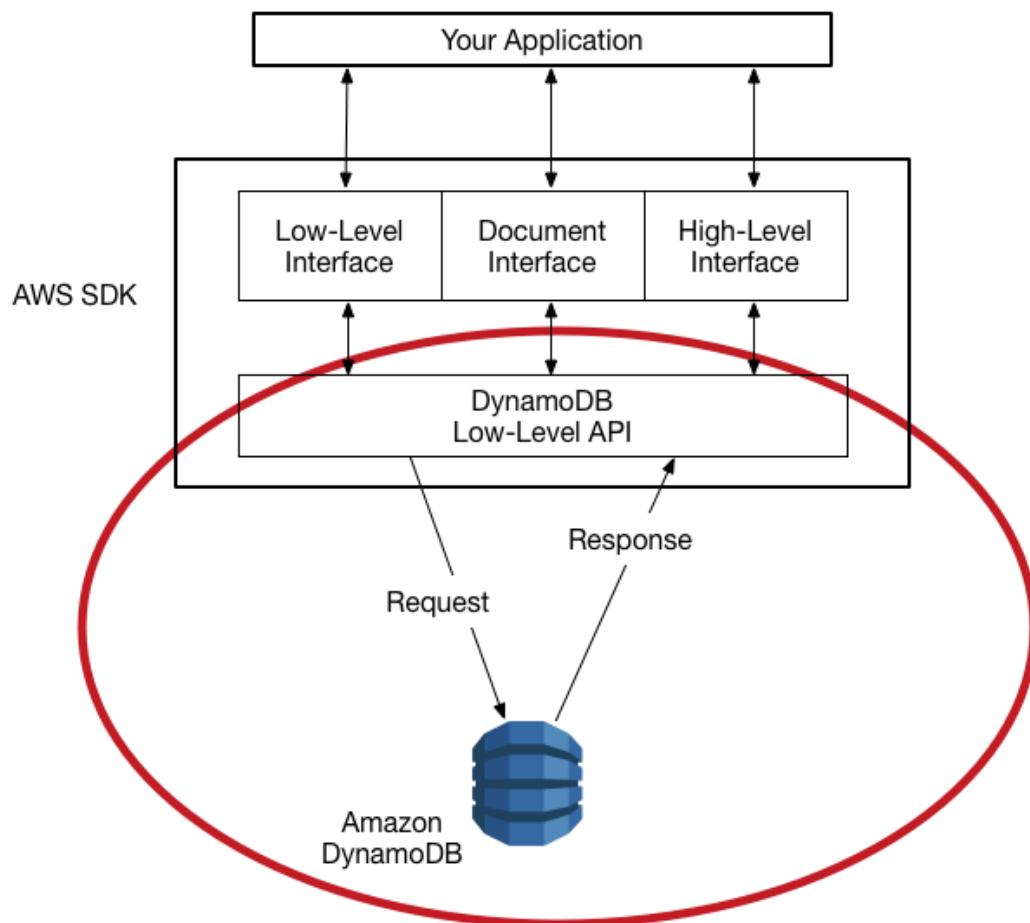
        MusicItem keySchema = new MusicItem();
        keySchema.setArtist("No One You Know");
        keySchema.setSongTitle("Call Me Today");

        try {
            MusicItem result = mapper.load(keySchema);
            if (result != null) {
                System.out.println(
                    "The song was released in " + result.getYear());
            } else {
                System.out.println("No matching song was found");
            }
        } catch (Exception e) {
            System.err.println("Unable to retrieve data: ");
            System.err.println(e.getMessage());
        }
    }
}
```

DynamoDB Low-Level API

Topics

- [Request Format \(p. 187\)](#)
- [Response Format \(p. 187\)](#)
- [Data Type Descriptors \(p. 188\)](#)
- [Numeric Data \(p. 188\)](#)
- [Binary Data \(p. 189\)](#)



The DynamoDB *low-level API* is the protocol-level interface for Amazon 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 [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#). 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, but DynamoDB does not store data persistently in JSON format.

Note

For more information about JSON, see the [Introducing JSON](#) at 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 a request as shown following:

```
POST / HTTP/1.1
Host: dynamodb.<region>.<domain>;
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 above, the HTTP(S) response payload contains the results from the operation, as in this 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 cannot 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 \(p. 189\)](#).

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 \(p. 187\)](#) and [Response Format \(p. 187\)](#) 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 \(p. 12\)](#).

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](#) at the Internet Engineering Task Force (IETF) website.

Error Handling

This section describes runtime errors and how to handle them. It also describes error messages and codes that are specific to DynamoDB.

Topics

- [Error Components \(p. 189\)](#)
- [Error Messages and Codes \(p. 190\)](#)
- [Error Handling in Your Application \(p. 192\)](#)
- [Error Retries and Exponential Backoff \(p. 193\)](#)
- [Batch Operations and Error Handling \(p. 194\)](#)

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 will 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: LDM6CJP8RMQ1FHKSC1RBVJFPNVV4KQNSO5AEMF66Q9ASUAAJG
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"}
```

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 will 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 will 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 \(p. 492\)](#).

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 10.

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 \(p. 185\)](#).

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 \(p. 193\)](#).

OK to retry? Yes

ResourceInUseException

Message: *The resource which you are attempting to change is in use.*

Example: You tried to recreate an existing table, or delete a table currently in the `CREATING` state.

OK to retry? No

ResourceNotFoundException

Message: *Requested resource not found.*

Example: Table which 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 might be returned if you perform any of the following operations too rapidly: `CreateTable`; `UpdateTable`; `DeleteTable`.

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 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 Amazon Web Services. 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.

Internal Server Error (HTTP 500)

DynamoDB could not process your request.

OK to retry? Yes

Note

You may encounter Internal Server Errors while working with items. These are expected during the lifetime of a table. Any failed requests can be retried immediately.

Service Unavailable (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 will need to add logic to catch and respond to errors. Typical approaches include using `try-catch` blocks or an `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.

The following Java code snippet attempts to delete an item from a DynamoDB table, and performs rudimentary error handling. (In this case, it simply informs the user that the request failed).

```
Table table = dynamoDB.getTable("Movies");

try {
    Item item = table.getItem("year", 1978, "title", "Superman");
```

```

        if (item != null) {
            System.out.println("Result: " + item);
        } else {
            //No such item exists in the table
            System.out.println("Item not found");
        }

    } catch (AmazonServiceException ase) {
        System.err.println("Could not complete operation");
        System.err.println("Error Message: " + ase.getMessage());
        System.err.println("HTTP Status: " + ase.getStatusCode());
        System.err.println("AWS Error Code: " + ase.getErrorCode());
        System.err.println("Error Type: " + ase.getErrorType());
        System.err.println("Request ID: " + ase.getRequestId());

    } catch (AmazonClientException ace) {
        System.err.println("Internal error occurred communicating with DynamoDB");
        System.out.println("Error Message: " + ace.getMessage());
    }
}

```

In this code snippet, the `try-catch` construct handles two different kinds of exceptions:

- `AmazonServiceException`—thrown if the client request was correctly transmitted to DynamoDB, but DynamoDB was unable to process the request and returned an error response instead.
- `AmazonClientException`—thrown if the client was unable to get a response from a service, or if the client was unable to parse the response from a service.

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 and reduces operational costs for the developer.

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 you need to revise the request itself to correct the problem before trying again.

In addition to simple retries, each AWS SDK implements 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. For more information, see [Best Practices for Tables \(p. 666\)](#).

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 for [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 will not cause the entire `BatchGetItem` operation to fail. On the other hand, if *all* of the five reads operations fail, then the entire `BatchGetItem` will fail.

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` parameter of the request. 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: DynamoDBMapper \(p. 195\)](#)

.NET

- [.NET: Document Model \(p. 232\)](#)
- [.NET: Object Persistence Model \(p. 253\)](#)

Java: DynamoDBMapper

Topics

- [Supported Data Types \(p. 197\)](#)
- [Java Annotations for DynamoDB \(p. 198\)](#)
- [The DynamoDBMapper Class \(p. 202\)](#)
- [Optional Configuration Settings for DynamoDBMapper \(p. 209\)](#)
- [Example: CRUD Operations \(p. 210\)](#)
- [Example: Batch Write Operations \(p. 212\)](#)
- [Example: Query and Scan \(p. 218\)](#)
- [Optimistic Locking With Version Number \(p. 227\)](#)
- [Mapping Arbitrary Data \(p. 229\)](#)

The AWS SDK for Java provides a `DynamoDBMapper` class, allowing you to map your client-side classes to 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 access your tables, perform various create, read, update and delete (CRUD) operations, and execute queries.

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 Tables: Java \(p. 316\)](#).

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 snippet 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 Sample Data \(p. 280\)](#), 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 `SomeProp` property is marked with the `@DynamoDBIgnore` annotation. When you upload a `CatalogItem` instance to the table, your `DynamoDBMapper` instance does not include `SomeProp` property. In addition, the mapper does not return this attribute when you retrieve an item from the table.

After you have defined 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 snippet 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 snippet 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 a number of built-in features such as optimistic locking, auto-generated partition key and sort key values, and object versioning.

Supported Data Types

This section describes the supported primitive Java data types, collections, and arbitrary data types.

DynamoDB supports the following primitive data types and primitive wrapper classes.

- `String`
- `Boolean, boolean`
- `Byte, byte`
- `Date` (as ISO8601 millisecond-precision string, shifted to UTC)
- `Calendar` (as ISO8601 millisecond-precision string, shifted to UTC)
- `Long, long`
- `Integer, int`
- `Double, double`
- `Float, float`
- `BigDecimal`
- `BigInteger`

DynamoDB supports the Java `Set` collection types. If your mapped collection property is not a Set, then an exception is thrown.

The following table summarizes how the preceding Java types map to the DynamoDB types.

Java type	DynamoDB type
All number types	<code>N</code> (number type)
Strings	<code>S</code> (string type)
Boolean	<code>N</code> (number type), 0 or 1. Alternatively, you can use <code>@DynamoDBNativeBooleanType</code> to map a Java Boolean to the DynamoDB <code>BOOL</code> data type. For more information, see Java Annotations for DynamoDB (p. 198) .
ByteBuffer	<code>B</code> (binary type)

Java type	DynamoDB 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 \(p. 229\)](#).

Java Annotations for DynamoDB

This section describes the annotations that are available for mapping your classes and properties to tables and attributes.

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 \(p. 198\)](#)
- [DynamoDBAutoGeneratedKey \(p. 198\)](#)
- [DynamoDBDocument \(p. 199\)](#)
- [DynamoDBHashKey \(p. 200\)](#)
- [DynamoDBIgnore \(p. 200\)](#)
- [DynamoDBIndexHashKey \(p. 201\)](#)
- [DynamoDBIndexRangeKey \(p. 201\)](#)
- [DynamoDBRangeKey \(p. 201\)](#)
- [DynamoDBTable \(p. 201\)](#)
- [DynamoDBTypeConverted \(p. 202\)](#)
- [DynamoDBTyped \(p. 202\)](#)
- [DynamoDBVersionAttribute \(p. 202\)](#)

[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 auto-generated. `DynamoDBMapper` will generate a random [UUID](#) when saving these attributes. Only String properties can be marked as auto-generated keys.

The following snippet demonstrates using auto-generated 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());
    }
}
```

[DynamoDBDocument](#)

Indicates that a class can be serialized as a DynamoDB document.

For example, suppose you wanted to map a JSON document to a DynamoDB attribute of type Map (M). The following code snippet 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 snippet:

```

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(pix);

item.setId(123);

mapper.save(item);

```

The resulting `ProductCatalog` item would look like this (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; it cannot be a collection type.

Assume that you have a table, `ProductCatalog`, that has `Id` as the primary key. The following Java code snippet 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.

[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; it cannot 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 must also 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; it cannot 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 snippet 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 snippet 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 snippet.

```
@DynamoDBTable(tableName="Managers")
public class Manager extends Developer { ...}
```

DynamoDBTypeConverted

Annotation to mark a property as using a custom type-converter. May 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 \(p. 229\)](#).

DynamoDBTyped

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 type, see [Data Types \(p. 12\)](#). For more information about versioning, see [Optimistic Locking With Version Number \(p. 227\)](#).

The `DynamoDBMapper` Class

The `DynamoDBMapper` class is the entry point to DynamoDB. It provides access to a DynamoDB endpoint and enables you to access your data in various tables, perform various CRUD operations on items, and execute 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 \(p. 203\)](#)
- [load \(p. 203\)](#)
- [delete \(p. 203\)](#)
- [query \(p. 203\)](#)
- [queryPage \(p. 205\)](#)
- [scan \(p. 205\)](#)
- [scanPage \(p. 206\)](#)
- [parallelScan \(p. 206\)](#)
- [batchSave \(p. 206\)](#)
- [batchLoad \(p. 207\)](#)
- [batchDelete \(p. 207\)](#)

- [batchWrite \(p. 207\)](#)
- [count \(p. 208\)](#)
- [generateCreateTableRequest \(p. 208\)](#)
- [createS3Link \(p. 208\)](#)
- [getS3ClientCache \(p. 209\)](#)

save

Saves the specified object to the table. The object that you wish 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 annotated with `@DynamoDBAutoGeneratedKey`, then they are given a random universally unique identifier (UUID) if left uninitialized. Version fields annotated with `@DynamoDBVersionAttribute` will be 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.

```
mapper.save(obj, new DynamoDBMapperConfig(DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

If you have versioning enabled, then 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 \(p. 227\)](#).

load

Retrieves an item from a table. You must provide the primary key of the item that you wish 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.

```
CatalogItem item = mapper.load(CatalogItem.class, item.getId(),  
                               new  
                               DynamoDBMapperConfig(DynamoDBMapperConfig.ConsistentReads.CONSISTENT));
```

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 \(p. 15\)](#).

delete

Deletes an item from the table. You must pass in an object instance of the mapped class.

If you have versioning enabled, then 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 \(p. 227\)](#).

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 0 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, ... )
```

Now, assume that you have created a mapping between a `Reply` class and the corresponding `Reply` table in DynamoDB. The following Java code snippet uses `DynamoDBMapper` to find all replies in the past two weeks for a specific thread subject.

Example

```
String forumName = "DynamoDB";
String forumSubject = "DynamoDB 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, you only need to iterate over the `latestReplies` collection.

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 this:

```
@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 must specify `withIndexName` so that DynamoDB knows which index to query. In the following code snippet, we are querying a global secondary index. Because global secondary indexes support eventually consistent reads, but not strongly consistent reads, we 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` will only return the first "page" of data - that is, the amount of data that will fit within 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 0 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 have mapped a Java class to the `Reply` table, you can use the `DynamoDBMapper` to scan the table. For example, the following Java code snippet 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, you only need to iterate over the `replies` collection.

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 \(p. 203\)](#) section, where we use the `@DynamoDBIndexHashKey` and `@DynamoDBIndexRangeKey` annotations to specify the index partition key and sort key.

The following code snippet 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> indexItems = 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` will only return the first "page" of data - that is, the amount of data that will fit 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 snippet 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 sample illustrating usage of `parallelScan`, see [Example: Query and Scan \(p. 218\)](#).

[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 snippet saves two items (books) to the ProductCatalog table.

```
Book book1 = new Book();
book1.id = 901;
```

```
book1.productCategory = "Book";
book1.title = "Book 901 Title";

Book book2 = new Book();
book2.id = 902;
book2.productCategory = "Book";
book2.title = "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 snippet 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 snippet 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 snippet 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.name = "Test BatchWrite Forum";

// Create a Thread item to save
Thread threadItem = new Thread();
threadItem.forumName = "AmazonDynamoDB";
threadItem.subject = "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);
```

[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 snippet 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.id = 150;
item.title = "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, then an `S3ClientCache` can help you keep the clients organized by 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 = new DynamoDBMapperConfig(
    DynamoDBMapperConfig.SaveBehavior.CLOBBER,
    DynamoDBMapperConfig.ConsistentReads.CONSISTENT,
    null, //TableNameOverride - leaving this at default setting
    DynamoDBMapperConfig.PaginationLoadingStrategy.EAGER_LOADING
);

DynamoDBMapper mapper = new DynamoDBMapper(client, mapperConfig, cp);
```

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`.

- 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 keep 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.
 - `CLOBBER`—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.

- 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 run time.

You can override the default configuration object for `DynamoDBMapper` per operation, as needed.

Example: 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 code example then uses the `DynamoDBMapper` to save a book object, retrieve it, update it and delete the book item.

Note

This code sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
package com.amazonaws.codesamples.datamodeling;  
  
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;  
  
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("611-111111111");
    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 = new
    DynamoDBMapperConfig(DynamoDBMapperConfig.ConsistentReads.CONSISTENT);
    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.");
        }
    }
}

```

Example: Batch Write Operations

The following Java code example declares Book, Forum, Thread, and Reply classes and maps them to the DynamoDB tables using the `DynamoDBMapper` class.

The code illustrate 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 Sample Data \(p. 280\)](#). For step-by-step instructions to test the following sample, see [Java Code Samples \(p. 285\)](#).

Example

```

// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
package com.amazonaws.codesamples.datamodeling;

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;

public class DynamoDBMapperBatchWriteExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    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.id = 901;
    book1.inPublication = true;
    book1.ISBN = "902-11-11-1111";
    book1.pageCount = 100;
    book1.price = 10;
    book1.productCategory = "Book";
    book1.title = "My book created in batch write";

    Book book2 = new Book();
    book2.id = 902;
    book2.inPublication = true;
    book2.ISBN = "902-11-12-1111";
    book2.pageCount = 200;
    book2.price = 20;
    book2.productCategory = "Book";
    book2.title = "My second book created in batch write";

    Book book3 = new Book();
    book3.id = 903;
    book3.inPublication = false;
    book3.ISBN = "902-11-13-1111";
    book3.pageCount = 300;
    book3.price = 25;
    book3.productCategory = "Book";
    book3.title = "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.name = "Test BatchWrite Forum";
    forumItem.threads = 0;
    forumItem.category = "Amazon Web Services";

    // Create Thread item to save
    Thread threadItem = new Thread();
```

```

threadItem.forumName = "AmazonDynamoDB";
threadItem.subject = "My sample question";
threadItem.message = "BatchWrite message";
List<String> tags = new ArrayList<String>();
tags.add("batch operations");
tags.add("write");
threadItem.tags = 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 = new
DynamoDBMapperConfig(DynamoDBMapperConfig.SaveBehavior.CLOBBER);
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;
    }
}
```

Example: Query and Scan

The Java example in this section defines the following classes and maps them to the tables in DynamoDB. For more information about creating sample tables, see [Creating Tables and Loading Sample Data \(p. 280\)](#).

- `Book` class maps to `ProductCatalog` table
- `Forum`, `Thread` and `Reply` classes maps to the same name tables.

The example then executes the follow 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.

- Execute 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 `ProductCatalog` table to find books whose price is less than a specified value.

For performance reasons, you should use `query` instead of the `scan` operation. However, there are times you might need to scan a table. Suppose 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`) and price is less than 0.

- Perform a parallel scan of the `ProductCatalog` table to find bicycles of a specific type.

Note

This code sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// Licensed under the Apache License, Version 2.0.
package com.amazonaws.codesamples.datamodeling;

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;

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;
            FindBicyclesOfSpecificTypeWithMultipleThreads(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, 101);
        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
    long endDateMilli = (new Date()).getTime() - (7L * 24L * 60L * 60L * 1000L); // One
    // week
    long ago = (new Date()).getTime() - (1L * 24L * 60L * 60L * 1000L); // One day
    // 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 FindBicyclesOfSpecificTypeWithMultipleThreads(DynamoDBMapper
mapper, int numberOfThreads,
String bicycleType) throws Exception {

    System.out.println("FindBicyclesOfSpecificTypeWithMultipleThreads: 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;
    }
}
}

```

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 DynamoDB. If you use this strategy, then 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, you simply try again by retrieving the item and then attempting 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.

To support optimistic locking, 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 will 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.

For example, the following Java code snippet 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 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 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.

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 \(p. 209\)](#).

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.

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 \(p. 197\)](#)), you can use types in your application for which there is no direct mapping to the 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 sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

Example

```
package com.amazonaws.codesamples.datamodeling;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import com.amazonaws.regions.Regions;
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.DynamoDBTypeConverted;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTypeConverter;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

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(data[0].trim());
                itemDimension.setHeight(data[1].trim());
                itemDimension.setThickness(data[2].trim());
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }

        return itemDimension;
    }
}
```

.NET: Document Model

Topics

- [Operations Not Supported by the Document Model \(p. 232\)](#)
- [Working with Items in DynamoDB Using the AWS SDK for .NET Document Model \(p. 232\)](#)
- [Getting an Item - Table.GetItem \(p. 236\)](#)
- [Deleting an Item - Table.DeleteItem \(p. 237\)](#)
- [Updating an Item - Table.UpdateItem \(p. 238\)](#)
- [Batch Write - Putting and Deleting Multiple Items \(p. 239\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Document Model \(p. 241\)](#)
- [Example: Batch Operations Using AWS SDK for .NET Document Model API \(p. 244\)](#)
- [Querying Tables in DynamoDB Using the AWS SDK for .NET Document Model \(p. 245\)](#)

The AWS SDK for .NET provides document model classes that wrap some of the low-level DynamoDB operations, to further simplify 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.

Operations Not Supported by the Document Model

You cannot use the document model classes to create, update, and delete tables. The document model does support most common data operations, however.

Working with Items in DynamoDB Using the AWS SDK for .NET Document Model

Topics

- [Putting an Item - Table.PutItem Method \(p. 233\)](#)
- [Specifying Optional Parameters \(p. 235\)](#)

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# snippet creates a `Table` object that represents the `ProductCatalog` table in 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 of these data operations have two types of overloads; one that takes the minimum required parameters and another that also takes operation specific optional 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 can also 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, the `GetItemOperationConfig` class enables you to provide options for the `GetItem` operation and the `PutItemOperationConfig` class enables you 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, then the `PutItem` operation replaces the entire

existing item. The new item will be identical to the `Document` object that you provided to the `PutItem` method. Note that this means that 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. Execute 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. Execute `Table.PutItem` by providing the `Document` instance as a parameter.

The following C# code snippet 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 snippet:

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);
item.Add("RelatedItems", relatedItems);

table.PutItem(book);
```

To add a Map attribute to the book, you define another Document. The following code snippet 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 \(p. 338\)](#). 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 snippet 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 snippet:

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 \(p. 15\)](#)) 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 snippet 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 \(p. 15\)](#).

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:

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 snippet 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 either pass the item's primary key as a parameter or if you have 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 snippet.

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 snippet 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 `ReturnValues` 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,
```

```
    ReturnValue = ReturnValue.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 you wish 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 mid-level `UpdateItem` operation does not support the `Add` action (see [UpdateItem](#)) supported by the underlying DynamoDB operation.

Note

The `PutItem` operation ([Putting an Item - Table.PutItem Method \(p. 233\)](#)) can also can perform an update. If you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. Note that, 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 will remain unchanged.

The following are the steps to update an item using the AWS SDK for .NET document model.

1. Execute 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 wish 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 snippet demonstrates the preceding tasks. The code sample updates an item in the Book table. The `UpdateItem` operation updates the existing `Authors` attribute, deletes the `PageCount` attribute, and adds a new attribute `XYZ`. 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 snippet 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;

UpdateOperationConfig config = new UpdateOperationConfig()
{
    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. Execute the `CreateBatchWrite` method on the table instance you created in the preceding step and create `DocumentBatchWrite` object.

3. Use `DocumentBatchWrite` object methods to specify documents you wish to upload or delete.
4. Call the `DocumentBatchWrite.Execute` method to execute the batch operation.

When using the document model API, you can specify any number of operations in a batch. However, note that 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 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 in to several smaller batches. Additionally, if a response to a batch write returns unprocessed items, the document model API will automatically send another batch request with those unprocessed items.

The following C# code snippet demonstrates the preceding steps. The code snippet 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 AWS SDK for .NET Document Model API \(p. 244\)](#).

You can use the batch write operation to perform put and delete operations on multiple tables. The following are the steps to put or delete multiple items from multiple table using the AWS SDK for .NET document model.

1. You create `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 in it.
3. Execute the `MultiTableDocumentBatchWrite.Execute` method.

The following C# code snippet demonstrates the preceding steps. The code snippet uses 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:

- Create a book item in the ProductCatalog table.
- Retrieve the book item.
- Update the book item. The code example shows a normal update that adds new attributes and updates existing attributes. It also shows a conditional update which updates the book price only if the existing price value is as specified in the code.
- Delete the book item.

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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.
    ReturnValues = ReturnValues.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.ToString()).ToArray());
        Console.WriteLine(attribute + ": " + stringValue);
    }
}

```

```
        select primitive.Value).ToArray());
    Console.WriteLine("{0} - {1}", attribute, stringValue);
}
}
}
```

Example: Batch Operations Using AWS SDK for .NET Document Model API

Topics

- [Example: Batch Write Using AWS SDK for .NET Document Model \(p. 244\)](#)

Example: Batch Write Using 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:

- To illustrate a single table batch write, it adds two items to the ProductCatalog table.
- To illustrate 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 Sample Data \(p. 280\)](#), 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 \(p. 756\)](#). For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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"] = "902-11-11-1111";
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();
}
}

```

Querying Tables in DynamoDB Using the AWS SDK for .NET Document Model

Topics

- Table.Query Method in the AWS SDK for .NET (p. 246)

- [Table.Scan Method in the AWS SDK for .NET \(p. 250\)](#)

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 \(p. 15\)](#).

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 snippet 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 snippet. 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 execute 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 snippet 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 execute the following sample queries:

- The following queries are executed against the Reply table.
 - Find forum thread replies that were posted in the last 15 days.

This query is executed 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 snippet 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. `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 can also 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 snippet executes the same preceding query (find forum threads in which the `ForumId` is 101 and the `Tag` attribute contains the "sortkey" keyword). However, this time 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 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 execute 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 (p. 255)
 - DynamoDBContext Class (p. 257)
 - Supported Data Types (p. 261)
 - Optimistic Locking Using Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 262)
 - Mapping Arbitrary Data with DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 263)
 - Batch Operations Using AWS SDK for .NET Object Persistence Model (p. 266)
 - Example: CRUD Operations Using the AWS SDK for .NET Object Persistence Model (p. 269)
 - Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model (p. 271)
 - Example: Query and Scan in DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 275)

The AWS SDK for .NET provides an object persistence model that enables you to map your client-side classes to the 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 execute 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 Tables: .NET \(p. 321\)](#).

To show you how the object persistence model works, let's walk through an example. We'll start with the ProductCatalog table. It has Id as the primary key.

```
ProductCatalog(Id, ...)
```

Suppose 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 snippet.

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 non-primary 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, then you must define the mapping by explicitly adding the `DynamoDBProperty` attribute.

In the preceding example, `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 can also 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 \(p. 263\)](#).

The object persistence model supports optimistic locking. During an update operation this ensures you have the latest copy of the item you are about to update. For more information, see [Optimistic Locking Using Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 262\)](#).

DynamoDB Attributes

This section describes the attributes the object persistence model offers so 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 examples 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 the same name table attribute, then 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), then 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 Sample Data \(p. 280\)](#).

```
[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 extends Developer { ... }
```

You can add the optional parameter, `LowerCamelCaseProperties`, to request DynamoDB to lower case the first letter of the property name when storing the objects to a table as shown in the following C# snippet.

```
[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. To more information about versioning, see [Optimistic Locking Using Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 262\)](#).

DynamoDBContext Class

The `DynamoDBContext` class is the entry point to the DynamoDB database. It provides a connection to DynamoDB and enables you to access your data in various tables, perform various CRUD operations, and execute 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 the table(s), 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 the table(s), 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 \(p. 268\)](#).

[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 \(p. 266\)](#).

[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 \(p. 232\)](#).

Suppose you have a `Document` object named `doc`, containing a representation of a Forum item. (To see how to construct this object, see the description for the `ToDocument` method below.) You can use `FromDocument` to retrieve the Forum item from the `Document` as shown in the following C# code snippet.

Example

```
forum101 = context.FromDocument<Forum>(101);
```

Note

If your `Document` object implements the `IEnumerable` interface, you can use the `FromDocuments` method instead. This will allow you to iterate over all of the class instances in the `Document`.

[FromQuery](#)

Executes 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](#)

Executes 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.

[GetTargetTable](#)

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 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 on the eventual consistency model, see [Read Consistency \(p. 15\)](#).

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 you have a client-side `Reply` class mapped to the `Reply` table in DynamoDB. The following C# code snippet 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 Sample Data \(p. 280\)](#).

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 only need to iterate over the `IEnumerable`.

If your table has a simple primary key (partition key), then you cannot 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 does not exist in the table, the method adds a new item to the table. If 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 Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 262\)](#).

Note

To perform this operation in the background, use the `SaveAsync` method instead.

[Scan](#)

Performs an entire table scan.

You can filter scan result by specifying a scan condition. The condition can be evaluated on any attributes in the table. Suppose you have a client-side class `Book` mapped to the `ProductCatalog` table in DynamoDB. The following C# snippet 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 \(p. 232\)](#).

Suppose 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 snippet.

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 optionally add this 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 directives to delete the specific attribute.
- **SkipVersionCheck**— - This parameter informs `DynamoDBContext` to not compare versions when saving or deleting an item. For more information about versioning, see [Optimistic Locking Using Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 262\)](#).
- **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# snippet 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 you send using this context.

Instead of setting these parameters at the DynamoDBContext level, you can specify them for individual operations you execute using DynamoDBContext as shown in the following C# code snippet. 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)
MemoryStream, byte[]	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.

Optimistic Locking Using Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model

The optimistic locking support in the object persistence model ensures that the item version for your application is same as the item version on the server-side before updating or deleting the item. Suppose 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 on 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 snippet 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, and then 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 `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 snippet.

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, then 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, the `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 you make using the context. For more information, see [Specifying Optional Parameters for DynamoDBContext \(p. 260\)](#).

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 snippet. The code example uses the context to delete a book item. The `Delete` method sets the optional `skipVersionCheck` property to true, disabling version check.

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 \(p. 261\)](#)), you can use types in your application for which there is no direct mapping to the 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", and when you retrieve a book, it converts the string to a `DimensionType` instance.

The example maps the `Book` type to the `ProductCatalog` table. For illustration, it saves a sample `Book` instance, retrieves it, updates its dimensions and saves the updated `Book` again.

For step-by-step instructions on how to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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 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:

- Execute `CreateBatchWrite` method of the `DynamoDBContext` and create an instance of the `BatchWrite` class.
 - Specify the items 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 either specify the primary key of the item or a client-side object that maps to the item 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 object persistence model, you can specify any number of operations in a batch. However, note that 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 your batch write request exceeded the allowed number of

write requests or exceeded the maximum allowed HTTP payload size, it breaks the batch in to several smaller batches. Additionally, if a response to a batch write returns unprocessed items, the API will automatically send another batch request with those unprocessed items.

Suppose that you have defined a C# class Book class that maps to the ProductCatalog table in DynamoDB. The following C# code snippet 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:
 - Execute 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.
 - Execute 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 have 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 snippet. The code snippet 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 execute 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 \(p. 271\)](#).

Note

DynamoDB batch API limits the number of writes in 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 size exceed 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 sample 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(devBatch.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:
 - Execute 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.
 - Execute 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 snippet 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 = orderDetailBatch.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. It uses the object persistence attributes to map these properties to the `ProductCatalog` table in DynamoDB. The code example then uses the `DynamoDBContext` to illustrate typical CRUD operations. The example creates a sample `Book` instance and saves it to the `ProductCatalog` table. The example then retrieves the book item, and updates its `ISBN` and `Authors` properties. Note that the update replaces the existing authors list. The example finally deletes the book item.

For more information about the `ProductCatalog` table used in this example, see [Creating Tables and Loading Sample Data \(p. 280\)](#). For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

Note

The following example does not work with .NET core as it does not 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 = "111-111111001",
                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 the DynamoDB tables using the object persistence model attributes.

The code 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 Sample Data \(p. 280\)](#). For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

Note

The following example does not work with .NET core as it does not 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 Sample Data \(p. 280\)](#).

- Book class maps to ProductCatalog table
 - Forum, Thread, and Reply classes maps to the same name tables.

The example then executes the following query and scan operations using the `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.

- Execute 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 ProductCatalog table to find books whose price is less than zero.

For performance reasons, you should use a query instead of a scan operation. However, there are times you might need to scan a table. Suppose 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 (ProductCategory is book) at price of less than 0.

For instructions to create a working sample, see [.NET Code Samples \(p. 287\)](#).

Note

The following example does not work with .NET core as 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 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
{
    // PK 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,
        // 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;
    }
}
```

Running the Code Samples In This Developer Guide

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 \(p. 52\)](#).

The code samples in this Developer Guide provide more in-depth coverage of DynamoDB operations, using the following programming languages:

- [Java Code Samples \(p. 285\)](#)
- [.NET Code Samples \(p. 287\)](#)

Before you can begin with this exercise, you will need to sign up for AWS, get your access key and secret key, and set up the AWS Command Line Interface on your computer. If you haven't done so, see [Setting Up DynamoDB \(Web Service\) \(p. 46\)](#).

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 \(p. 44\)](#).

Creating Tables and Loading Sample Data

In this section, you will use the AWS Management Console to create tables in DynamoDB. You will then load data into these tables using the AWS Command Line Interface (AWS CLI).

These tables and their data are used as examples throughout this Developer Guide.

Note

If you are an application developer, we recommend that you also read [Getting Started with DynamoDB](#), where you use the downloadable version of DynamoDB. This lets you learn about the DynamoDB low-level API for free, without having to pay any fees for throughput, storage, or data transfer. For more information, see [Getting Started with DynamoDB \(p. 52\)](#).

Topics

- [Step 1: Create Example Tables \(p. 281\)](#)
- [Step 2: Load Data into Tables \(p. 283\)](#)
- [Step 3: Query the Data \(p. 284\)](#)
- [Step 4: \(Optional\) Clean up \(p. 285\)](#)
- [Summary \(p. 285\)](#)

Step 1: Create Example Tables

In this section, you will use the AWS Management Console to create tables in DynamoDB for two simple use cases.

Use Case 1: Product Catalog

Suppose you want to store product information in DynamoDB. Each product has its own distinct attributes, so you will need to store different information about each of these products.

You can create a *ProductCatalog* table, where each item is uniquely identified by a single, numeric attribute: *Id*.

Table Name	Primary Key
<i>ProductCatalog</i>	Partition key: <i>Id</i> (Number)

Use Case 2: Forum Application

Suppose you want to build an application for message boards, or discussion forums. The Amazon Web Services [Discussion Forums](#) is one example of such an application: Customers can engage with the developer community, ask questions, or reply 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. Each thread might receive any number of replies.

You can model this application by creating three tables: *Forum*, *Thread*, and *Reply*.

Table Name	Primary Key
<i>Forum</i>	Partition key: <i>Name</i> (String)
<i>Thread</i>	Partition key: <i>ForumName</i> (String) Sort key: <i>Subject</i> (String)
<i>Reply</i>	Partition key: <i>Id</i> (String) Sort key: <i>ReplyDateTime</i> (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>	Partition key: <i>PostedBy</i> (String) Sort key: <i>Message</i> (String)

Create the ProductCatalog Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.

3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table name** field, type `ProductCatalog`.
 - b. For the **Primary key**, in the **Partition key** field, type `id`. Set the data type to **Number**.
4. When the settings are as you want them, choose **Create**.

Create the Forum Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table name** field, type `Forum`.
 - b. For the **Primary key**, in the **Partition key** field, type `name`. Set the data type to **String**.
4. When the settings are as you want them, choose **Create**.

Create the Thread Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table name** field, type `Thread`.
 - b. For the **Primary key**, do the following:
 - In the **Partition key** field, type `ForumName`. Set the data type to **String**.
 - Choose **Add sort key**.
 - In the **Sort key** field, type `Subject`. Set the data type to **String**.
4. When the settings are as you want them, choose **Create**.

Create the Reply Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table name** field, type `Reply`.
 - b. For the **Primary key**, do the following:
 - In the **Partition key** field, type `id`. Set the data type to **String**.
 - Choose **Add sort key**.
 - In the **Sort key** field, type `ReplyDateTime`. Set the data type to **String**.
 - c. In the **Table settings** section, deselect **Use default settings**.
 - d. In the **Secondary indexes** section, choose **Add index**.
 - e. In the **Add index** window, do the following:
 - For the **Primary key**, do the following:
 - In the **Partition key** field, type `PostedBy`. Set the data type to **String**.
 - Select **Add sort key**.
 - In the **Sort key** field, type `Message`. Set the data type to **String**.

- In the **Index name** field, type `PostedBy-Message-Index`.
 - Set the **Projected attributes** to **All**.
 - Choose **Add index**.
4. When the settings are as you want them, choose **Create**.

Step 2: Load Data into Tables

In this step, you will load sample data into the tables that you created. You could enter the data manually into the DynamoDB console; however, to save time, you will use the AWS Command Line Interface instead.

Note

If you have not yet set up the AWS CLI, see [Using the CLI \(p. 49\)](#) for instructions.

You will download a `.zip` archive that contains JSON files with sample data for each table. For each file, you will use the AWS CLI to load the data into DynamoDB. Each successful data load will produce the following output:

```
{  
    "UnprocessedItems": {}  
}
```

Download the Sample Data File Archive

1. Download the sample data archive (`sampledata.zip`) using this link:
 - [sampledata.zip](#)
2. Extract the `.json` data files from the archive.
3. Copy the `.json` data files to your current directory.

Load the Sample Data Into DynamoDB Tables

1. To load the *ProductCatalog* table with data, enter the following command:

```
aws dynamodb batch-write-item --request-items file://ProductCatalog.json
```

2. To load the *Forum* table with data, enter the following command:

```
aws dynamodb batch-write-item --request-items file://Forum.json
```

3. To load the *Thread* table with data, enter the following command:

```
aws dynamodb batch-write-item --request-items file://Thread.json
```

4. To load the *Reply* table with data, enter the following command:

```
aws dynamodb batch-write-item --request-items file://Reply.json
```

Verify Data Load

You can use the AWS Management Console to verify the data that you loaded into the tables.

To verify the data using the AWS Management Console

1. 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 *ProductCatalog*.
4. Choose the **Items** tab to view the data that you loaded into the table.
5. To view an item in the table, choose its Id. (If you want, you can also edit the item.)
6. To return to the list of tables, choose **Cancel**.

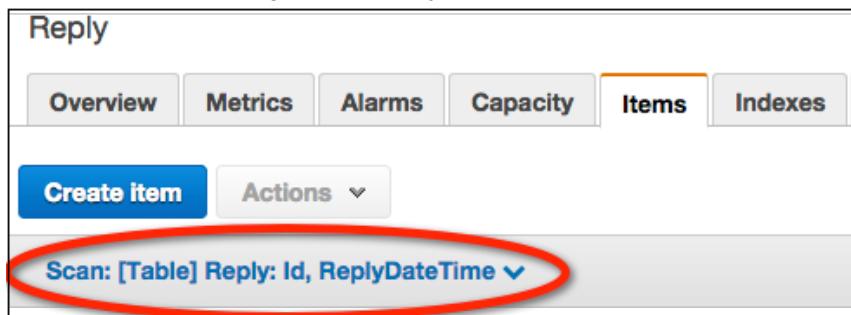
Repeat this procedure for each of the other tables you created:

- *Forum*
- *Thread*
- *Reply*

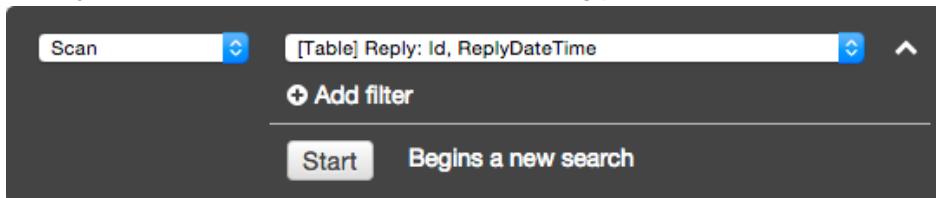
Step 3: Query the Data

In this step, you will try some simple queries against the tables that you created, using the DynamoDB console.

1. 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 *Reply*.
4. Choose the **Items** tab to view the data that you loaded into the table.
5. Choose the data filtering link, located just below the **Create item** button.



When you do this, the console reveals a data filtering pane.



6. In the data filtering pane, do the following:
 - a. Change the operation from **Scan** to **Query**.
 - b. For the **Partition key**, enter the value `Amazon.DynamoDB#DynamoDB Thread 1`.
 - c. Choose **Start**. Only the items that match your query criteria are returned from the *Reply* table.
7. The *Reply* table has a global secondary index on the `PostedBy` and `Message` attributes. You can use the data filtering pane to query the index. Do the following:
 - a. Change the query source from this:

[Table] Reply: Id, ReplyDateTime

to this:

[Index] PostedBy-Message-Index: PostedBy, Message

- b. For the **Partition key**, enter the value `User A`.
- c. Choose **Start**. Only the items that match your query criteria are returned from `PostedBy-Message-Index`.

Take some time to explore your other tables using the DynamoDB console:

- *ProductCatalog*
- *Forum*
- *Thread*

Step 4: (Optional) Clean up

This Amazon DynamoDB Developer Guide refers to these sample tables repeatedly, to help illustrate table and item operations using the DynamoDB low-level API and various AWS SDKs. You might find these tables useful for reference, if you plan to read the rest of this Developer Guide.

However, if you don't want to keep these tables, you should delete them to avoid being charged for resources you don't need.

To Delete the Sample Tables

1. 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 *ProductCatalog*.
4. Choose **Delete Table**. You will be asked to confirm your selection.

Repeat this procedure for each of the other tables you created:

- *Forum*
- *Thread*
- *Reply*

Summary

In this exercise, you used the DynamoDB console to create several tables in DynamoDB. You then used the AWS CLI to load data into the tables, and performed some basic operations on the data using the DynamoDB console.

The DynamoDB console and the AWS CLI are helpful for getting started quickly. However, you probably want to learn more about how DynamoDB works, and how to write application programs with DynamoDB. The rest of this Developer Guide addresses those topics.

Java Code Samples

Topics

- [Java: Setting Your AWS Credentials \(p. 286\)](#)
- [Java: Setting the AWS Region and Endpoint \(p. 287\)](#)

This Developer Guide contains Java code snippets and ready-to-run programs. You can find these code samples in the following sections:

- [Working with Items in DynamoDB \(p. 327\)](#)
- [Working with Tables in DynamoDB \(p. 290\)](#)
- [Working with Queries \(p. 410\)](#)
- [Working with Scans \(p. 427\)](#)
- [Improving Data Access with Secondary Indexes \(p. 446\)](#)
- [Java: DynamoDBMapper \(p. 195\)](#)
- [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#)

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 Samples (using Eclipse)

1. Download and install the [Eclipse](#) IDE.
2. Download and install the [AWS Toolkit for Eclipse](#).
3. Start Eclipse and from 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**, type 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**, type a name for your class in **Name** (use the same name as the code sample that you want to run), and then choose **Finish** to create the class.
9. Copy the code sample from the documentation page you are reading into the Eclipse editor.
10. To run the code, choose **Run** in 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 samples in this Developer 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 samples in this Developer 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 samples access DynamoDB in the US West (Oregon) region. You can change the region by modifying the `AmazonDynamoDB` properties.

The following code snippet 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 Amazon 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 samples using DynamoDB locally on your computer, you need to set the endpoint, like this:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
.build();
```

.NET Code Samples

Topics

- [.NET: Setting Your AWS Credentials \(p. 288\)](#)
- [.NET: Setting the AWS Region and Endpoint \(p. 289\)](#)

This Developer Guide contains .NET code snippets and ready-to-run programs. You can find these code samples in the following sections:

- [Working with Items in DynamoDB \(p. 327\)](#)
- [Working with Tables in DynamoDB \(p. 290\)](#)
- [Working with Queries \(p. 410\)](#)
- [Working with Scans \(p. 427\)](#)
- [Improving Data Access with Secondary Indexes \(p. 446\)](#)
- [.NET: Document Model \(p. 232\)](#)
- [.NET: Object Persistence Model \(p. 253\)](#)
- [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#)

You can get started quickly by using the AWS SDK for .NET with the Toolkit for Visual Studio.

To Run the .NET Code Samples (using Visual Studio)

1. Download and install [Microsoft Visual Studio](#).
 2. Download and install the [Toolkit for Visual Studio](#).
 3. Start Visual Studio and choose **File**, **New**, and then **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 sample from the documentation page you are reading 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, type `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 the **Start** button 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 samples in this Developer 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 samples in this Developer 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 samples 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 AWS region where Amazon 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 samples access DynamoDB in the US West (Oregon) region. You can change the region by modifying the `AWSRegion` entry in the `App.config` file. Alternatively, you can change the region by modifying the `AmazonDynamoDBClient` properties.

The following code snippet 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 samples using DynamoDB locally on your computer, you need to set the endpoint:

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// Set the endpoint URL
clientConfig.ServiceURL = "http://localhost:8000";
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

Working with DynamoDB

This chapter goes into details about the following topics:

Topics

- [Working with Tables in DynamoDB \(p. 290\)](#)
- [Working with Items in DynamoDB \(p. 327\)](#)
- [Working with Queries \(p. 410\)](#)
- [Working with Scans \(p. 427\)](#)
- [Improving Data Access with Secondary Indexes \(p. 446\)](#)
- [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#)

Working with Tables in DynamoDB

This section describes how to use the AWS CLI and the AWS SDKs to create, update, and delete tables.

Note

You can also perform these same tasks using the AWS Management Console. For more information, see [Using the Console \(p. 48\)](#)

This section also provides more information about throughput capacity, using DynamoDB auto scaling or manually setting provisioned throughput.

Topics

- [Basic Operations for Tables \(p. 291\)](#)
- [Throughput Settings for Reads and Writes \(p. 294\)](#)
- [Item Sizes and Capacity Unit Consumption \(p. 296\)](#)
- [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 298\)](#)
- [Tagging for DynamoDB \(p. 314\)](#)
- [Working with Tables: Java \(p. 316\)](#)
- [Working with Tables: .NET \(p. 321\)](#)

Basic Operations for Tables

Topics

- [Creating a Table \(p. 291\)](#)
- [Describing a Table \(p. 292\)](#)
- [Updating a Table \(p. 292\)](#)
- [Deleting a Table \(p. 293\)](#)
- [Listing Table Names \(p. 293\)](#)
- [Describing Provisioned Throughput Limits \(p. 293\)](#)

Creating a Table

Use the `CreateTable` operation to create a 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 EU (Ireland) - however, these two tables would be entirely different from each other. For more information, see [Naming Rules and Data Types \(p. 11\)](#).
- **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 \(p. 5\)](#).
- **Throughput settings.** 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 [Throughput Settings for Reads and Writes \(p. 294\)](#) and [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 298\)](#).

Example

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 below:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "BillingMode": "PAY_PER_REQUEST",
        "GlobalSecondaryIndexes": [],
        "HashKeyElement": {
            "AttributeName": "Artist",
            "AttributeType": "S"
        },
        "IndexCount": 0,
        "ItemCount": 0,
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "LastModified": "2018-01-12T23:00:00Z",
        "LocalSecondaryIndexes": [],
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 1024,
        "TableName": "Music"
    }
}
```

```
        "AttributeName": "SongTitle",
        "AttributeType": "S"
    }
],
"TableName": "Music",
"KeySchema": [
    {
        "AttributeName": "Artist",
        "KeyType": "HASH"
    },
    {
        "AttributeName": "SongTitle",
        "KeyType": "RANGE"
    }
],
"TableStatus": "CREATING",
"CreationDateTime": 1491338657.039,
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 5
},
"TableSizeBytes": 0,
"ItemCount": 0,
"TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music"
}
}
```

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 will require DynamoDB to allocate more resources for the table.

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.

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.

- Manipulate global secondary indexes on the table (see [Global Secondary Indexes \(p. 448\)](#)).
- Enable or disable DynamoDB Streams on the table (see [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#)).

Example

This 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`.

Deleting a Table

You can remove an unused table with the `DeleteTable` operation. Note that deleting a table is an unrecoverable operation.

This 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.

Listing Table Names

The `ListTables` operation returns the names of the DynamoDB tables for the current AWS account and region.

Example

```
aws dynamodb list-tables
```

Describing Provisioned Throughput Limits

The `DescribeLimits` operation returns the current read and write capacity limits for the current AWS account and region.

Example

```
aws dynamodb describe-limits
```

The output shows the upper limits of read and write capacity units for the current AWS account and region.

For more information about these limits, and how to request limit increases, see [Provisioned Throughput Default Limits \(p. 731\)](#).

Throughput Settings for Reads and Writes

Topics

- [Read Capacity Units \(p. 294\)](#)
- [Write Capacity Units \(p. 295\)](#)
- [Request Throttling and Burst Capacity \(p. 295\)](#)
- [Choosing Initial Throughput Settings \(p. 295\)](#)
- [Modifying Throughput Settings \(p. 296\)](#)

When you create a new table in DynamoDB, you must specify its *provisioned throughput capacity*—the amount of read and write activity that the table will be able to support. DynamoDB uses this information to reserve sufficient system resources to meet your throughput requirements.

Note

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 \(p. 298\)](#).

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 \(p. 18\)](#).) For optimal throughput, you should distribute read requests as evenly as possible across these partitions. For example, suppose that you provision a table with 10,000 read capacity units. If you issue 10,000 read requests for a single item in the table, all of the read activity will be concentrated on a single partition. However, if you spread your requests across all items in the table, DynamoDB can access the partitions in parallel.

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.

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 \(p. 15\)](#).

For example, suppose that you create a table with 10 provisioned read capacity units. This will allow 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 will consume the same throughput as reading a 4 KB item.

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 will allow you to perform 10 writes per second, for items up to 1 KB size per second.

Item sizes for writes are rounded up to the next 1 KB multiple. For example, writing a 500-byte item will consume the same throughput as writing a 1 KB item.

Request Throttling and Burst Capacity

If your application performs reads or writes at a higher rate than your table can support, DynamoDB will begin 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 \(p. 193\)](#).

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 will use *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. Burst capacity is available on a best-effort basis, and DynamoDB does not guarantee that this capacity is always available. For more information, see [Use Burst Capacity Sparingly \(p. 673\)](#).

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 will 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.)

Note

For recommendations on provisioned throughput and related topics, see [Best Practices for Tables \(p. 666\)](#).

Modifying Throughput Settings

If you have enabled DynamoDB auto scaling for a table, then its throughput capacity will be 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 . You can increase throughput capacity as often as needed, and decrease it up to nine times per table in a single UTC calendar day. For more information, see [Limits in DynamoDB \(p. 731\)](#).

Item Sizes and Capacity Unit Consumption

Topics

- [Capacity Unit Consumption for Reads \(p. 296\)](#)
- [Capacity Unit Consumption for Writes \(p. 297\)](#)

Before you choose read and write capacity settings for your table, you should first understand your data and how your application will access it. These inputs can help you determine your table's overall storage and throughput needs, and how much throughput capacity your application will require. 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. You can use the following guidelines to estimate attribute sizes:

- Strings are Unicode with UTF-8 binary encoding. The size of a string is $(\text{length of attribute name}) + (\text{number of UTF-8-encoded bytes})$.
- Numbers are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed. The size of a number is approximately $(\text{length of attribute name}) + (\text{1 byte per two significant digits}) + (\text{1 byte})$.
- 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 $(\text{length of attribute name}) + (\text{number of raw bytes})$.
- The size of a null attribute or a Boolean attribute is $(\text{length of attribute name}) + (\text{1 byte})$.
- An attribute of type List or Map requires 3 bytes of overhead, regardless of its contents. The size of a List or Map is $(\text{length of attribute name}) + \text{sum}(\text{size of nested elements}) + (\text{3 bytes})$. The size of an empty List or Map is $(\text{length of attribute name}) + (\text{3 bytes})$.

Note

We recommend that you choose shorter attribute names rather than long ones. This will help you optimize capacity unit consumption and reduce the amount of storage required for your data.

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 `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), take this number and divide it 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 will calculate the size as 12 KB (4 KB + 8 KB), not 8 KB (1.5 KB + 6.5 KB).
- `Query`—reads multiple items that have the same partition key value. All of the 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 of the items in a table. DynamoDB considers the size of the items that are evaluated, not the size of the items returned by the scan.

If you perform a read operation on an item that does not exist, DynamoDB will still consume 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, 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 total item size is 80 KB, the operation consumes only 10 capacity units.

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 will calculate 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 will still consume write capacity units from the table:

- If the item exists, the number of write capacity units consumed depends on the size of the item. (For example, a failed conditional write of a 1 KB item would consume one write capacity unit; if the item were twice that size, the failed conditional write would consume two write capacity units.)
- If the item does not exist, DynamoDB will consume one write capacity unit.

Managing Throughput Capacity Automatically with DynamoDB Auto Scaling

Note

To get started quickly with DynamoDB auto scaling, see [Using the AWS Management Console With DynamoDB Auto Scaling \(p. 300\)](#).

Many database workloads are cyclical in nature or are difficult to predict in advance. For 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. Another example might be a new mobile gaming app that is experiencing 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.

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 \(p. 300\)](#).

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.

Note

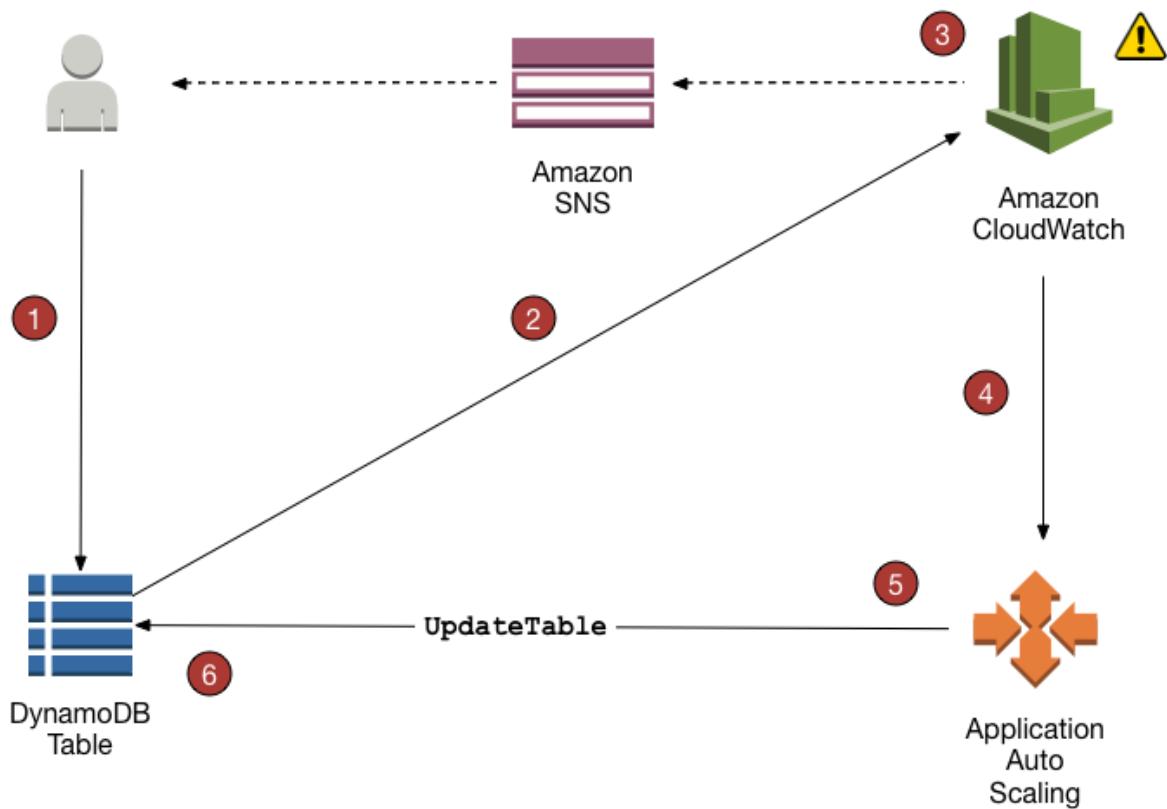
In addition to tables, DynamoDB auto scaling also supports global secondary indexes. Every global secondary index has its own provisioned throughput capacity, separate from that of its base table. When you create a scaling policy for a global secondary index, Application Auto Scaling adjusts the provisioned 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 \(p. 300\)](#).

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 AWS Management 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.

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 [Use Burst Capacity Sparingly \(p. 673\)](#).

To enable DynamoDB auto scaling for the *ProductCatalog* table, you create a scaling policy. This policy specifies 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, and 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 your upper and lower boundaries for 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.

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. You can decrease the `ReadCapacityUnits` or `WriteCapacityUnits` settings for a table up to four times any time per day. A day is defined according to the GMT time zone. Additionally, if there was no decrease in the past four hours, an additional dial down is allowed, effectively bringing maximum number of decreases in a day to nine times (4 decreases in the first 4 hours, and 1 decrease for each of the subsequent 4 hour windows in a day). All other DynamoDB limits remain in effect, as described in [Limits in 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 make manual adjustments, we recommend that you refresh your Application Auto Scaling policy so that the CloudWatch thresholds are updated accordingly.
- 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. You can do this by choosing **Apply same settings to global secondary indexes** in the AWS Management Console. For more information, see [Enabling DynamoDB Auto Scaling on Existing Tables \(p. 302\)](#).

Using the AWS Management Console With DynamoDB Auto Scaling

Topics

- [Before You Begin: Grant User Permissions for DynamoDB Auto Scaling \(p. 301\)](#)
- [Creating a New Table with Auto Scaling Enabled \(p. 302\)](#)
- [Enabling DynamoDB Auto Scaling on Existing Tables \(p. 302\)](#)

- [Viewing Auto Scaling Activities in the Console \(p. 303\)](#)
- [Modifying or Disabling DynamoDB Auto Scaling Settings \(p. 303\)](#)

When you use the AWS Management Console to create a new table, 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.

Before You Begin: Grant 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, IAM users will require some additional privileges.

Important

`application-autoscaling:*` permissions are required to delete an autoscaling enabled table. The AWS-managed policy `DynamoDBFullAccess` attached next includes such permissions.

To set up an IAM user for DynamoDB console access and DynamoDB auto scaling, add the following two policies:

To attach the `AmazonDynamoDBFullAccess` policy

1. Open the IAM console at <https://console.aws.amazon.com/iam/>.
2. On the IAM console dashboard, choose **Users**, and then choose your IAM user from the list.
3. On the **Summary** page, choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. From the list of policies, choose **AmazonDynamoDBFullAccess**, and then choose **Next: Review**.
6. Choose **Add permissions**.

To add a custom inline policy

1. On the IAM console dashboard, choose **Users**, and then choose your IAM user from the list.
2. On the **Summary** page, choose **Add inline policy**.
3. On the **Set Permissions** page, choose **Custom Policy**, and then choose **Select**.
4. On the **Review Policy** page, do the following:
 - a. **Policy Name**—type a name for the policy. For example: *MyDynamoDBAutoscalingPolicy*
 - b. **Policy Document**—copy and paste the following:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iam:AttachRolePolicy",  
                "iam>CreatePolicy",  
                "iam>CreateRole"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

5. After you have done this, choose **Apply Policy**.

Creating a New Table with Auto Scaling Enabled

Note

DynamoDB auto scaling requires the presence of a role (*DynamoDBAutoscaleRole*) that performs auto scaling actions on your behalf. You do not need to create this role yourself. Instead, the DynamoDB console creates the role automatically, when you create a new table with auto scaling for the first time.

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 DynamoDB table** page, enter a **Table name** and **Primary key** details.
4. If you see the following message, then your AWS account does not currently have the *DynamoDBAutoscaleRole*:

 **You do not have the required role to enable Auto Scaling by default.**
Please refer to [documentation](#).

In this case, do the following:

1. Deselect **Use default settings**.
2. In the **Auto scaling** section, ensure that **New role: DynamoDBAutoscaleRole** is selected.

Otherwise, simply ensure that **Use default settings** is selected. (Your AWS account already has *DynamoDBAutoscaleRole*.)

5. When the settings are as you want them, choose **Create**. Your table will be created with default auto scaling parameters.

Enabling DynamoDB Auto Scaling on Existing Tables

Note

DynamoDB auto scaling requires the presence of a role (*DynamoDBAutoscaleRole*) that performs auto scaling actions on your behalf. The DynamoDB console will create the role for you when you create a new table with auto scaling for the first time.

If you have never used DynamoDB auto scaling before, see [Creating a New Table with Auto Scaling Enabled \(p. 302\)](#). The DynamoDB console creates *DynamoDBAutoscaleRole* automatically, when you create a new table with auto scaling for the first time.

To enable DynamoDB auto scaling for an existing table, do the following:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose the table that you want to work with, and then choose **Capacity**.
3. In the **Auto Scaling** section, do the following:
 - Select **Read capacity**, **Write capacity**, or both. (For write capacity, note that you can choose **Same settings as read**.) For each of these, do the following:
 - **Target utilization**—Type your target utilization percentage for the table.
 - **Minimum provisioned capacity**—Type your lower boundary for the auto scaling range.
 - **Maximum provisioned capacity**—Type your upper boundary for the auto scaling range.
 - **Apply same settings to global secondary indexes**—Leave this option at its default setting (enabled).

Note

For best performance, we recommend that you enable **Apply same settings to 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.

(For **Write capacity**, note that you can choose **Same settings as read**.)

In the **IAM Role** section, ensure that **Existing role with pre-defined policies** is selected, and that **Role Name** is set to **DynamoDBAutoscaleRole**.

4. When the settings are as you want them, choose **Save**.

Viewing Auto Scaling Activities in the Console

As your application drives read and write traffic to your table, DynamoDB auto scaling dynamically modifies the table's throughput settings.

To view these auto scaling activities in the DynamoDB console, choose the table that you want to work with. Choose **Capacity**, and then expand the **Scaling activities** section. When your table's throughput settings are modified, you will see informational messages here.

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 **Capacity** tab for your table and modify the settings in the **Auto Scaling** section. For more information about these settings, see [Enabling DynamoDB Auto Scaling on Existing Tables \(p. 302\)](#).

To disable DynamoDB auto scaling, go to the **Capacity** tab for your table and clear **Read capacity**, **Write capacity**, or both.

Using the AWS CLI to Manage DynamoDB Auto Scaling

Topics

- [Before You Begin \(p. 304\)](#)
- [Step 1: Create a Service Role for Application Auto Scaling \(p. 304\)](#)
- [Step 2: Create a DynamoDB Table \(p. 305\)](#)
- [Step 3: Register a Scalable Target \(p. 306\)](#)
- [Step 4: Create a Scaling Policy \(p. 306\)](#)
- [Step 5: Drive Write Traffic to TestTable \(p. 308\)](#)
- [Step 6: View Application Auto Scaling Actions \(p. 308\)](#)
- [\(Optional\) Step 7: Clean Up \(p. 309\)](#)

Instead of using the AWS Management Console, you can use the AWS Command Line Interface (AWS CLI) to manage 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, so that the 50 percent target utilization can be maintained.
- Verify that DynamoDB has successfully adjusted the provisioned write capacity for *TestTable*.

Before You Begin

You need to 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, go to the AWS Command Line Interface User Guide and follow these instructions:

- [Installing the AWS CLI](#)
- [Configuring the AWS CLI](#)

Install Python

Part of this tutorial requires you to run a Python program (see [Step 5: Drive Write Traffic to *TestTable* \(p. 308\)](#)). If you don't already have Python installed, you can download it using this link: <https://www.python.org/downloads>

Step 1: Create a Service Role for Application Auto Scaling

Before you can work with Application Auto Scaling, you must create a service role for it. A *service role* is an IAM role that authorizes an AWS service to act on your behalf. The service role allows Application Auto Scaling to modify the provisioned throughput settings for your DynamoDB resources, as if you were modifying them yourself.

In this step, you create an IAM policy and attach that policy to an IAM role. You can then assign the role to Application Auto Scaling so that it can perform DynamoDB operations on your behalf.

1. Create a file named `trust-relationship.json` with the following contents:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "application-autoscaling.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

2. Create the service role:

```
aws iam create-role \  
    --role-name MyIAMAutoscalingServiceRole \  
    --assume-role-policy-document file://trust-relationship.json
```

3. Create a file named `service-role-policy.json` with the following contents:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeTable",  
                "dynamodb:UpdateTable",  
                "cloudwatch:PutMetricAlarm",  
                "cloudwatch:DescribeAlarms",  
                "cloudwatch:DeleteAlarms"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

4. Create an IAM policy for the service role:

```
aws iam create-policy \  
    --policy-name MyIAMAutoscalingServicePolicy \  
    --policy-document file://service-role-policy.json
```

In the output, take note of the Amazon Resource Name (ARN) for the policy you created. For example: `arn:aws:iam::111122223333:policy/MyIAMAutoscalingServicePolicy`

5. Attach the policy to the service role:

```
aws iam attach-role-policy \  
    --role-name MyIAMAutoscalingServiceRole \  
    --policy-arn arn
```

Replace `arn` with the actual policy ARN from the previous step.

Step 2: 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 3: Register a Scalable Target

You will now 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 to 10 capacity units.

1. When you register the scalable target, you specify the ARN for *MyIAMAutoscalingServiceRole*, which you created in [Step 1: Create a Service Role for Application Auto Scaling \(p. 304\)](#). Type the following command to retrieve the role ARN:

```
aws iam get-role \  
  --role-name MyIAMAutoscalingServiceRole \  
  --query "Role.Arn"
```

Now type the following command to register the scalable target. (Replace *roleARN* with the ARN for *MyIAMAutoscalingServiceRole*.)

```
aws application-autoscaling register-scalable-target \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable" \  
  --scalable-dimension "dynamodb:table:WriteCapacityUnits" \  
  --min-capacity 5 \  
  --max-capacity 10 \  
  --role-arn roleARN
```

2. To verify the registration, use the following command:

```
aws application-autoscaling describe-scalable-targets \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable"
```

Step 4: 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 to 10 capacity units (see [Step 3: Register a Scalable Target \(p. 306\)](#)) 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 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 3: Register a Scalable Target \(p. 306\)](#) and [Step 4: Create a Scaling Policy \(p. 306\)](#).

Step 5: Drive Write Traffic to *TestTable*

Now you can test your scaling policy by writing data to *TestTable*. To do this, you will 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. Type 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 6: 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. Type 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 will take 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. Type 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 7: 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
```

4. To delete the IAM policy, you must first determine its ARN:

```
aws iam list-policies --scope Local
```

In the output, take note of the ARN for *MyIAMAutoscalingServicePolicy*. For example:

arn:aws:iam::111122223333:policy/MyIAMAutoscalingServicePolicy

5. Detach the policy from the service role:

```
aws iam detach-role-policy \
--role-name MyIAMAutoscalingServiceRole \
--policy-arn arn
```

Replace *arn* with the actual policy ARN from the previous step.

You can now delete the policy:

```
aws iam delete-policy \
--policy-arn arn
```

Finally, delete the service role:

```
aws iam delete-role \
--role-name MyIAMAutoscalingServiceRole
```

Application Programming With DynamoDB Auto Scaling

In addition to using the AWS Management Console and AWS CLI, you can write applications that interact with 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.

The program requires that you supply an ARN for a valid Application Auto Scaling service role. (For example: "arn:aws:iam::122517410325:role/MyIAMAutoscalingServiceRole.) In the following program, replace `SERVICE_ROLE_ARN_Goes_Here` with the actual ARN. For more information, see [Step 1: Create a Service Role for Application Auto Scaling \(p. 304\)](#).

```

package com.amazonaws.codesamples.autoscaling;

import com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
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 = new
AWSApplicationAutoScalingClient();

    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());
}
}
```

Tagging for DynamoDB

You can label DynamoDB resources with tags. Tags allow you to categorize your resources in different ways, for example, by purpose, owner, environment, or other criteria. Tags can help you:

- Quickly identify a resource based on the tags you've 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 allowing you to create reports across services that carry a specific tag.

To get started with tagging, you should:

1. Understand [Tagging Restrictions \(p. 314\)](#).
2. Create tags by using [Tagging Operations \(p. 315\)](#).
3. Use [Cost Allocation Reports \(p. 315\)](#) 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

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 will be updated to the new value.
- Tag keys and values are case sensitive.
- Maximum key length: 128 Unicode characters
- Maximum value length: 256 Unicode characters
- Allowed characters are letters, whitespace, and numbers, plus the following special characters: + - = . _ : /
- Maximum number of tags per resource: 50
- AWS-assigned tag names and values are automatically assigned the aws: prefix, which you cannot assign. AWS-assigned tag names do not count toward the tag limit of 50. User-assigned tag names have the prefix user: in the cost allocation report.

- You cannot tag a resource at the same time you create it. Tagging is a separate action that can be performed only after the resource is created.
- You cannot backdate the application of a tag.

Tagging Operations

This section describes how to use the DynamoDB console or CLI to add, list, edit, or delete tags. You can then activate these user-defined tags so that they appear on the Billing and Cost Management console for cost allocation tracking. For more information, see [Cost Allocation Reports \(p. 315\)](#).

For bulk editing, you can also use the Tag Editor in the AWS Management Console. For more information, see [Working with Tag Editor](#).

To use the API instead, see [Amazon DynamoDB API Reference](#).

Topics

- [Tagging \(console\) \(p. 315\)](#)
- [Tagging \(CLI\) \(p. 315\)](#)

Tagging (console)

To use the console to add, list, edit, or delete tags:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>.
2. Choose a table, and then choose the **Settings** tab.

You can add, list, edit, or delete tags here. In this example, the `Movies` tag was created with a value of `moviesProd` for the `Movies` table.

Tagging (CLI)

To add 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 of the tags 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:

1. Sign in to the AWS Management Console and open the Billing and Cost Management console at <https://console.aws.amazon.com/billing/home#/>.
2. In the navigation pane, choose **Cost Allocation Tags**.
3. Under **AWS-Generated Cost Allocation Tags**, choose **Activate**.

To activate user-defined tags:

1. Sign in to the AWS Management Console and open the Billing and Cost Management console at <https://console.aws.amazon.com/billing/home#/>.
2. In the navigation pane, choose **Cost Allocation Tags**.
3. 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 Tables: Java

Topics

- [Creating a Table \(p. 316\)](#)
- [Updating a Table \(p. 317\)](#)
- [Deleting a Table \(p. 318\)](#)
- [Listing Tables \(p. 318\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for Java Document API \(p. 319\)](#)

You can use the AWS SDK for Java 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 table operations 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. Execute the `createTable` method by providing the request object as a parameter.

The following code snippet 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 will not be 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

You can increase throughput capacity as often as needed, and decrease it up to nine times per table in a single UTC calendar day. For more information, see [Limits in DynamoDB \(p. 731\)](#).

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. Execute the `updateTable` method by providing the `ProvisionedThroughput` instance as a parameter.

The following code snippet 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:

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. Execute the `deleteTable` method by providing the `Table` instance as a parameter.

The following code snippet 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 execute 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 sample uses the AWS SDK for Java Document API to create, update, and delete a 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 Samples \(p. 285\)](#).

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
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(  
                    .withReadCapacityUnits(5).withWriteCapacityUnits(5));  
            dynamoDB.createTable(request);  
        } catch (Exception e) {  
            System.out.println("An error occurred while creating the table: " + e.getMessage());  
        }  
    }  
  
    static void listMyTables() {  
        ListTablesResult result = dynamoDB.listTables();  
        System.out.println("List of tables: " + result.getTables());  
    }  
  
    static void getTableInformation() {  
        TableDescription desc = dynamoDB.getTable(tableName).get();  
        System.out.println("Table information: " + desc);  
    }  
  
    static void updateExampleTable() {  
        ProvisionedThroughput update = new  
ProvisionedThroughput().withReadCapacityUnits(10).withWriteCapacityUnits(10);  
        dynamoDB.updateTable(tableName, update);  
    }  
  
    static void deleteExampleTable() {  
        dynamoDB.deleteTable(tableName);  
    }  
}
```

```

        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 Tables: .NET

Topics

- [Creating a Table \(p. 321\)](#)
- [Updating a Table \(p. 323\)](#)
- [Deleting a Table \(p. 323\)](#)
- [Listing Tables \(p. 324\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for .NET Low-Level API \(p. 324\)](#)

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 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. Execute the appropriate method provided by the client that you created in the preceding step.

Note

The examples in this section do not work with .NET core as it does not support synchronous methods. For more information, see [AWS Asynchronous APIs for .NET](#).

Creating a Table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values.

The following are the steps to create a table 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, primary key, and the provisioned throughput values.

3. Execute the `AmazonDynamoDBClient.CreateTable` method by providing the request object as a parameter.

The following C# code snippet 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 the table 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 anytime.

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 up to nine times per table in a single UTC calendar day. For more information, see [Limits in DynamoDB \(p. 731\)](#).

The following are the steps to update a table using the .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. Execute the `AmazonDynamoDBClient.UpdateTable` method by providing the request object as a parameter.

The following C# code snippet 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

The following are the steps to delete a table using the .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. Execute the `AmazonDynamoDBClient.DeleteTable` method by providing the request object as a parameter.

The following C# code snippet 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 execute 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# code snippet. Along with the page size, the request sets the `ExclusiveStartTableName` parameter. Initially, `ExclusiveStartTableName` is null, however, after fetching the first page of result, to retrieve the next page of result, 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 sample, see [.NET Code Samples \(p. 287\)](#).

```
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");
}
}
```

Working with Items in DynamoDB

Topics

- [Reading an Item \(p. 328\)](#)
- [Writing an Item \(p. 329\)](#)
- [Return Values \(p. 331\)](#)
- [Batch Operations \(p. 331\)](#)
- [Atomic Counters \(p. 333\)](#)
- [Conditional Writes \(p. 334\)](#)
- [Using Expressions in DynamoDB \(p. 338\)](#)
- [Time To Live \(p. 362\)](#)
- [Working with Items: Java \(p. 368\)](#)
- [Working with Items: .NET \(p. 387\)](#)

In 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 \(p. 2\)](#).

DynamoDB provides four operations for basic create/read/update/delete (CRUD) functionality:

- `PutItem` – create an item.
- `GetItem` – read an item.
- `UpdateItem` – update an item.
- `DeleteItem` – delete an item.

Each of these operations require you to specify the primary key of the item 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. For best practices, see [Best Practices for Items \(p. 676\)](#).

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.

`GetItem` request performs an eventually consistent read, by default. You can use the `ConsistentRead` parameter to request a strongly consistent read instead. (This will consume additional read capacity units, but it will return 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 \(p. 341\)](#).)

To return the number of read capacity units consumed by `GetItem`, set the `ReturnConsumedCapacity` parameter to `TOTAL`.

Example

The following 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 need to 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.

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 file `item.json`:

```
{  
    "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 you want to modify and their new values. (For more information, see [Update Expressions \(p. 354\)](#).) Within the update expression, you use expression attribute values as placeholders for the actual values. (For more information, see [Expression Attribute Values \(p. 345\)](#).)

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 \(p. 331\)](#).)

```
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 file `key.json`:

```
{  
    "ForumName": {"S": "Amazon DynamoDB"},  
    "Subject": {"S": "New discussion thread"}  
}
```

The arguments for `--expression-attribute-values` are stored in the file `expression-attribute-values.json`:

```
{  
    ":zero": {"N":"0"},  
    ":lastpostedby": {"S":"barney@example.com"}  
}
```

DeleteItem

`DeleteItem` deletes the item with the specified key.

Example

This 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 will 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 will automatically create the item if it does not 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` does not 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* of the requests in the batch fail. For example, suppose 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 file `request-items.json`:

```
{
    "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 file `request-items.json`:

```
{
    "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 will increment each time you call `UpdateItem`.

You might use an atomic counter to keep track of 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 should fail, 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 over- or undercounting cannot be tolerated (For example, in a banking application). In these case, it is safer to use a conditional update instead of an atomic counter.

For more information, see [Incrementing and Decrementing Numeric Attributes \(p. 358\)](#).

Example

The following AWS CLI example increments the `Price` of a product by 5. (Because `UpdateItem` is not idempotent, the `Price` will increase every time you run this example.)

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id": { "N": "601" }}' \
--update-expression "SET Price = Price + :incr" \
--expression-attribute-values '{":incr":{"N":"5"}}' \
```

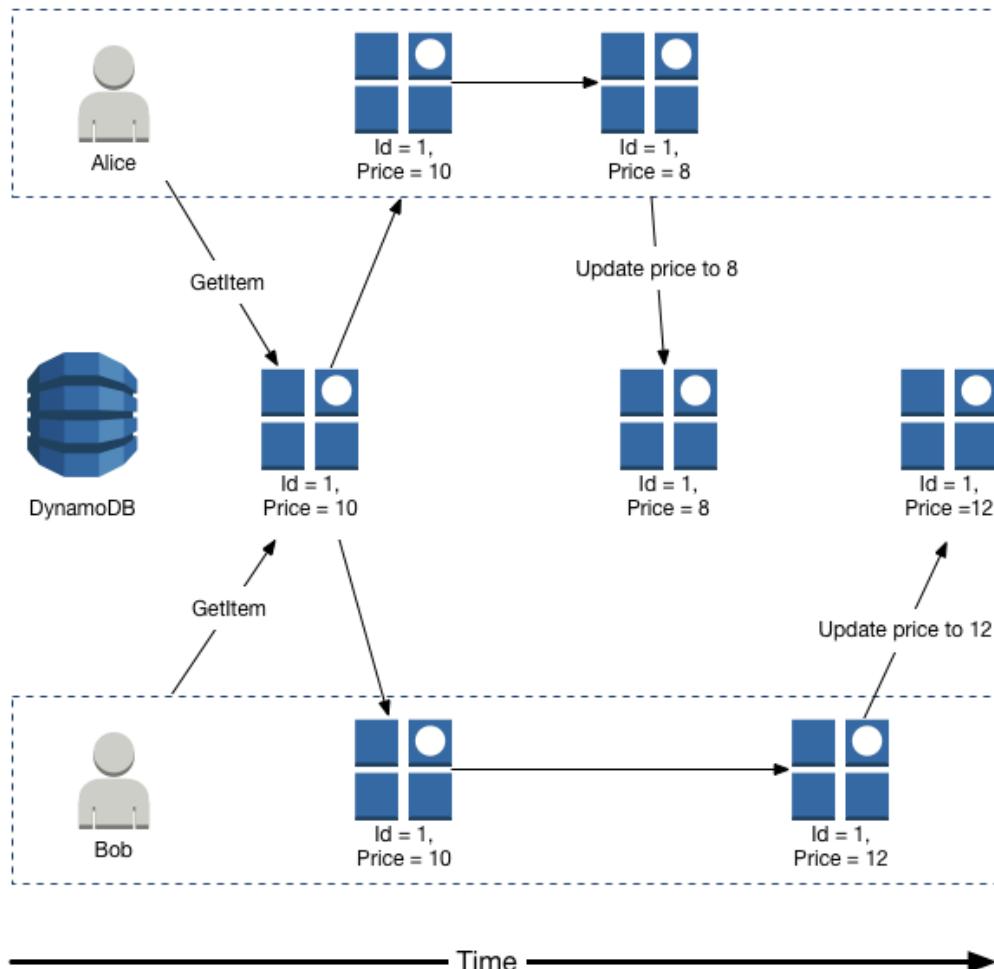
```
--return-values UPDATED_NEW
```

Conditional Writes

By default, the DynamoDB write operations (`PutItem`, `UpdateItem`, `DeleteItem`) are *unconditional*: each of these operations will overwrite an existing item that has the specified primary key.

DynamoDB optionally supports conditional writes for these operations. A conditional write will succeed only if the item attributes meet one or more expected conditions. Otherwise, it returns an error. 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"}  
}
```

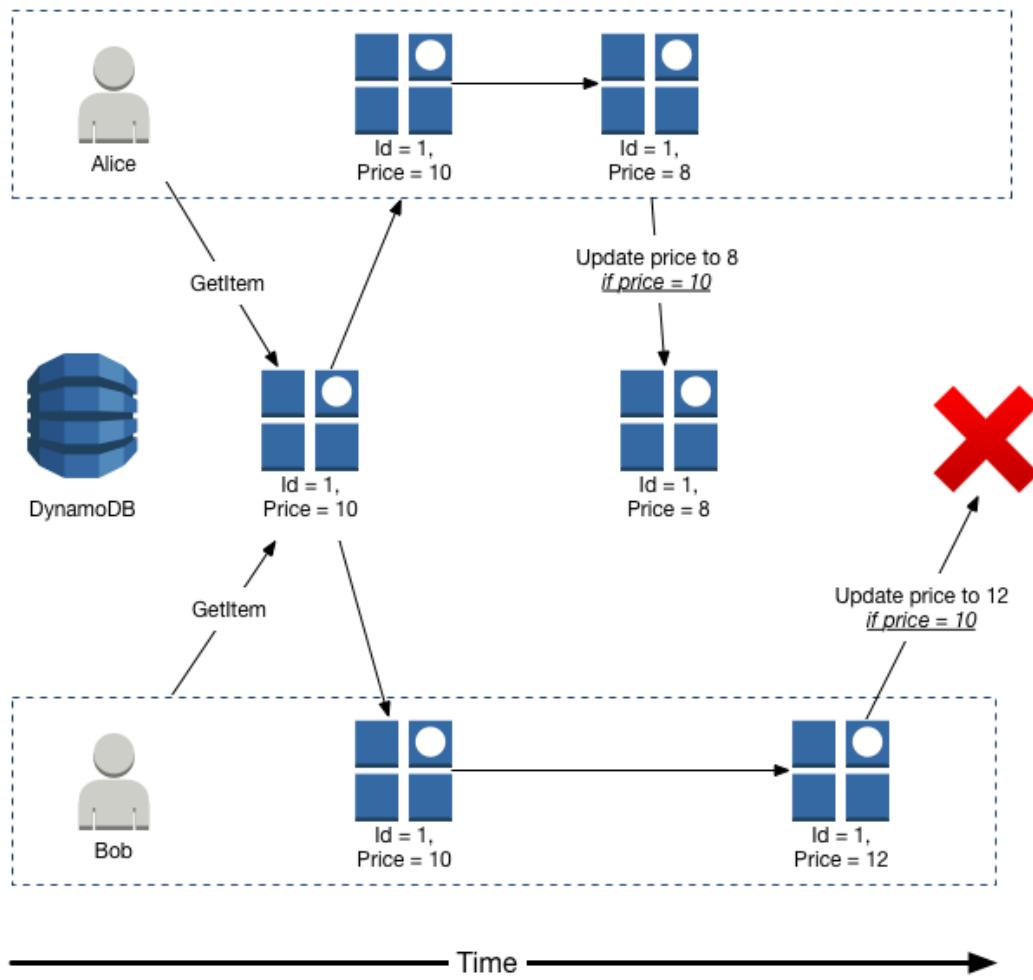
Now suppose that Bob issues a similar `UpdateItem` request later, but changes the `Price` to 12. For Bob, the `--expression-attribute-values` parameter looks like this:

```
{  
    ":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 will fail.

Now consider the following diagram, showing how conditional writes would prevent Alice's update from being overwritten:



Alice first attempts 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 file `expression-attribute-values.json`:

```
{
  ":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 this:

```
{  
    ":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 \(p. 346\)](#).

Conditional Write Idempotence

Conditional writes are *idempotent*. This means that you can send the same conditional write request to DynamoDB multiple times, but the request will have no further effect on the item after the first time DynamoDB performs the update.

For example, suppose 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 conditional writes are idempotent, you can retry the same `UpdateItem` request, and DynamoDB will update 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 will still consume write capacity from the table:

- If the item does not currently exist in the table, DynamoDB will consume one write capacity unit.
- If the item does exist, then the number of write capacity units consumed depends on the size of the item. For example, a failed conditional write of a 1 KB item would consume one write capacity unit. If the item were twice that size, the failed conditional write would consume two write capacity units.

Note

Write operations consume write capacity units only. They never consume read capacity units.

A failed conditional write will return a `ConditionalCheckFailedException`. When this occurs, you will not 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 \(p. 644\)](#) in [Monitoring DynamoDB \(p. 642\)](#).)

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 \(p. 789\)](#).

New applications should use expressions rather than the legacy parameters.

Topics

- [Specifying Item Attributes \(p. 338\)](#)
- [Projection Expressions \(p. 341\)](#)
- [Expression Attribute Names \(p. 342\)](#)
- [Expression Attribute Values \(p. 345\)](#)
- [Condition Expressions \(p. 346\)](#)
- [Update Expressions \(p. 354\)](#)

Specifying Item Attributes

This section describes how to refer to item attributes in an expression. You can work with any attribute, even if it is deeply nested within multiple lists and maps.

Topics

- [Top-Level Attributes \(p. 340\)](#)
- [Nested Attributes \(p. 340\)](#)
- [Document Paths \(p. 341\)](#)

A Sample Item: *ProductCatalog*

In this section, we will consider an item in the *ProductCatalog* table. (This table is described in [Example Tables and Data \(p. 739\)](#).) Here is a representation of the item:

ProductCatalog

```
{  
    "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 will be 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:

- `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 twelfth element in that list.

The number within the square brackets must be a non-negative integer. Therefore, the following expressions are invalid:

- `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 \(p. 338\)](#).)

- A top-level scalar attribute:

`ProductDescription`

- A top-level list attribute. (This will return 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, 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 does not meet this requirement, you will need to define an expression attribute name as a placeholder. For more information, see [Expression Attribute Names \(p. 342\)](#).

Projection Expressions

To read data from a table, you use operations such as `GetItem`, `Query`, or `Scan`. DynamoDB returns all of the item attributes by default. To get just some, rather than all of the attributes, use a projection expression.

A *projection expression* is a string that identifies the attributes 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 \(p. 338\)](#):

- A single top-level attribute.

`Title`

- Three top-level attributes. DynamoDB will retrieve the entire `Color` set.

`Title, Price, Color`

- Four top-level attributes. DynamoDB will return the entire contents of `RelatedItems` and `ProductReviews`.

`Title, Description, RelatedItems, ProductReviews`

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 does not meet this requirement, you will need to define an expression attribute name as a placeholder. For more information, see [Expression Attribute Names \(p. 342\)](#).

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 arguments for `--key` are stored in the file `key.json`:

```
{  
    "Id": { "N": "123" }  
}
```

For programming language-specific code samples, see [Getting Started with DynamoDB \(p. 52\)](#).

Expression Attribute Names

An *expression attribute name* is a placeholder that you use in an expression, as an alternative to an actual attribute name. An expression attribute name must begin with a `#`, and be followed by one or more alphanumeric characters.

This section describes several situations in which you will need to use expression attribute names.

Note

The examples in this section use the AWS CLI. For programming language-specific code samples, see [Getting Started with DynamoDB \(p. 52\)](#).

Topics

- [Reserved Words \(p. 343\)](#)
- [Attribute Names Containing Dots \(p. 343\)](#)
- [Nested Attributes \(p. 344\)](#)
- [Repeating Attribute Names \(p. 344\)](#)

Reserved Words

On some occasions, 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 \(p. 780\)](#).)

For example, the following AWS CLI example would fail because `COMMENT` is a reserved word:

```
aws dynamodb get-item \
--table-name ProductCatalog \
--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 this:

```
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 or contains a space, a special character, or a reserved word, then you *must* use an expression attribute name to replace that attribute's name in the expression.

Attribute Names Containing Dots

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 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 \(p. 338\)](#)):

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 would need to 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.

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 value 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.#1s` 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 this:

- `#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* 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 `:`, and be followed by one or more alphanumeric characters.

For example, suppose 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 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 file `values.json`:

```
{  
    ":c": { "S": "Black" },  
    ":p": { "N": "500" }  
}
```

Note

A `Scan` operation reads every item in a table; therefore, you should avoid using `Scan` with large tables.

The filter expression is applied to the `Scan` results, and items that do not match the filter expression are discarded.

If you define an expression attribute value, you must use it consistently throughout the entire expression. Also, you cannot omit the `:` symbol.

Expression attribute values are used with condition expressions, update expressions, and filter expressions.

Note

For programming language-specific code samples, see [Getting Started with DynamoDB \(p. 52\)](#).

Condition Expressions

To manipulate data in a 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 following are some AWS CLI examples of using condition expressions. These examples are based on the `ProductCatalog` table, which was introduced in [Specifying Item Attributes \(p. 338\)](#). The partition key for this table is `Id`; there is no sort key. The following `PutItem` operation creates a sample `ProductCatalog` item that we will refer to in the examples:

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

The arguments for `--item` are stored in the file `item.json`. (For simplicity, only a few item attributes are used.)

```
{  
    "Id": {"N": "456"},  
    "ProductCategory": {"S": "Sporting Goods"},  
    "Price": {"N": "650"}  
}
```

Topics

- [Preventing Overwrites of an Existing Item \(p. 346\)](#)
- [Checking for Attributes in an Item \(p. 347\)](#)
- [Conditional Deletes \(p. 347\)](#)
- [Conditional Updates \(p. 348\)](#)
- [Comparison Operator and Function Reference \(p. 349\)](#)

Preventing Overwrites of an Existing Item

The `PutItem` operation will overwrite an item with the same key (if it exists). If you want to avoid this, use a condition expression. This will allow the write to proceed only if the item in question does not already have the same key:

```
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 \(p. 349\)](#).

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 will succeed; otherwise, it will fail.

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 will delete 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)"
```

Note

For more information about `attribute_not_exists`, `attribute_exists`, and other functions, see [Comparison Operator and Function Reference \(p. 349\)](#).

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.

Let us revisit the item from [Condition Expressions \(p. 346\)](#):

```
{
  "Id": {
    "N": "456"
  },
  "Price": {
    "N": "650"
  },
  "ProductCategory": {
    "S": "Sporting Goods"
  }
}
```

Now 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 will attempt 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 file `values.json`:

```
{  
  ":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*—placeholder for an actual value. For more information, see [Expression Attribute Values \(p. 345\)](#). For more information about `IN`, `AND`, and other keywords, , see [Comparison Operator and Function Reference \(p. 349\)](#).

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 specify the changes you want to make to an item. For more information, see [Update Expressions \(p. 354\)](#).

Suppose that you started with the item shown in [Condition Expressions \(p. 346\)](#):

```
{  
  "Id": {"N": "456"},  
  "Price": {"N": "650"},  
  "ProductCategory": {"S": "Sporting Goods"}  
}
```

The following example performs an `UpdateItem` operation. It attempts to reduce the `Price` of a product by 75—but the condition expression prevents the update if the current `Price` is below 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 `--item` are stored in the file `values.json`:

```
{  
    ":discount": { "N": "75"},  
    ":limit": { "N": "500"}  
}
```

If the starting `Price` is 650, then 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*—placeholder for an actual value. For more information, see [Expression Attribute Values \(p. 345\)](#). For more information about `>` and other operators, see [Comparison Operator and Function Reference \(p. 349\)](#).

Comparison Operator and Function Reference

This section covers the built-in functions and keywords for writing condition expressions in DynamoDB.

Topics

- [Syntax for Condition Expressions \(p. 349\)](#)
- [Making Comparisons \(p. 350\)](#)
- [Functions \(p. 350\)](#)
- [Logical Evaluations \(p. 353\)](#)
- [Parentheses \(p. 354\)](#)
- [Precedence in Conditions \(p. 354\)](#)

Syntax for 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 (<i>path</i>)</code>	<p>True if the item contains the attribute specified by <i>path</i>.</p> <p>Example: Check whether an item in the <i>Product</i> table has a side view picture.</p> <ul style="list-style-type: none"> • <code>attribute_exists (Pictures.SideView)</code>
<code>attribute_not_exists (<i>path</i>)</code>	<p>True if the attribute specified by <i>path</i> does not exist in the item.</p> <p>Example: Check whether an item has a <i>Manufacturer</i> attribute</p> <ul style="list-style-type: none"> • <code>attribute_not_exists (Manufacturer)</code>
<code>attribute_type (<i>path</i>, <i>type</i>)</code>	<p>True if the attribute at the specified path is of a particular data type. The <i>type</i> parameter must be one of the following:</p> <ul style="list-style-type: none"> • <i>s</i> — String • <i>ss</i> — String Set • <i>n</i> — Number • <i>ns</i> — Number Set • <i>b</i> — Binary

Function	Description
	<ul style="list-style-type: none"> • <code>BS</code> — Binary Set • <code>BOOL</code> — Boolean • <code>NULL</code> — Null • <code>L</code> — List • <code>M</code> — Map <p>You must use an expression attribute value for the <code>type</code> parameter.</p> <p>Example: Check whether the <code>QuantityOnHand</code> attribute is of type List. In this example, <code>:v_sub</code> is a placeholder for the string <code>L</code>.</p> <ul style="list-style-type: none"> • <code>attribute_type</code> (<code>ProductReviews.FiveStar, :v_sub</code>) <p>You must use an expression attribute value for the second parameter.</p>
<code>begins_with (path, substr)</code>	<p>True if the attribute specified by <code>path</code> 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 (<i>path</i>, <i>operand</i>)</code>	<p>True if the attribute specified by <i>path</i> is:</p> <ul style="list-style-type: none"> • a String that contains a particular substring. • a Set that contains a particular element within the set. <p>In either case, <i>operand</i> must be a String.</p> <p>The path and the operand must be distinct; that is, <code>contains (a, a)</code> will return 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> <p>The expression attribute value <code>:v_sub</code> is a placeholder for Red.</p>

Function	Description
<code>size (path)</code>	<p>Returns a number representing an attribute's size. The following are valid data types for use with <code>size</code>.</p> <p>If the attribute is of type String, <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 Binary, <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>, which 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 Set data type, <code>size</code> returns the number of elements in the set.</p> <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"> • <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"> • <code>size(ProductReviews.OneStar) > :v_sub</code>

Logical Evaluations

Use the `AND`, `OR` and `NOT` keywords to perform logical evaluations. In the list following, `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.

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.

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

To update an existing item in a table, you use the `UpdateItem` operation. You must provide the key of the item you want to update. You must also provide an update expression, indicating the attributes you want to modify and the values 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 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 \(p. 341\)](#).

Topics

- [SET—Modifying or Adding Item Attributes \(p. 355\)](#)
- [REMOVE—Deleting Attributes From An Item \(p. 359\)](#)
- [ADD—Updating Numbers and Sets \(p. 360\)](#)
- [DELETE—Removing Elements From A Set \(p. 362\)](#)

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 attribute already exist, 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 by 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 we will refer to in the examples:

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

The arguments for `--item` are stored in the file `item.json`. (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 \(p. 356\)](#)
- [Adding Lists and Maps \(p. 356\)](#)
- [Adding Elements To a List \(p. 357\)](#)
- [Adding Nested Map Attributes \(p. 357\)](#)

- [Incrementing and Decrementing Numeric Attributes \(p. 358\)](#)
- [Appending Elements To a List \(p. 358\)](#)
- [Preventing Overwrites of an Existing Attribute \(p. 359\)](#)

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 file `values.json`:

```
{  
  ":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 file `values.json`:

```
{  
  ":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
```

The arguments for --expression-attribute-values are stored in the file values.json:

```
{
  ":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 does not already exist, `SET` will append 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 file names.json:

```
{
  "#pr": "ProductReviews",
  "#5star": "FiveStar",
  "#3star": "ThreeStar"
}
```

The arguments for --expression-attribute-values are stored in the file values.json:

```
{  
  ":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:

- `list_append (list1, list2)`

The function takes two lists as input, and appends `list2` to `list1`.

Example

In [Adding Elements To a List \(p. 357\)](#), we created the `RelatedItems` list and populated it with two elements: Hammer and Nails. Now we will 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 file `values.json`:

```
{  
  ":vals": {  
    "L": [  
      { "S": "Screwdriver" },
```

```
        {"S": "Hacksaw" }  
    ]  
}  
}
```

Finally, we will append one more element to the *beginning* of `RelatedItems`. To do this, we will 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:

- `if_not_exists (path, value)`

If the item does not contain an attribute at the specified `path`, then `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. To perform multiple `REMOVE` actions, separate them by commas.

The following is a syntax summary for `REMOVE` in an update expression. The only operand is the document path for the attribute you want to remove:

```
remove-action ::=  
    path
```

Example

Remove some attributes from an item. (If the attributes do not 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 \(p. 358\)](#), we modified a list attribute (`RelatedItems`) so that it contained five elements:

- [0]—Chisel
- [1]—Hammer
- [2]—Nails
- [3]—Screwdriver
- [4]—Hacksaw

The following 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 removing `Hammer` and `Nails`, 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 value(s) to an item.

If the attribute already exists, then 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, then the value is mathematically added to the existing attribute. (If the value is a negative number, then it is subtracted from the existing attribute.)

- If the attribute is a set, and the value you are adding is also a set, then the value is appended to the existing set.

Note

The `ADD` action only supports number and set data types.

To perform multiple `ADD` actions, separate them by 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
```

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"}}' \  
    --update-expression "ADD QuantityOnHand :q" \  
    --expression-attribute-values '{":q": {"N": "5"}}' \  
    --return-values ALL_NEW
```

Now that `QuantityOnHand` exists, you can re-run 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, we 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":["Orange", "Purple", "Red"]}}' \  
    --return-values ALL_NEW
```

```
--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 only supports 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 by 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`. That you want to delete. You must specify `subset` as a set type.

```
delete-action ::=  
    path value
```

Example

In [Adding Elements To A Set \(p. 361\)](#), we created the `Colors` 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
```

Time To Live

Time To Live (TTL) for DynamoDB allows you to define when items in a table expire so that they can be automatically deleted from the database.

TTL is provided at no extra cost as a way to reduce storage usage and reduce the cost of storing irrelevant data without using provisioned throughput. With TTL enabled on a table, you can set a timestamp for deletion on a per-item basis, allowing you to limit storage usage to only those records that are relevant.

TTL is useful if you have continuously accumulating data that loses relevance after a specific time period. For example: session data, event logs, usage patterns, and other temporary data. If you have sensitive data that must be retained only for a certain amount of time according to contractual or regulatory obligations, TTL helps you ensure that it is removed promptly and as scheduled.

To get started with TTL:

1. Understand [Time To Live: How It Works \(p. 363\)](#).
2. Read the [Before You Begin Using Time To Live \(p. 364\)](#) section.

3. Enable TTL on a specific table and choose the name of an attribute to hold the expiration timestamp. Then, you can add or update items in the table with timestamps in the attribute you chose. For more information, see [Enabling Time To Live \(p. 364\)](#).

Time To Live: How It Works

When Time To Live is enabled on a table, a background job checks the TTL attribute of items to see if they are expired.

TTL compares the current time in epoch time format to the time stored in the Time To Live attribute of an item. If the epoch time value stored in the attribute is less than the current time, the item is marked as expired and subsequently deleted.

Note

The epoch time format is the number of seconds elapsed since 12:00:00 AM January 1st, 1970 UTC.

DynamoDB deletes expired items on a best-effort basis to ensure availability of throughput for other data operations.

Important

DynamoDB typically deletes expired items within 48 hours of expiration. The exact duration within which an item truly gets deleted after expiration is specific to the nature of the workload and the size of the table. Items that have expired and not been deleted will still show up in reads, queries, and scans.

As items are deleted, they are removed from any Local Secondary Index and Global Secondary Index immediately in the same eventually consistent way as a standard delete operation.

For example, consider a table named *SessionData* that tracks the session history of users. Each item in *SessionData* is identified by a partition key (*UserName*) and a sort key (*SessionId*). Additional attributes like *UserName*, *SessionId*, *CreationTime* and *ExpirationTime* track the session information.

The following diagram shows how the items in the table would be organized. The *ExpirationTime* attribute is set as the Time To Live (TTL) attribute. (Not all of the attributes are shown)

SessionData

UserName	SessionId	CreationTime	ExpirationTime (TTL)	SessionInfo	...
user1	746865726527731461931200	1461938400	{JSON Document}
user2	6e6f7468696e671461920400	1461927600	{JSON Document}
user3	746f20736565201461922200	1461929400	{JSON Document}
user4	686572652121211461925380	1461932580	{JSON Document}
user5	6e6572642e2e2e1461920400	1461927600	{JSON Document}
...

In this example each item has an *ExpirationTime* attribute value set when it is created. Consider the first record:

SessionData

UserName	SessionId	CreationTime	ExpirationTime (TTL)	SessionInfo	...
user1	74686572652773	1461931200	1461938400	{JSON Document}	...

In this example, the item *CreationTime* is set to Friday, April 29 12:00 PM UTC 2016 and the *ExpirationTime* is set 2 hours later at Friday, April 29 2:00 PM UTC 2016. The item will expire when the current time, in epoch format, is greater than the time in the *ExpirationTime* attribute. In this case, the item with the key { *Username*: *user1*, *SessionId*: 74686572652773 } will expire after 2:00 PM (1461938400).

Note

Due to the potential delay between expiration and deletion time, you might get expired items when you query for items. If you don't need to view expired items when you issue a read request, you should filter out the expired items using the expiration attribute that you have defined.

You can do this by using a *filter expression* that returns only items where the Time To Live expiration value is greater than the current time in epoch format. For more information, see [Filter Expressions for Query \(p. 412\)](#) and [Filter Expressions for Scan \(p. 427\)](#).

Before You Begin Using Time To Live

Before you enable Time To Live on a table, consider the following:

- Ensure that any existing timestamp values in the specified Time To Live attribute are correct and in the right format.
- Items with an expiration time greater than 5 years in the past are not deleted.
- If data recovery is a concern, we recommend that you back up your table.
 - For a 24-hour recovery window, you can use Amazon DynamoDB Streams. For more information, see [DynamoDB Streams and Time To Live \(p. 523\)](#).
 - For a full backup, you can use AWS Data Pipeline. For more information, see [Exporting and Importing DynamoDB Data Using AWS Data Pipeline \(p. 722\)](#).
- You can use IAM policies to prevent unauthorized updates to the TTL attribute or configuration of the Time To Live feature. If you only allow access to specified actions in your existing IAM policies, ensure that your policies are updated to allow dynamodb:UpdateTimeToLive for roles that need to enable or disable Time To Live on tables. For more information, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 614\)](#).
- Consider whether you need to do any post-processing of deleted items. The Streams records of TTL deletes are marked and you can monitor them using an AWS Lambda function. For more information on the additions to the Streams record, see [DynamoDB Streams and Time To Live \(p. 523\)](#).

Enabling Time To Live

This section describes how to use the DynamoDB console or CLI to enable Time To Live. To use the API instead, see [Amazon DynamoDB API Reference](#).

Topics

- [Enable Time To Live \(console\) \(p. 365\)](#)
- [Enable Time To Live \(CLI\) \(p. 367\)](#)

Enable Time To Live (console)

To enable Time To Live using the DynamoDB console:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>.
2. Choose **Tables** and then choose the table that you want to modify.
3. In **Table details**, next to **TTL attribute**, choose **Manage TTL**.

Table details

Table name	ttl-test
Primary partition key	name (String)
Primary sort key	-
Time to live attribute	DISABLED Manage TTL
Table status	Active
Creation date	April 20, 2016 at 2:23:18 PM UTC-7

4. In the **Manage TTL** dialog box, choose **Enable TTL** and then type the **TTL attribute** name.

Enable TTL

TTL is a mechanism to set a specific timestamp for expiring items from an attribute on the items in the table. The attribute should be a Number. When the timestamp expires, the corresponding item is deleted from the table.

TTL attribute

ttl



Enabling TTL can take up to 1 hour to propagate. You cannot make further TTL changes until the action is complete. Be careful when enabling TTL on tables containing important data from your application.

DynamoDB Streams

Enable with view type **New and Old**.
Streams are currently not enabled



Enabling Streams gives a 24-hour backlog of writes. Maximum Write Capacity Limit may be exceeded.

Preview TTL

Before enabling TTL, it is recommended you run a preview to see sample items that will be deleted when TTL is enabled on this table.

Run preview

preview items expiring by

February 15, 2017

09

There are three settings in **Manage TTL**:

- **Enable TTL** – Choose this to either enable or disable TTL on the table. It may take up to one hour for the change to fully process.
 - **TTL Attribute** – The name of the DynamoDB attribute to store the TTL timestamp for items.
 - **24-hour backup streams** – Choose this to enable Amazon DynamoDB Streams on the table. For more information about how you can use DynamoDB Streams for backup, see [DynamoDB Streams and Time To Live \(p. 523\)](#).
5. (Optional) To preview some of the items that will be deleted when TTL is enabled, choose **Run preview**.

Warning

This provides you with a sample list of items. It does not provide you with a complete list of items that will be deleted by TTL.

6. Choose **Continue** to save the settings and enable TTL.

Now that TTL is enabled, the TTL attribute is marked **TTL** when you view items in the DynamoDB console.

You can view the date and time that an item will expire by hovering your mouse over the attribute.

	id	ttl (TTL)
<input type="checkbox"/>	7	1460232057
<input type="checkbox"/>	10	1459700753
<input type="checkbox"/>	3	1459221815 
<input type="checkbox"/>	2	1459221806
<input type="checkbox"/>	21	1459703052

UTC: March 29, 2016 at 3:23:35 AM UTC
 Local: March 28, 2016 at 8:23:35 PM UTC-7
 Region (N. Virginia): March 28, 2016 at 11:23:35 PM UTC

Enable Time To Live (CLI)

To enable TTL on the "TTLExample" table:

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-specification
  "Enabled=true, AttributeName=ttl"
```

To describe TTL on the "TTLExample" table:

```
aws dynamodb describe-time-to-live --table-name TTLExample
{
  "TimeToLiveDescription": {
    "AttributeName": "ttl",
    "TimeToLiveStatus": "ENABLED"
  }
}
```

To add an item to the "TTLExample" table with the Time To Live attribute set using the BASH shell and CLI:

```
EXP=`date -d '+5 days' +%s`
```

```
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl": {"N": "'`$EXP'"}}'
```

This example started with the current date and added five 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, five days is 432000 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. For example:

- Linux Terminal: `date +%s`
- Python: `import time; long(time.time())`
- Java: `System.currentTimeMillis() / 1000L`
- JavaScript: `Math.floor(Date.now() / 1000)`

Working with Items: Java

Topics

- [Putting an Item \(p. 368\)](#)
- [Getting an Item \(p. 371\)](#)
- [Batch Write: Putting and Deleting Multiple Items \(p. 373\)](#)
- [Batch Get: Getting Multiple Items \(p. 374\)](#)
- [Updating an Item \(p. 375\)](#)
- [Deleting an Item \(p. 377\)](#)
- [Example: CRUD Operations Using the AWS SDK for Java Document API \(p. 377\)](#)
- [Example: Batch Operations Using AWS SDK for Java Document API \(p. 381\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API \(p. 385\)](#)

You can use the AWS SDK for Java Document API to perform typical create, read, update, and delete (CRUD) operations on 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 you have to write. For more information, see [Java: DynamoDBMapper \(p. 195\)](#).

The following sections describe Java snippets to perform several Java Document API item actions. To run complete working examples instead, see:

- [Example: CRUD Operations Using the AWS SDK for Java Document API \(p. 377\)](#)
- [Example: Batch Operations Using AWS SDK for Java Document API \(p. 381\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API \(p. 385\)](#)

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 \(p. 375\)](#).

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 snippet demonstrates the preceding tasks. The snippet 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 snippet uses an optional parameter to specify a condition for uploading the item. If the condition you specify is not met, then the AWS Java SDK throws a `ConditionalCheckFailedException`. The code snippet specifies the following optional parameters in the `putItem` method:

- A `ConditionExpression` that defines the conditions for the request. The snippet defines the condition that the existing item that has the same primary key is replaced only if it has an `ISBN` attribute that equals a specific value.
- A map for `ExpressionAttributeValues` that will be used in the condition. In this case, there is only one substitution required: The placeholder `:val` in the condition expression will be 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 will parse the JSON document and map 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 snippet 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 snippet demonstrates the preceding steps. The code snippet 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", 101);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `getItem` method. For example, the following Java code snippet 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 \(p. 15\)](#).)

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 \(p. 341\)](#).

The parameters of the `getItem` method do not 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 code example below 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 example above looks like this:

```
{
    "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 \(p. 370\)](#) section, we stored a JSON document in a Map attribute named `VendorInfo`. You can use the `getItem` method to retrieve the entire document in JSON format, or use document path notation to retrieve only some of the elements in the document. The following Java code snippet 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 example above looks like this:

```
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 snippet 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 this:

```
{"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 will need to create one `TableWriteItems` instance per table.
3. Call the `batchWriteItem` method by providing the `TableWriteItems` object(s) 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 limit 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 [Limits in DynamoDB \(p. 731\)](#).

The following Java code snippet 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 \(p. 381\)](#).

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 will need to create one `TableKeysAndAttributes` instance per table.
3. Call the `batchGetItem` method by providing the `TableKeysAndAttributes` object(s) that you created in the preceding step.

The following Java code snippet 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 snippet 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` will replace the entire item. If there are attributes in the existing item that are not specified in the input, `putItem` will remove those attributes from the item.

In general, we recommend that you use `updateItem` whenever you want to modify any item attributes. The `updateItem` method will only modify the item attributes that you specify in the input, and the other attributes in the item will remain unchanged.

Follow these steps:

1. Create an instance of the `Table` class to represent the table 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 snippet demonstrates the preceding tasks. The snippet 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` will be 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, then the AWS Java SDK throws a `ConditionalCheckFailedException`. For example, the following Java code snippet 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 code snippet 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 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 code snippet 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 snippet specifies includes 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 sample illustrates CRUD operations on an 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, allowing 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: DynamoDBMapper \(p. 195\)](#).

Note

This code sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
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)

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id", 120)
            .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

Topics

- [Example: Batch Write Operation Using the AWS SDK for Java Document API \(p. 381\)](#)
- [Example: Batch Get Operation Using the AWS SDK for Java Document API \(p. 383\)](#)

This section provides examples of batch write and batch get operations using the AWS SDK for Java Document API.

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 you have to write. For more information, see [Java: DynamoDBMapper \(p. 195\)](#).

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 Sample Data \(p. 280\)](#) 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 \(p. 749\)](#).

For step-by-step instructions to test the following sample, see [Java Code Samples \(p. 285\)](#).

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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 sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
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.toJSONPretty());  
                }  
            }  
  
            // 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, allowing 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: DynamoDBMapper \(p. 195\)](#).

If you followed the [Creating Tables and Loading Sample Data \(p. 280\)](#) section, you should already have created the Reply table. You can also create this tables programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 749\)](#).

For step-by-step instructions to test the following sample, see [Java Code Samples \(p. 285\)](#).

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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

Topics

- [Putting an Item \(p. 388\)](#)
- [Getting an Item \(p. 389\)](#)
- [Updating an Item \(p. 390\)](#)
- [Atomic Counter \(p. 392\)](#)
- [Deleting an Item \(p. 393\)](#)
- [Batch Write: Putting and Deleting Multiple Items \(p. 394\)](#)
- [Batch Get: Getting Multiple Items \(p. 395\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Low-Level API \(p. 398\)](#)
- [Example: Batch Operations Using AWS SDK for .NET Low-Level API \(p. 401\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API \(p. 407\)](#)

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 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. Execute 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.

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 \(p. 390\)](#).

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. Execute the `PutItem` method by providing the `PutItemRequest` object that you created in the preceding step.

The following C# code snippet 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# code snippet. The sample 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 = "444-444444444" } },
        { "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 = "444-444444444"}}
    },
    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 \(p. 395\)](#).

The following are the steps to retrieve 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 `GetItemRequest` class.

To get an item, you must provide the table name and primary key of the item.

3. Execute the `GetItem` method by providing the `.GetItemRequest` object that you created in the preceding step.

The following C# code snippet 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# code snippet. 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 \(p. 15\)](#).

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 \(p. 388\)](#). For example, if you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. Note that, 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` only updates 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. Execute the `UpdateItem` method by providing the `UpdateItemRequest` object that you created in the preceding step.

The following C# code snippet 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# code snippet. 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 code snippet 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. Execute 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 snippet. 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.
    ReturnValue = "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. Execute 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 limit 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 snippet 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 then executes `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 AWS SDK for .NET Low-Level API \(p. 401\)](#).

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 .NET SDK 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. Execute 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 limit or some other transient error.

The following C# code snippet 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" } },
                        { "Name", new AttributeValue { S = "Amazon S3" } }
                    }
                }
            },
            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 snippet. The code samples 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 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 Sample Data \(p. 280\)](#), 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 \(p. 756\)](#).

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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>()
    };
    request.Key["Id"] = new AttributeValue {
        N = "1000"
    };
    request.ProjectionExpression = "Id, ISBN, Title, Authors",
    request.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>()
    };
    request.Key["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.
        ReturnValue = "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 AWS SDK for .NET Low-Level API

Topics

- [Example: Batch Write Operation Using the AWS SDK for .NET Low-Level API \(p. 402\)](#)
- [Example: Batch Get Operation Using the AWS SDK for .NET Low-Level API \(p. 404\)](#)

This section provides examples of batch operations, batch write and batch get, that 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 Sample Data \(p. 280\)](#), 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 \(p. 756\)](#).

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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 tableName = "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. 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 Sample Data \(p. 280\)](#), 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 \(p. 756\)](#).

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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>()
            };
            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 = 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 Sample Data \(p. 280\)](#), 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 \(p. 756\)](#).

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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();
        }
}
}

```

Working with Queries

Topics

- [Key Condition Expression \(p. 410\)](#)
- [Filter Expressions for Query \(p. 412\)](#)
- [Limiting the Number of Items in the Result Set \(p. 413\)](#)
- [Paginating the Results \(p. 413\)](#)
- [Counting the Items in the Results \(p. 414\)](#)
- [Capacity Units Consumed by Query \(p. 414\)](#)
- [Read Consistency for Query \(p. 415\)](#)
- [Querying Tables and Indexes: Java \(p. 415\)](#)
- [Querying Tables and Indexes: .NET \(p. 420\)](#)

The `Query` operation finds items based on primary key values. You can query any table or secondary index that has a composite primary key (a partition key and a sort key).

You must provide the name of the partition key attribute, and a single value for that attribute. `Query` will return all of the items with that partition key value. You can optionally provide a sort key attribute, and use a comparison operator to refine the search results.

Key Condition Expression

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 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 <= 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`
- `a BETWEEN b 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 CLI examples demonstrate the use of key condition expressions. Note that these expressions use placeholders (such as `:name` and `:sub`) instead of actual values. For more information, see [Expression Attribute Names \(p. 342\)](#) and [Expression Attribute Values \(p. 345\)](#).

Example

Query the `Thread` table for a particular `ForumName` (partition key). All of the items with that `ForumName` value will be 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 file values.json:

```
{
  ":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 file values.json:

```
{
  ":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 second character (if present) is 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 \(p. 780\)](#).) If an attribute name does not meet these requirements, you will need to define an expression attribute name as a placeholder. For more information, see [Expression Attribute Names \(p. 342\)](#).

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 will be 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` is applied to the results. If `LastEvaluatedKey` is present in the response and is non-null, you will need to paginate the result set (see [Paginating the Results \(p. 413\)](#)).

Filter Expressions for *Query*

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* will consume 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 identical to that of a condition expression. Filter expressions can use the same comparators, functions, and logical operators as a condition expression. For more information, [Condition Expressions \(p. 346\)](#).

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" \
--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 file `values.json`:

```
{
  ":fn": {"S": "Amazon DynamoDB#DynamoDB Thread 1"},
  ":num": {"N": "3"}
}
```

Note that `views` is a reserved word in DynamoDB (see [Reserved Words in DynamoDB \(p. 780\)](#)), so this example uses `#v` as a placeholder. For more information, see [Expression Attribute Names \(p. 342\)](#).

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.

Limiting the Number of Items in the Result Set

The `Query` operation allows 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.

For example, suppose you `Query` a table, with a `Limit` value of 6, and without a filter expression. The `Query` result will contain the first six items from the table that match the key condition expression from the request.

Now suppose you add a filter expression to the `Query`. In this case, DynamoDB will apply the filter expression to the six items that were returned, discarding those that do not match. The final `Query` result will contain 6 items or fewer, depending on the number of items that were filtered.

Paginating the 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` will only return 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, 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 `Query` request, with the same parameters as the previous one—but 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. (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 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":"1989"}}' \  
  --page-size 5 \  
  --debug
```

Ordinarily, the AWS CLI handles pagination automatically; however, in this example, the CLI's `--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 this:

```
2017-07-07 11:13:15,603 - MainThread - botocore.parsers - DEBUG - Response body:
```

```
b'{"Count":5,"Items":[{"title":{"S":"Always"}},{"title":{"S":"Back to the Future Part II"}}, {"title":{"S":"Batman"}},{"title":{"S":"Bill & Ted's Excellent Adventure"}}, {"title":{"S":"Black Rain"}]}, "LastEvaluatedKey":{"year":{"N":"1989"}}, "title":{"S":"Black Rain"}}, "ScannedCount":5}
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI will then issue 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":2,"Items":[{"title":{"S":"Uncle Buck"}}, {"title":{"S":"Weekend at Bernie's"}]}, "ScannedCount":2}'
```

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 samples 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 `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 do not use a filter expression, then `ScannedCount` and `Count` will have the same value.

If the size of the `Query` result set is larger than 1 MB, then `ScannedCount` and `Count` will represent only a partial count of the total items. You will need to perform multiple `Query` operations in order to retrieve all of the results (see [Paginating the Results \(p. 413\)](#)).

Each `Query` response will contain 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, provided that it has a composite primary key (partition key and sort key). `Query` operations consume read capacity units, as follows:

If you <code>Query</code> 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 item size, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed will be the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number will also be the same whether or not you use a filter expression.

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 \(p. 15\)](#).

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. 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 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 you have to write. For more information, see [Java: DynamoDBMapper \(p. 195\)](#).

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 item(s) 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 snippet demonstrates the preceding tasks. The snippet assumes you have a `Reply` table that stores replies for forum threads. For more information, see [Creating Tables and Loading Sample Data \(p. 280\)](#).

```
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 snippet retrieves forum thread replies posted in the past 15 days. The snippet specifies optional parameters using:

- 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 snippet uses a `QuerySpec` object which 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 time 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 snippet modifies the query specification shown above. 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 Sample Data \(p. 280\)](#).

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 you have to write. For more information, see [Java: DynamoDBMapper \(p. 195\)](#).

Example

```

Forum ( Name, ... )
Thread ( ForumName, Subject, Message, LastPostedBy, LastPostDateTime, ... )
Reply ( Id, ReplyDateTime, Message, PostedBy, ... )

```

In this Java code example, you execute 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 your code processes all the pages in the query result.
- Find replies in the last 15 days.
- Find replies in a specific date range.

Both the preceding two queries shows how you can specify sort key conditions to narrow the query results and use other optional query parameters.

Note

This code sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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. 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 low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class and provide query operation parameters.
3. Execute 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 snippet demonstrates the preceding tasks. The snippet assumes you have a `Reply` table stores replies for forum threads. For more information, see [Creating Tables and Loading Sample Data \(p. 280\)](#).

Example

```
Reply ( <emphasis role="underline">Id</emphasis>, <emphasis
role="underline">ReplyDateTime</emphasis>, ... )
```

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 Amazon 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 snippet retrieves forum thread replies posted in the past 15 days. The snippet 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 execute the `Query` method, you get one page of results that has the specified number of items. To fetch the next page, you execute 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# code snippet 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 Sample Data \(p. 280\)](#).

Example

```
Forum ( <emphasis role="underline">Name</emphasis>, ... )
Thread ( <emphasis role="underline">ForumName</emphasis>, <emphasis
        role="underline">Subject</emphasis>, Message, LastPostedBy, LastPostDateTime, ... )
Reply ( <emphasis role="underline">Id</emphasis>, <emphasis
        role="underline">ReplyDateTime</emphasis>, Message, PostedBy, ... )
```

In this C# code example, you execute 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 illustrate the use of pagination to process multipage result. Amazon 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.

Both of the preceding two queries shows how you can specify sort key conditions to narrow query results and use other optional query parameters.

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

```
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.ISO8601DateFormat);

var request = new QueryRequest
{
    TableName = "Reply",
    ReturnConsumedCapacity = "TOTAL",
    KeyConditionExpression = "Id = :v_replyId and ReplyDateTime
> :v_interval",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {"": replyId
        S = replyId
    }},
    {"": twoWeeksAgoString
    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> {
                {"": replyId
                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

Topics

- [Filter Expressions for Scan \(p. 427\)](#)
- [Limiting the Number of Items in the Result Set \(p. 428\)](#)
- [Paginating the Results \(p. 428\)](#)
- [Counting the Items in the Results \(p. 429\)](#)
- [Capacity Units Consumed by Scan \(p. 429\)](#)
- [Read Consistency for Scan \(p. 430\)](#)
- [Parallel Scan \(p. 430\)](#)
- [Scanning Tables and Indexes: Java \(p. 431\)](#)
- [Scanning Tables and Indexes: .NET \(p. 438\)](#)

A `Scan` operation 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 will be empty.

A single `Scan` request can retrieve a maximum of 1 MB of data; DynamoDB can optionally apply a filter expression to this data, narrowing the results before they are returned to the user.

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` will consume 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. For more information, [Condition Expressions \(p. 346\)](#).

Example

The following 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"}}'
```

Limiting the Number of Items in the Result Set

The `Scan` operation allows 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.

For example, suppose you `Scan` a table, with a `Limit` value of 6, and without a filter expression. The `Scan` result will contain the first six items from the table that match the key condition expression from the request.

Now suppose you add a filter expression to the `Scan`. In this case, DynamoDB will apply the filter expression to the six items that were returned, discarding those that do not match. The final `Scan` result will contain 6 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` will only return 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—but 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, then 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 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 CLI's `--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 this:

```
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 will then issue 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 samples 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 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 do not use a filter expression, then `ScannedCount` and `Count` will have the same value.

If the size of the `Scan` result set is larger than 1 MB, then `ScannedCount` and `Count` will represent only a partial count of the total items. You will need to perform multiple `Scan` operations in order to retrieve all of the results (see [Paginating the Results \(p. 428\)](#)).

Each `Scan` response will contain the `ScannedCount` and `Count` for the items that were processed by that particular `Scan` request. To obtain 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 <code>Scan</code> 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 item size, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed will be the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number will also be the same whether or not you use a filter expression.

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 \(p. 15\)](#).

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 will ensure that all of the write operations that completed before the `Scan` began will be 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`, in order 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` completes, you can apply the write activity from the stream to the table.

Note

Note that a `Scan` operation with `ConsistentRead` set to `true` will consume 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. 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` will take to complete. In addition, a sequential `Scan` might not always be able to fully utilize 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.

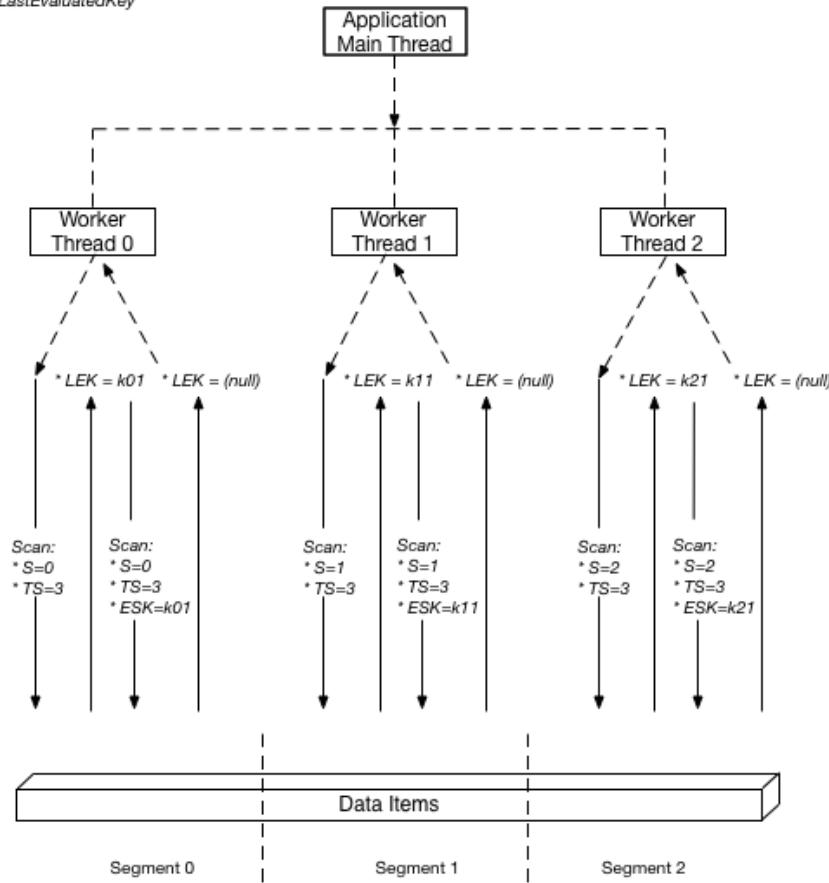
To address these issues, the `Scan` operation can logically divide a table or secondary index into multiple *segments*, with multiple application workers scanning the segments in parallel. Each worker can be a thread (in programming languages that support multithreading) or an operating system process. 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.

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. Execute 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 snippet 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");

ScanResult 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 snippet scans the `ProductCatalog` table to find items that are priced less than 0. The snippet 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);

ScanResult 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 execute the `scan` method, you get one page of results that has the specified number of items. To fetch the next page, you execute 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 snippet 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);

    ScanResult 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, allowing 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: DynamoDBMapper \(p. 195\)](#).

Note

This code sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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 {
        findProductsForPriceLessThanZero();
    }

    private static void findProductsForPriceLessThanZero() {
        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 run time statistics for each parallel request.

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 you have to write. For more information, see [Java: DynamoDBMapper \(p. 195\)](#).

Note

This code sample assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. For step-by-step instructions to run the following example, see [Java Code Samples \(p. 285\)](#).

```

// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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 parallelScanTableName = "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(parallelScanTableName);
            createTable(parallelScanTableName, 10L, 5L, "Id", "N");

            // Upload sample data for scan
            uploadSampleProducts(parallelScanTableName, scanItemCount);

            // Scan the table using multiple threads
            parallelScan(parallelScanTableName, 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",
            "111-1111111111")
            .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);

    }
    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.

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. Execute the `Scan` method and provide the `QueryRequest` 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 snippet 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# code snippet 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 execute the `Scan` method, you get one page of results that has the specified number of items. To fetch the next page, you execute 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 snippet 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 example provides a working sample that scans the `ProductCatalog` table to find items priced less than 0.

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

```
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> {
                {"<price>", new AttributeValue {
                    N = "0"
                }},
                {"<filter>", "Price < :val"},
                {"<projection>", "Id, Title, Price"}
            };
            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 re-creates the ProductCatalog table, 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 run time statistics.

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

```
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");
}
}
```

Improving Data Access with Secondary Indexes

Topics

- [Global Secondary Indexes \(p. 448\)](#)
- [Local Secondary Indexes \(p. 484\)](#)

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 gives 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.
- **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.

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 table, or delete an existing global secondary index. For more information, see Managing Global Secondary Indexes (p. 455) .	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 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.	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; these updates consume write capacity units from the base table.
Projected Attributes	With global secondary index queries or scans, you can only request the attributes that are projected into the index. DynamoDB will 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 will automatically fetch 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 attempt to concurrently create more than one table with a secondary index, DynamoDB will return a `LimitExceededException`.

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 [Limits in DynamoDB \(p. 731\)](#). 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 5 global secondary indexes and up to 5 local secondary indexes per table.

To get a detailed listing of secondary indexes on a table, use the `DescribeTable` operation. `DescribeTable` will return 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 Local Secondary Indexes \(p. 682\)](#) and [Best Practices for Global Secondary Indexes \(p. 684\)](#), respectively.

Global Secondary Indexes

Topics

- [Attribute Projections \(p. 451\)](#)
- [Querying a Global Secondary Index \(p. 452\)](#)
- [Scanning a Global Secondary Index \(p. 453\)](#)
- [Data Synchronization Between Tables and Global Secondary Indexes \(p. 453\)](#)

- [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 454\)](#)
- [Storage Considerations for Global Secondary Indexes \(p. 455\)](#)
- [Managing Global Secondary Indexes \(p. 455\)](#)
- [Working with Global Secondary Indexes: Java \(p. 467\)](#)
- [Working with Global Secondary Indexes: .NET \(p. 474\)](#)

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. To illustrate, consider a table named *GameScores* that keeps track of 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, making it difficult to answer questions such as these:

- 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*. Since the base table's primary key attributes are always projected into an index, the *UserId* attribute is also present. The following diagram shows what *GameTitleIndex* index would look like:

GameTitleIndex

<i>GameTitle</i>	<i>TopScore</i>	<i>UserId</i>
“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 queried *GameTitleIndex*, as shown in the diagram above, the query would not be able to access any non-key attributes other than *TopScore* (though 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. Here is some data that we could use to 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 will propagate it to *GameTitleIndex*. If we then query the index using Comet Quest for *GameTitle* and 0 for *TopScore*, the following data is returned:

<i>GameTitle</i>	<i>TopScore</i>	<i>UserId</i>
"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 keeps track of data items where its key attribute(s) actually exist. For example, suppose that you added another new item to the *GameScores* table, but only provided the required primary key attributes:

<i>UserId</i>	<i>GameTitle</i>
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:

<i>UserId</i>	<i>GameTitle</i>	<i>TopScore</i>
"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:

<i>GameTitle</i>	<i>TopScore</i>	<i>UserId</i>
"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 diagram above, *GameTitleIndex* does not have any additional projected attributes. An application can use *GameTitle* and *TopScore* in queries; however, it is not possible to efficiently determine which user has the highest score for a particular game, or the highest ratio of wins vs. losses. The most 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 will cost to store it, and the less your write costs will be.
- If your application will frequently access some non-key attributes, you should consider projecting those attributes into a global secondary index. The additional storage costs for the global secondary index will 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 will give 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.

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 \(p. 427\)](#).

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.

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, and the *TopScore* sort key in the index is of type Number. If you attempt to add an item to the *GameScores* table and specify a different data type for either *GameTitle* or *TopScore*, DynamoDB will return 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 in order to write a new item to the *GameScores* table. In this case, Amazon DynamoDB does not write any data to the index for this particular item.

A table with many global secondary indexes will incur higher costs for write activity than tables with fewer indexes. For more information, see [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 454\)](#).

Provisioned Throughput Considerations for Global Secondary Indexes

When you create a global secondary index, 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, then the write activity on the table will be throttled.

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 will be 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 $2 \times 4 \text{ KB} = 8 \text{ KB}$ 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 on how DynamoDB calculates provisioned throughput usage, see [Throughput Settings for Reads and Writes \(p. 294\)](#).

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 2000 bytes of data. Now suppose that you query this index and that the query returns 8 items. The total size of the matching items is $2000 \text{ bytes} \times 8 \text{ items} = 16,000 \text{ bytes}$; this is then rounded up to the nearest 4 KB boundary. Since 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, then the global secondary index will consume 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. Note that if a write to a table does not require a global secondary index update, then no write capacity is consumed from the index.

In order 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 will be 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, then 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 will 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, then DynamoDB does not 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.

Topics

- [Creating a Table With Global Secondary Indexes \(p. 456\)](#)
- [Describing the Global Secondary Indexes on a Table \(p. 456\)](#)
- [Adding a Global Secondary Index To an Existing Table \(p. 456\)](#)
- [Modifying an Index Creation \(p. 458\)](#)
- [Deleting a Global Secondary Index From a Table \(p. 459\)](#)
- [Detecting and Correcting Index Key Violations \(p. 459\)](#)

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 5 global secondary indexes 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 [Global Secondary Indexes \(p. 448\)](#)), 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 \(p. 12\)](#).

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 \(p. 454\)](#).

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 will include 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.

Note

You cannot cancel an in-flight global secondary index creation.

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`.

Behind the scenes, DynamoDB builds the index in two phases:

Resource Allocation

DynamoDB allocates the compute and storage resources that will be 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 cannot delete its parent table; nor can you 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 keeps track of 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 `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 modify the provisioned throughput of the table and any of its global secondary indexes.

Note

During the backfilling phase, some writes of violating items may succeed while others will be rejected. After backfilling, all writes to items that violate the new index's key schema will be rejected. We recommend that you run the Violation Detector tool after the backfill phase completes, to detect and resolve any key violations that may have occurred. For more information, see [Detecting and Correcting Index Key Violations \(p. 459\)](#).

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 will not be charged for this read activity.

When the index build is complete, its status changes to `ACTIVE`. You will not be able to `Query` or `Scan` the index until it is `ACTIVE`.

Note

In some cases, DynamoDB will not be able to write data from the table to the index due to 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, or if the size of an attribute exceeds the maximum length

for an index key attribute. Index key violations do not interfere with global secondary index creation; however, when the index becomes `ACTIVE`, the violating keys will not be 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 \(p. 459\)](#).

[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 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 on CloudWatch metrics related to DynamoDB, see [DynamoDB Metrics \(p. 644\)](#).

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 may 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.

[Modifying an Index 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, then 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 will change to `UPDATING`, and `Backfilling` will be true until the index is ready for use.

During the backfilling phase, you cannot 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 \(p. 457\)](#).

Deleting a Global Secondary Index From a Table

If you no longer need a global secondary index, you can delete it using the `UpdateTable` operation.

You can only delete 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.

Detecting and Correcting Index Key Violations

During the backfill phase of global secondary index creation, 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 will remain in the table, but the index will not have a corresponding entry for that item.

An *index key violation* occurs if:

- There is a data type mismatch between an attribute value and the index key schema data type. For example, suppose 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.

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 will run on Windows, Mac, or Linux computers. Violation Detector requires Java 1.7 (or above) and Maven.

- <https://github.com/awslabs/dynamodb-online-index-violation-detector>

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 \(p. 460\)](#).
- `-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 will not be modified. If the value is set to `delete`, items with key violations will be 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 will be updated with new, non-violating values. If the value is set to `delete`, items with key violations will be 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?
<code>awsCredentialsFile</code>	<p>The fully qualified name of a file containing your AWS credentials. The credentials file must be in the following format:</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>accessKey = access_key_id_goes_here secretKey = secret_key_goes_here</pre> </div>	Yes
<code>dynamoDBRegion</code>	The AWS region in which the table resides. For example: <code>us-west-2</code> .	Yes

Parameter Name	Description	Required?
<code>tableName</code>	The name of the DynamoDB table to be scanned.	Yes
<code>gsiHashKeyName</code>	The name of the index partition key.	Yes
<code>gsiHashKeyType</code>	The data type of the index partition key—String, Number, or Binary: <code>S</code> <code>N</code> <code>B</code>	Yes
<code>gsiRangeKeyName</code>	The name of the index sort key. Do not specify this parameter if the index only has a simple primary key (partition key).	No
<code>gsiRangeKeyType</code>	The data type of the index sort key—String, Number, or Binary: <code>S</code> <code>N</code> <code>B</code> Do not specify this parameter if the index only has a simple primary key (partition key).	No
<code>recordDetails</code>	Whether to write the full details of index key violations to the output file. If set to <code>true</code> (the default), full information about the violating items are reported. If set to <code>false</code> , only the number of violations is reported.	No
<code>recordGsiValueInViolationRecord</code>	Whether to write the values of the violating index keys to the output file. If set to <code>true</code> (default), the key values are reported. If set to <code>false</code> , 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/filename.csv detectionOutputPath = s3://bucket/filename.csv</pre> <p>Information in the output file appears in CSV format (comma-separated values). If you do not set <code>detectionOutputPath</code>, then the output file is named <code>violation_detection.csv</code> and is written to your current working directory.</p>	No
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 will be scanned in a sequential manner. If the value is 2 or higher, then Violation Detector will divide 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 will distribute its read activity evenly across multiple partitions.</p> <p>For more information on parallel scans in DynamoDB, see Parallel Scan (p. 430).</p>	No

Parameter Name	Description	Required?
<code>numOfViolations</code>	The upper limit of index key violations to write to the output file. If set to <code>-1</code> (the default), the entire table will be scanned. If set to a positive integer, then Violation Detector will stop after it encounters that number of violations.	No
<code>numOfRecords</code>	The number of items in the table to be scanned. If set to <code>-1</code> (the default), the entire table will be scanned. If set to a positive integer, then Violation Detector will stop after it scans that many items in the table.	No
<code>readWriteIOPSPercent</code>	Regulates the percentage of provisioned read capacity units that are consumed during the table scan. Valid values range from <code>1</code> to <code>100</code> . The default value (<code>25</code>) means that Violation Detector will consume no more than 25% of the table's provisioned read throughput.	No
<code>correctionInputPath</code>	<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?
correctionOutputPath	<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 Simple Storage Service (Amazon S3). The following are examples:</p> <pre>correctionOutputPath = //local/path/filename.csv</pre> <pre>correctionOutputPath = s3://bucket/filename.csv</pre> <p>Information in the output file appears in CSV format (comma-separated values). If you do not set <code>correctionOutputPath</code>, then 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 Sample Data \(p. 280\)](#). 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	-2
102	20
103	200
201	100
202	200
203	300
204	400
205	500

Note that 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 we add another item to the `ProductCatalog` table:

Id (Primary Key)	Price
999	"Hello"

The table now has a total of nine items.

Now we will 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 = ./gsi_violation_check.csv
correctionInputPath = ./gsi_violation_check.csv
numOfSegments = 1
readWriteIOPSPercent = 40
```

Next, you run Violation Detector as in this 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: ./gsi_violation_check.csv
```

If the `recordDetails` config parameter is set to `true`, then 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 comma-separated value format (CSV). 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 our `ProductCatalog` example, suppose that we want to delete the violating item from the table. To do this, we 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: ./gsi_violation_check.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 a 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.

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 snippet 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 snippet demonstrates the preceding steps. The snippet 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.

Note that since `Precipitation` is not a key attribute for the table, it is not required; however, `WeatherData` items without `Precipitation` will 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 snippet 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, we must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following Java code snippet 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 `IssueId` and the sort key is `Title`. 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, and then queries the data using the global secondary indexes. Finally, the program deletes the *Issues* table.

For step-by-step instructions to test the following sample, see [Java Code Samples \(p. 285\)](#).

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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

Topics

- [Create a Table With a Global Secondary Index \(p. 474\)](#)
- [Describe a Table With a Global Secondary Index \(p. 476\)](#)
- [Query a Global Secondary Index \(p. 477\)](#)
- [Example: Global Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 478\)](#)

You can use the AWS SDK for .NET low-level API to create a 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. Execute the appropriate method provided by the client that you created in the preceding step.

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 snippet 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. Execute the `CreateTable` method by providing the request object as a parameter.

The following C# code snippet demonstrates the preceding steps. The snippet 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.

Note that since `Precipitation` is not a key attribute for the table, it is not required; however, `WeatherData` items without `Precipitation` will 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 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. Execute the `describeTable` method by providing the request object as a parameter.

The following C# code snippet 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. Execute the `query` method by providing the request object as a parameter.

The attribute name `Date` is a DynamoDB reserved word. Therefore, we must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following C# code snippet 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);
        }
    }
}

```

```

        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 `IssueId` and the sort key is `Title`. 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, and then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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
        {
    
```



```

        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;
        }
    }
}
}
```

Local Secondary Indexes

Topics

- [Attribute Projections \(p. 486\)](#)
- [Creating a Local Secondary Index \(p. 488\)](#)
- [Querying a Local Secondary Index \(p. 488\)](#)
- [Scanning a Local Secondary Index \(p. 489\)](#)
- [Item Writes and Local Secondary Indexes \(p. 489\)](#)
- [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 490\)](#)
- [Storage Considerations for Local Secondary Indexes \(p. 491\)](#)
- [Item Collections \(p. 492\)](#)
- [Working with Local Secondary Indexes: Java \(p. 494\)](#)
- [Working with Local Secondary Indexes: .NET \(p. 503\)](#)

Some applications only need to query data using the base table's primary key; however, there may be situations where an alternate sort key would be helpful. To give your application a choice of sort keys,

you can create one or more local secondary indexes on a table and issue `Query` or `Scan` requests against these indexes.

For example, consider the *Thread* table that is defined in [Creating Tables and Loading Sample Data \(p. 280\)](#). 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	Thread
"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 contiguously. 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. 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

<i>ForumName</i>	<i>LastPostDateTime</i>	<i>Subject</i>
"S3"	"2015-01-03:09:21:11"	"ddd"
	"2015-01-22:23:18:01"	"bbb"
	"2015-02-31:13:14:21"	"ccc"
	"2015-03-15:17:24:31"	"aaa"
"EC2"	"2015-01-18:07:33:42"	"zzz"
	"2015-02-12:11:07:56"	"yyy"
"RDS"	"2015-01-19:01:13:24"	"rrr"
	"2015-02-22:12:19:44"	"ttt"
	"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 will automatically fetch 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 \(p. 492\)](#).

Attribute Projections

With *LastPostIndex*, an application could use *ForumName* and *LastPostDateTime* as query criteria; however, to retrieve any additional attributes, DynamoDB would need to 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 web page 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 web page 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, note that 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 will cost to store it,

and the less your write costs will be. 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 will frequently access some non-key attributes, you should consider projecting those attributes into a local secondary index. The additional storage costs for the local secondary index will 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 will give 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 \(p. 12\)](#).

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 \(p. 493\)](#).

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 \(p. 12\)](#).

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 will return 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 will always return the latest updated values. If the query needs to fetch additional attributes from the base table, then 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, then 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 \(p. 427\)](#).

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 attempt to add an item to the *Thread* table and specify a different data type for *LastPostDateTime* (such as Number), DynamoDB will return a *ValidationException* because of the data type mismatch.

If you write an item to a table, you don't have to specify the attributes for any local secondary index sort key. Using *LastPostIndex* as an example, you would not need to specify a value for the *LastPostDateTime* attribute in order to write a new item to the *Thread* table. In this case, DynamoDB does not write any data to the index for this particular item.

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. For more information, see [Take Advantage of Sparse Indexes \(p. 683\)](#).

A table with many local secondary indexes will incur higher costs for write activity than tables with fewer indexes. For more information, see [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 490\)](#).

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 \(p. 493\)](#).

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, go to <https://aws.amazon.com/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 on how DynamoDB calculates provisioned throughput usage, see [Throughput Settings for Reads and Writes \(p. 294\)](#).

For index queries that read attributes that are not projected into the local secondary index, DynamoDB will need 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 above, 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 will consume 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 will 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 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 base table.

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. For more information about this behavior, see [Take Advantage of Sparse Indexes \(p. 683\)](#).

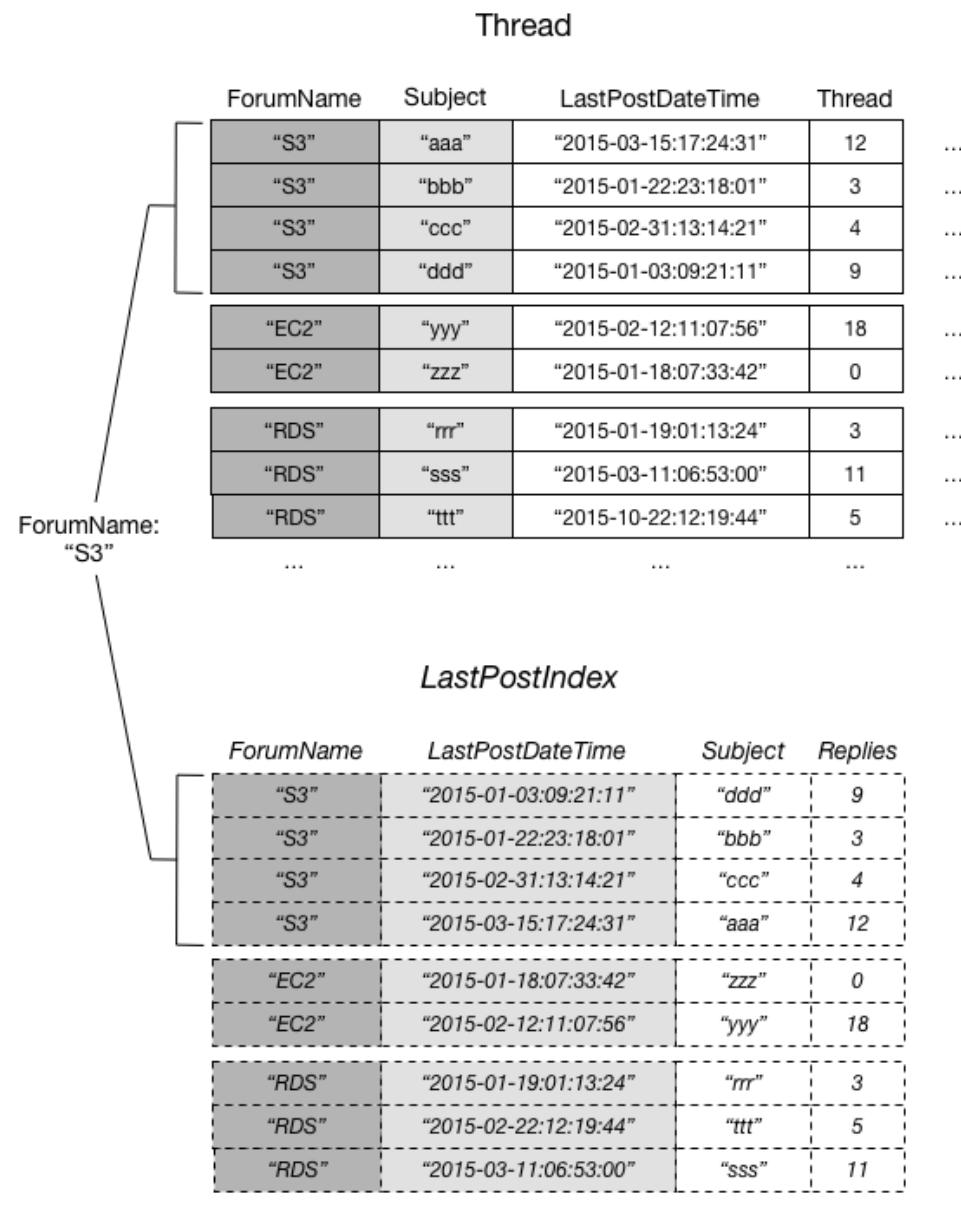
Item Collections

Note

The following 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:



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, then 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`

Each of these operations support 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

Here is a snippet 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 gigabyte. 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 is 10 GB. This limit does not apply to tables without local secondary indexes; only tables that have one or more local secondary indexes are affected.

If an item collection exceeds the 10 GB limit, DynamoDB will return an `ItemCollectionSizeLimitExceeded` exception 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 will still be able to 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 will also remove 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 will also reduce 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 will once again be able to 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

The table and index data for each item collection is stored in a single partition. Referring to the *Thread* table example, all of the base table and index items with the same *ForumName* attribute would be stored in the same partition. The "S3" item collection would be stored on one partition, "EC2" in another partition, and "RDS" in a third partition.

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. For more information see [Best Practices for Tables \(p. 666\)](#).

Working with Local Secondary Indexes: Java

Topics

- [Create a Table With a Local Secondary Index \(p. 494\)](#)
- [Describe a Table With a Local Secondary Index \(p. 496\)](#)
- [Query a Local Secondary Index \(p. 497\)](#)
- [Example: Local Secondary Indexes Using the Java Document API \(p. 497\)](#)

You can use the AWS SDK for Java Document API to create a 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.

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 snippet 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 snippet demonstrates the preceding steps. The snippet 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 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 snippet 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 you `Query` a table. You need to 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 \(p. 486\)](#).

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 snippet 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. 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, and then queries the data using the local secondary indexes. Finally, the program deletes the *CustomerOrders* table.

For step-by-step instructions to test the following sample, see [Java Code Samples \(p. 285\)](#).

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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 \(p. 504\)](#)
- [Describe a Table With a Local Secondary Index \(p. 505\)](#)
- [Query a Local Secondary Index \(p. 506\)](#)

- [Example: Local Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 507\)](#)

You can use the AWS SDK for .NET low-level API to create a 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 Samples \(p. 287\)](#).

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. Execute 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 you create a table. To do this, use `CreateTable` and provide your specifications for one or more local secondary indexes. The following C# code snippet 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. Execute the `CreateTable` method by providing the request object as a parameter.

The following C# code snippet demonstrates the preceding steps. The snippet 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. Execute the `describeTable` method by providing the request object as a parameter.

The following C# code snippet 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\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 need to 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 \(p. 486\)](#)

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. Execute the `query` method by providing the request object as a parameter.

The following C# code snippet 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 attrbs in response.Items)
{
    foreach (var attrib in attrbs)
    {
        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. 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, and then queries the data using the local secondary indexes. Finally, the program deletes the *CustomerOrders* table.

For step-by-step instructions to test the following sample, see [.NET Code Samples \(p. 287\)](#).

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"
    } },
    { 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"
    } },
    { 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"
    } }
};

```

```

        },
        { new KeySchemaElement() {
            AttributeName = "IsOpen", KeyType = "RANGE"
        }
    };
}

// 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;
        }
    }
}
```

Capturing Table Activity with DynamoDB Streams

Many applications can benefit from the ability to capture changes to items stored in a DynamoDB table, at the point in time when such changes occur. Here are some example use cases:

- An application in one AWS region modifies the data in a DynamoDB table. A second application in another AWS region reads these data modifications and writes the data to another table, creating a replica that stays in sync with the original table.
 - 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 global multi-player game has a multi-master topology, storing data in multiple AWS regions. Each master stays in sync by consuming and replaying the changes that occur in the remote regions.
 - 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 Streams enables solutions such as these, and many others. 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.

A *DynamoDB stream* is an ordered flow of information about changes to items in an Amazon DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

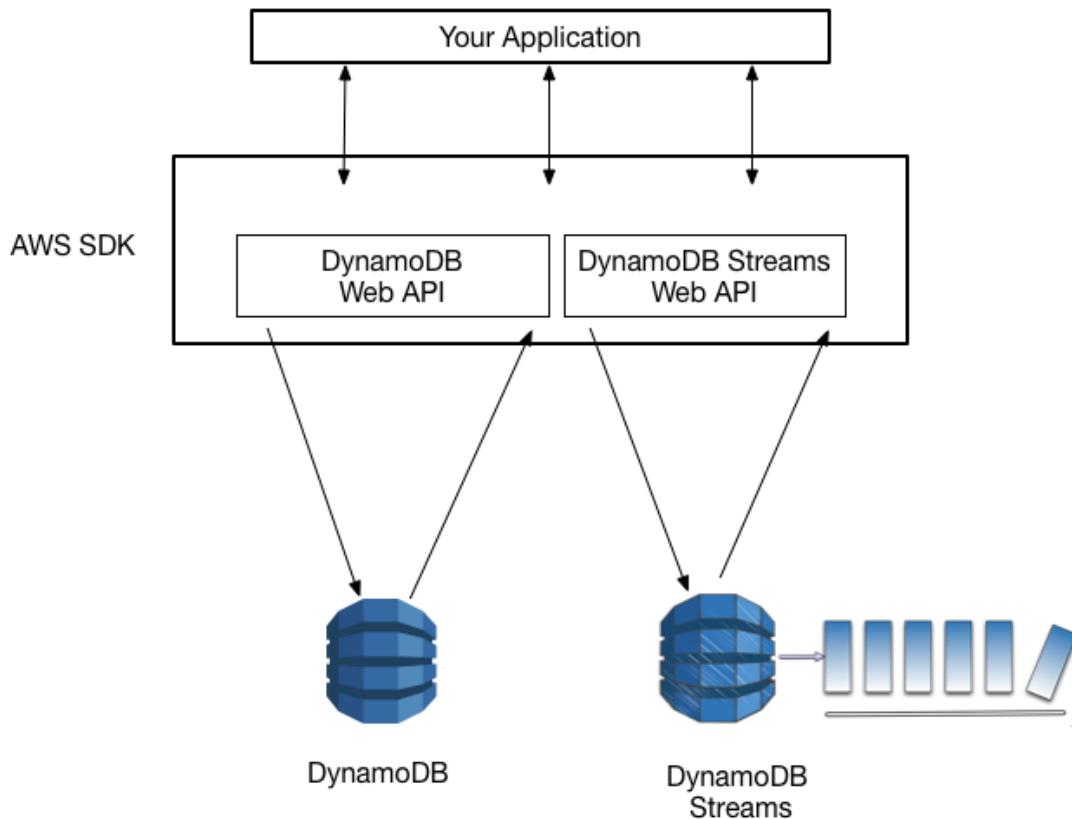
Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attribute(s) of the items that were modified. A *stream record* contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams guarantees 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.

Endpoints for DynamoDB Streams



AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application will need to access a DynamoDB endpoint. To read and process DynamoDB Streams records, your application will need to 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 will need to 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. 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. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. From the DynamoDB console dashboard, choose **Tables**
3. On the **Overview** tab, choose **Manage Stream**.
4. In the **Manage Stream** window, choose 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.
- When the settings are as you want them, choose **Enable**.
5. (Optional) To disable an existing stream, choose **Manage Stream** and then choose **Disable**.

You can also use the `createTable` or `UpdateTable` APIs 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, note that you will receive a `ResourceInUseException` if you attempt to enable a stream on a table that already has a stream, and you will receive a `ValidationException` if you attempt to disable a stream on a table which does not 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 will be created with a different stream descriptor.

Every stream is uniquely identified by an Amazon Resource Name (ARN). Here 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.

Reading and Processing a Stream

To read and process a stream, your application will need to 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.

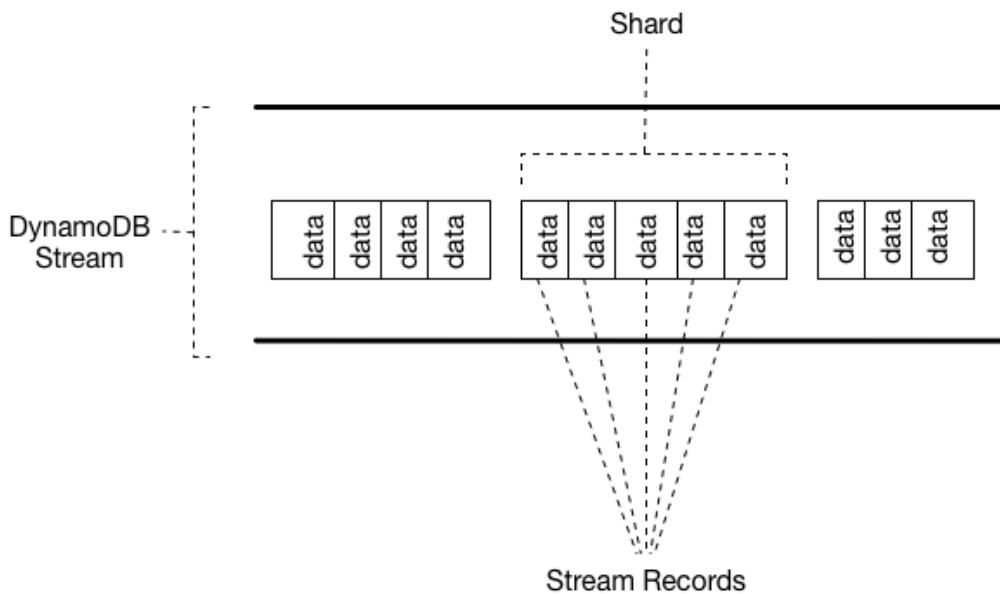
Stream records are organized into groups, or *shards*. Each shard acts as a container for multiple stream records, and contains information required for accessing and iterating through these records. The stream records within a shard are removed automatically after 24 hours.

Shards are ephemeral: They are created and deleted automatically, as needed. Any shard can also split into multiple new shards; this also occurs automatically. (Note that it is 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.

Because shards have a lineage (parent and children), applications must always process a parent shard before it processes a child shard. This will 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 will process the shards and stream records in the correct order, and automatically handle new or expired shards, as well as shards that split while the application is running. For more information, see [Using the DynamoDB Streams Kinesis Adapter to Process Stream Records \(p. 523\)](#).)

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, then DynamoDB Streams will *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 Amazon Resource Name (ARN) of the stream that you want to access.
- Determine which shard(s) in the stream contain the stream records that you are interested in.
- Access the shard(s) and retrieve the stream records that you want.

Note

No more than 2 processes at most should be reading from the same Streams shard at the same time. Having more than 2 readers per shard may 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 actions, including example requests and responses, go to 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 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 will continue to be readable for 24 hours. After this time, the data expires and the stream records are automatically deleted. Note that there is no mechanism for manually deleting an existing stream; you just need to 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 deleted by Time To Live by enabling Amazon DynamoDB Streams on the table and processing the Streams records of the expired items.

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"

The following JSON shows the relevant portion of a single Streams record.

```
"Records": [
    {
        ...
        "userIdentity":{
            "type":"Service",
            "principalId":"dynamodb.amazonaws.com"
        }
        ...
    }
]
```

Items deleted by other users will show the `principalId` of the account used to delete the items.

Using the DynamoDB Streams Kinesis Adapter to Process Stream Records

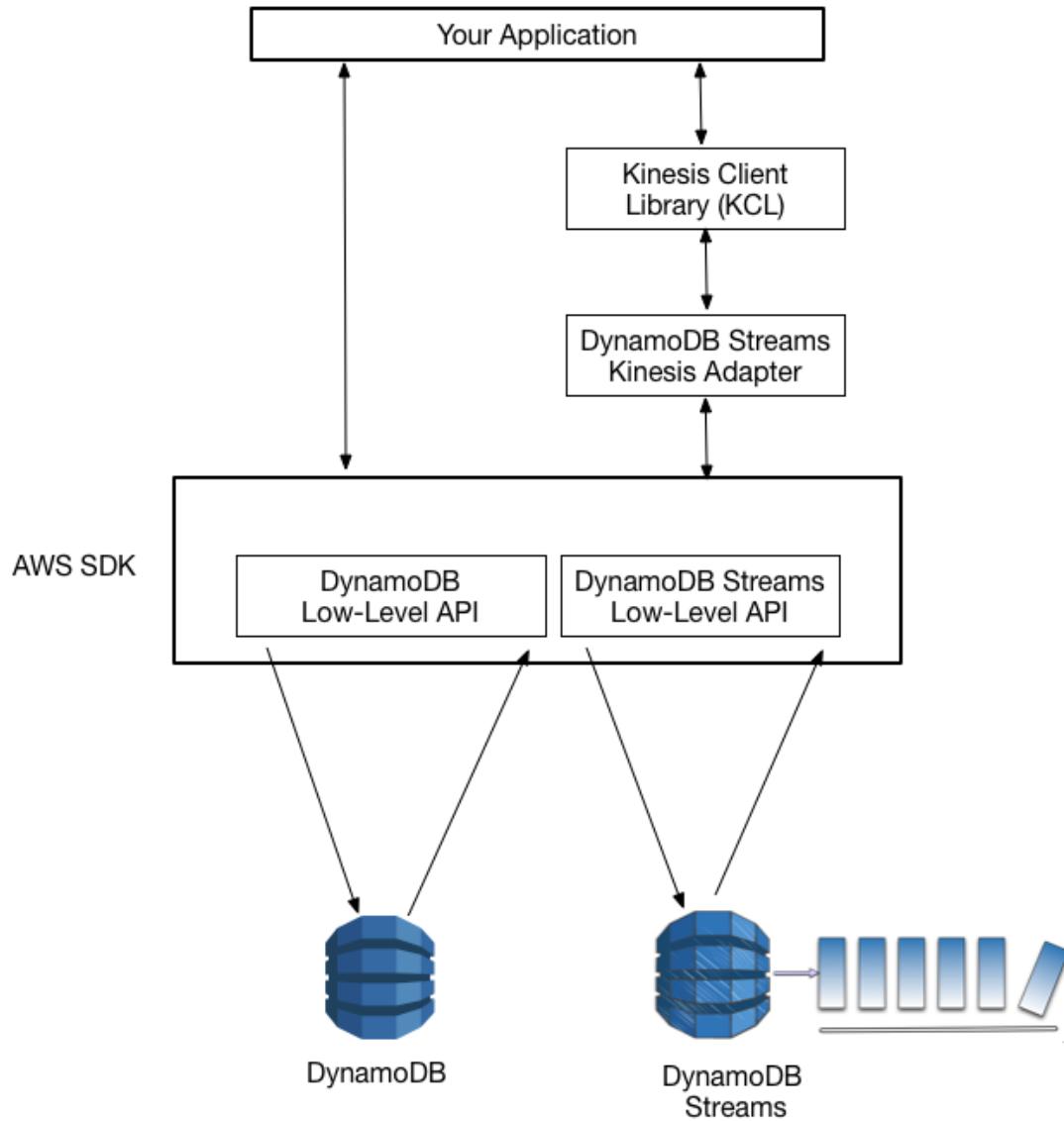
Using the Kinesis Adapter is the recommended way to consume Streams from DynamoDB.

The DynamoDB Streams API is intentionally similar to that of Kinesis 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` actions. (Even though these DynamoDB Streams actions are similar to their counterparts in Kinesis Streams, they are not 100% identical.)

You can write applications for Kinesis Streams using the Kinesis Client Library (KCL). The KCL simplifies coding by providing useful abstractions above the low-level Kinesis Streams API. For more information on the KCL, go to the [Amazon Kinesis Developer Guide](#).

As a DynamoDB Streams user, you can leverage 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 Streams interface, so that the KCL can be used for consuming and processing records from DynamoDB Streams.

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 above, go to [Developing Amazon Kinesis Consumers Using the Amazon Kinesis Client Library](#) in the *Amazon Kinesis Developer Guide*.

Walkthrough: DynamoDB Streams Kinesis Adapter

This section is a walkthrough of a Java application that uses the Kinesis Client Library and the DynamoDB Streams Kinesis Adapter. The application shows an example of data replication, where 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 \(p. 527\)](#).

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 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 \(p. 525\)](#)
- [Step 2: Generate Update Activity in Source Table \(p. 526\)](#)
- [Step 3: Process the Stream \(p. 526\)](#)
- [Step 4: Ensure Both Tables Have Identical Contents \(p. 527\)](#)
- [Step 5: Clean Up \(p. 527\)](#)
- [Complete Program: DynamoDB Streams Kinesis Adapter \(p. 527\)](#)

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`, meaning 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 code snippet shows the code 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 actions for writing the data. The following code snippet 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 leverage 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, go to [Developing Amazon Kinesis Consumers Using the Amazon Kinesis Client Library](#) in the [Amazon Kinesis Developer Guide](#).

The following code snippet 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) {
            e.printStackTrace();
        }
    }
}
```

Step 4: Ensure 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 code snippet 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 snippet.

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

Here is the complete Java program that performs the tasks described in this walkthrough. 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, make sure the client application has access to DynamoDB and CloudWatch using policies. For more information, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 614\)](#).

The source code consists of four .java files:

- StreamsAdapterDemo.java
- StreamsRecordProcessor.java
- StreamsRecordProcessorFactory.java
- StreamsAdapterDemoHelper.java

StreamsAdapterDemo.java

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
package com.amazonaws.codesamples;

import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWS CredentialsProvider;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
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.model.DeleteTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import com.amazonaws.services.dynamodbv2.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.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 AmazonDynamoDBStreamsAdapterClient adapterClient;
    private static AWS CredentialsProvider streamsCredentials;

    private static AmazonDynamoDB dynamoDBClient;
    private static AWS CredentialsProvider dynamoDBCredentials;

    private static AmazonCloudWatch cloudWatchClient;

    private static String serviceName = "dynamodb";
    private static String dynamodbEndpoint = "DYNAMODB_ENDPOINT_Goes_Here";
    private static String tablePrefix = "KCL-Demo";
    private static String streamArn;
```

```
/**  
 * @param args  
 */  
public static void main(String[] args) throws Exception {  
    System.out.println("Starting demo...");  
  
    String srcTable = tablePrefix + "-src";  
    String destTable = tablePrefix + "-dest";  
    streamsCredentials = new ProfileCredentialsProvider();  
    dynamoDBCredentials = new ProfileCredentialsProvider();  
    recordProcessorFactory = new StreamsRecordProcessorFactory(dynamoDBCredentials,  
dynamodbEndpoint, serviceName,  
    destTable);  
  
    adapterClient = new AmazonDynamoDBStreamsAdapterClient(streamsCredentials, new  
ClientConfiguration());  
  
    dynamoDBClient = AmazonDynamoDBClientBuilder.standard().build();  
  
    cloudWatchClient = AmazonCloudWatchClientBuilder.standard().build();  
  
    setUpTables();  
  
    workerConfig = new KinesisClientLibConfiguration("streams-adapter-demo", streamArn,  
streamsCredentials,  
        "streams-demo-  
worker").withMaxRecords(1000).withIdleTimeBetweenReadsInMillis(500)  
            .withInitialPositionInStream(InitialPositionInStream.TRIM_HORIZON);  
  
    System.out.println("Creating worker for stream: " + streamArn);  
    worker = new Worker(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

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
package com.amazonaws.codesamples;

import java.nio.charset.Charset;
import java.util.List;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.streamsadapter.model.RecordAdapter;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessor;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.model.Record;

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(String shardId) {
    checkpointCounter = 0;
}

@Override
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
checkpointer) {
    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) {
                    e.printStackTrace();
                }
            }
        }
    }
}

@Override
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
{
    if (reason == ShutdownReason.TERMINATE) {
        try {
            checkpointer.checkpoint();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

StreamsRecordProcessorFactory.java

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
package com.amazonaws.codesamples;  
  
import com.amazonaws.auth.AWS CredentialsProvider;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessor;  
import com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorFactory;  
  
public class StreamsRecordProcessorFactory implements IRecordProcessorFactory {  
  
    private final String tableName;  
  
    public StreamsRecordProcessorFactory(AWS CredentialsProvider dynamoDBCredentials, String dynamoDBEndpoint,  
                                         String serviceName, String tableName) {  
        this.tableName = tableName;  
    }  
  
    @Override  
    public IRecordProcessor createProcessor() {  
        AmazonDynamoDB dynamoDBClient = AmazonDynamoDBClientBuilder.standard().build();  
        return new StreamsRecordProcessor(dynamoDBClient, tableName);  
    }  
}
```

StreamsAdapterDemoHelper.java

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
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);
}
```

Walkthrough: DynamoDB Streams Low-Level API

This section is a walkthrough of a Java program that shows DynamoDB Streams in action. For the source code, see [Complete Program: Low-Level DynamoDB Streams API \(p. 537\)](#).

The program does the following:

1. Creates a DynamoDB table with a stream enabled.
2. Describes the stream settings for this table.
3. Modify data in the table.
4. Describe the shards in the stream.
5. Read the stream records from the shards.
6. Clean up.

Note

This code does not handle all exceptions, and will not work reliably under high-traffic conditions. The recommended way to consume Stream records from DynamoDB is through the Kinesis Adapter using the Kinesis Client Library (KCL), as described in [Using the DynamoDB Streams Kinesis Adapter to Process Stream Records \(p. 523\)](#).

These steps are described in the following sections, and the complete application is shown at the end of the walkthrough.

Topics

- [Step 1: Create a Table with a Stream Enabled \(p. 535\)](#)
- [Step 2: Describe the Stream Settings For The Table \(p. 535\)](#)
- [Step 3: Modify data in the table \(p. 535\)](#)

- [Step 4: Describe the Shards in the Stream \(p. 536\)](#)
- [Step 5: Read the Stream Records \(p. 536\)](#)
- [Step 6: Clean Up \(p. 537\)](#)
- [Complete Program: Low-Level DynamoDB Streams API \(p. 537\)](#)

Step 1: Create a Table with a Stream Enabled

The first step is to create a table in DynamoDB, as in the following code snippet. The table has a stream enabled, which will capture the NEW_AND_OLD_IMAGES of each item that is modified.

```
ArrayList<AttributeDefinition> attributeDefinitions =  
    new ArrayList<AttributeDefinition>();  
  
attributeDefinitions.add(new AttributeDefinition()  
    .withAttributeName("Id")  
    .withAttributeType("N"));  
  
ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();  
keySchema.add(new KeySchemaElement()  
    .withAttributeName("Id")  
    .withKeyType(KeyType.HASH)); //Partition key  
  
StreamSpecification streamSpecification = new StreamSpecification();  
streamSpecification.setStreamEnabled(true);  
streamSpecification.setStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES);  
  
CreateTableRequest createTableRequest = new CreateTableRequest()  
    .withTableName(tableName)  
    .withKeySchema(keySchema)  
    .withAttributeDefinitions(attributeDefinitions)  
    .withProvisionedThroughput(new ProvisionedThroughput()  
        .withReadCapacityUnits(1L)  
        .withWriteCapacityUnits(1L))  
    .withStreamSpecification(streamSpecification);
```

Step 2: Describe the Stream Settings For The Table

The `DescribeTable` API lets you view the current stream settings for a table. The following code snippet helps confirm that the stream is enabled, and that it will capture the correct data.

```
DescribeTableResult describeTableResult = dynamoDBClient.describeTable(tableName);  
  
String myStreamArn = describeTableResult.getTable().getLatestStreamArn();  
StreamSpecification myStreamSpec =  
    describeTableResult.getTable().getStreamSpecification();  
  
System.out.println("Current stream ARN for " + tableName + ":" + myStreamArn);  
System.out.println("Stream enabled: " + myStreamSpec.getStreamEnabled());  
System.out.println("Update view type: " + myStreamSpec.getStreamViewType());
```

Step 3: Modify data in the table

The next step is to make some changes to the data in the table. The following code snippet adds a new item to the table, updates an attribute in that item, and then deletes the item.

```
// Add a new item
```

```

int numChanges = 0;
System.out.println("Making some changes to table data");
Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
item.put("Id", new AttributeValue().withN("101"));
item.put("Message", new AttributeValue().withS("New item!"));
dynamoDBClient.putItem(tableName, item);
numChanges++;

// Update the item

Map<String, AttributeValue> key = new HashMap<String, AttributeValue>();
key.put("Id", new AttributeValue().withN("101"));
Map<String, AttributeValueUpdate> attributeUpdates =
    new HashMap<String, AttributeValueUpdate>();
attributeUpdates.put("Message", new AttributeValueUpdate()
    .withAction(AttributeAction.PUT)
    .WithValue(new AttributeValue().withS("This item has changed")));
dynamoDBClient.updateItem(tableName, key, attributeUpdates);
numChanges++;

// Delete the item

dynamoDBClient.deleteItem(tableName, key);
numChanges++;

```

Step 4: Describe the Shards in the Stream

The data modifications in the table will result in stream records being written to the table's stream.

In [Step 2: Describe the Stream Settings For The Table \(p. 535\)](#), we determined the current stream ARN and assigned it to the variable `myStreamArn`. We can use this with the `DescribeStream` action to obtain the shards in the stream.

Because we did not modify very much data in the DynamoDB table, there will only be one shard in the list. The following code snippet shows how to obtain this information.

```

DescribeStreamResult describeStreamResult =
    streamsClient.describeStream(new DescribeStreamRequest()
        .withStreamArn(myStreamArn));
String streamArn =
    describeStreamResult.getStreamDescription().getStreamArn();
List<Shard> shards =
    describeStreamResult.getStreamDescription().getShards();

```

Step 5: Read the Stream Records

For each shard in the list, we obtain a shard iterator, and then use the iterator to obtain the stream records and print them.

The following code snippet uses a loop to process the shard list, even though there is only one shard.

```

for (Shard shard : shards) {
    String shardId = shard.getShardId();
    System.out.println(
        "Processing " + shardId + " from stream " + streamArn);

    // Get an iterator for the current shard

    GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest()

```

```

        .withStreamArn(myStreamArn)
        .withShardId(shardId)
        .withShardIteratorType(ShardIteratorType.TRIM_HORIZON);
    GetShardIteratorResult getShardIteratorResult =
        streamsClient.getShardIterator(getShardIteratorRequest);
    String nextItr = getShardIteratorResult.getShardIterator();

    while (nextItr != null && numChanges > 0) {

        // Use the iterator to read the data records from the shard

        GetRecordsResult getRecordsResult =
            streamsClient.getRecords(new GetRecordsRequest().
                withShardIterator(nextItr));
        List<Record> records = getRecordsResult.getRecords();
        System.out.println("Getting records...");
        for (Record record : records) {
            System.out.println(record);
            numChanges--;
        }
        nextItr = getRecordsResult.getNextShardIterator();
    }
}

```

Step 6: Clean Up

The demo is complete, so we can delete the table. Note that the stream associated with this table will remain available for reading, even though the table is deleted. The stream will be automatically deleted after 24 hours.

```
dynamoDBClient.deleteTable(tableName);
```

Complete Program: Low-Level DynamoDB Streams API

Here is a complete Java program that performs the tasks described in this walkthrough. When you run it, you will see each stream record in its entirety:

```

Issuing CreateTable request for TestTableForStreams
Waiting for TestTableForStreams to be created...
Current stream ARN for TestTableForStreams: arn:aws:dynamodb:us-west-2:111122223333:table/TestTableForStreams/stream/2017-05-01T16:04:24.300
Stream enabled: true
Update view type: NEW_AND_OLD_IMAGES
Making some changes to table data
Processing shardId-00000001493654674152-5245c47a from stream arn:aws:dynamodb:us-west-2:111122223333:table/TestTableForStreams/stream/2017-05-01T16:04:24.300
Getting records...
{EventID: bb6acf1d94d80f579dfa18c05b8a7c8d,EventName: INSERT,EventVersion: 1.1,EventSource: aws:dynamodb,AwsRegion: us-west-2,Dynamodb: {ApproximateCreationDateTime: Mon May 01 09:04:00 PDT 2017,Keys: {Id={N: 101,}},NewImage: {Message={S: New item!,},Id={N: 101,}},SequenceNumber: 100000000001134015798,SizeBytes: 26,StreamViewType: NEW_AND_OLD_IMAGES,}}
{EventID: ea9cebe39d92adbf23089a179215fa5f,EventName: MODIFY,EventVersion: 1.1,EventSource: aws:dynamodb,AwsRegion: us-west-2,Dynamodb: {ApproximateCreationDateTime: Mon May 01 09:04:00 PDT 2017,Keys: {Id={N: 101,}},NewImage: {Message={S: This item has changed!,},Id={N: 101,}},OldImage: {Message={S: New item!,},Id={N: 101,}},SequenceNumber: 200000000001134015828,SizeBytes: 59,StreamViewType: NEW_AND_OLD_IMAGES,}}
{EventID: fc3fe36dc8d50dc0c4b513922d8c1bfd,EventName: REMOVE,EventVersion: 1.1,EventSource: aws:dynamodb,AwsRegion: us-west-2,Dynamodb: {ApproximateCreationDateTime: Mon May 01 09:04:00 PDT 2017,Keys: {Id={N: 101,}},OldImage: {Message={S: This item has changed!,}}}

```

```
Id={N: 101,}},SequenceNumber: 300000000001134015829,SizeBytes: 38,StreamViewType:  
NEW_AND_OLD_IMAGES},}  
Deleting the table...  
Demo complete
```

Example

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
// Licensed under the Apache License, Version 2.0.  
package com.amazonaws.codesamples;  
  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
import com.amazonaws.AmazonClientException;  
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().build();  
  
        AmazonDynamoDBStreams streamsClient =  
        AmazonDynamoDBStreamsClientBuilder.standard().build();  
  
        // Create the table  
        String tableName = "TestTableForStreams";  
        ArrayList<AttributeDefinition> attributeDefinitions = new  
        ArrayList<AttributeDefinition>();  
  
        attributeDefinitions.add(new  
        AttributeDefinition().withAttributeName("Id").withAttributeType("N"));  
  
        ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();  
        keySchema.add(new  
        KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition key
```

```

StreamSpecification streamSpecification = new StreamSpecification();
streamSpecification.setStreamEnabled(true);
streamSpecification.setStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES);

CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
    .withKeySchema(keySchema).withAttributeDefinitions(attributeDefinitions)
    .withProvisionedThroughput(new
ProvisionedThroughput().withReadCapacityUnits(1L).withWriteCapacityUnits(1L))
    .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();
}

// Determine the Streams settings for the table

DescribeTableResult describeTableResult = dynamoDBClient.describeTable(tableName);

String myStreamArn = describeTableResult.getTable().getLatestStreamArn();
StreamSpecification myStreamSpec =
describeTableResult.getTable().getStreamSpecification();

System.out.println("Current stream ARN for " + tableName + ": " + myStreamArn);
System.out.println("Stream enabled: " + myStreamSpec.getStreamEnabled());
System.out.println("Update view type: " + myStreamSpec.getStreamViewType());

// Add a new item

int numChanges = 0;
System.out.println("Making some changes to table data");
Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
item.put("Id", new AttributeValue().withN("101"));
item.put("Message", new AttributeValue().withS("New item!"));
dynamoDBClient.putItem(tableName, item);
numChanges++;

// Update the item

Map<String, AttributeValue> key = new HashMap<String, AttributeValue>();
key.put("Id", new AttributeValue().withN("101"));
Map<String, AttributeValueUpdate> attributeUpdates = new HashMap<String,
AttributeValueUpdate>();
attributeUpdates.put("Message", new
AttributeValueUpdate().withAction(AttributeAction.PUT)
    .WithValue(new AttributeValue().withS("This item has changed")));
dynamoDBClient.updateItem(tableName, key, attributeUpdates);
numChanges++;

// Delete the item

dynamoDBClient.deleteItem(tableName, key);
numChanges++;

// Get the shards in the stream

DescribeStreamResult describeStreamResult = streamsClient
    .describeStream(new DescribeStreamRequest().withStreamArn(myStreamArn));
String streamArn = describeStreamResult.getStreamDescription().getStreamArn();
List<Shard> shards = describeStreamResult.getStreamDescription().getShards();

```

```

// Process each shard

for (Shard shard : shards) {
    String shardId = shard.getShardId();
    System.out.println("Processing " + shardId + " from stream " + streamArn);

    // Get an iterator for the current shard

    GetShardIteratorRequest getShardIteratorRequest = new
GetShardIteratorRequest().withStreamArn(myStreamArn)

.withShardId(shardId).withShardIteratorType(ShardIteratorType.TRIM_HORIZON);
    GetShardIteratorResult getShardIteratorResult =
streamsClient.getShardIterator(getShardIteratorRequest);
    String nextItr = getShardIteratorResult.getShardIterator();

    while (nextItr != null && numChanges > 0) {

        // Use the iterator to read the data records from the shard

        GetRecordsResult getRecordsResult = streamsClient
            .getRecords(new GetRecordsRequest().withShardIterator(nextItr));
        List<Record> records = getRecordsResult.getRecords();
        System.out.println("Getting records...");
        for (Record record : records) {
            System.out.println(record);
            numChanges--;
        }
        nextItr = getRecordsResult.getNextShardIterator();
    }

    // Delete the table

    System.out.println("Deleting the table...");
    dynamoDBClient.deleteTable(tableName);

    System.out.println("Demo complete");
}
}

```

Cross-Region Replication

Important

AWS previously provided a cross-region replication solution based on AWS CloudFormation. This solution has now been deprecated in favor of an open source command line tool. For more information, please refer to the detailed instructions on GitHub:

- <https://github.com/aws-labs/dynamodb-cross-region-library/blob/master/README.md>

The DynamoDB cross-region replication solution uses the Amazon DynamoDB Cross-Region Replication Library. This library uses DynamoDB Streams to keep DynamoDB tables in sync across multiple regions in near real time. When you write to a DynamoDB table in one region, those changes are automatically propagated by the Cross-Region Replication Library to your tables in other regions.

You can leverage the Cross-Region Replication Library in your own applications, to build your own replication solutions with DynamoDB Streams. For more information, and to download the source code, go to the following GitHub repository:

- <https://github.com/awslabs/dynamodb-cross-region-library>

DynamoDB Streams and AWS Lambda Triggers

Topics

- [Tutorial: Processing New Items in a DynamoDB Table \(p. 541\)](#)
- [Best Practices \(p. 548\)](#)

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 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 you specify, such as sending a notification or initiating a workflow. For example, you can write a Lambda function to simply copy each stream record to persistent storage, such as Amazon Simple Storage Service (Amazon S3), to create a permanent audit trail of write activity in your table. Or suppose 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. (The function would simply ignore any stream records that are not updates to `GameScores` or that do not modify the `topScore` attribute.)

For more information about AWS Lambda, see the [AWS Lambda Developer Guide](#).

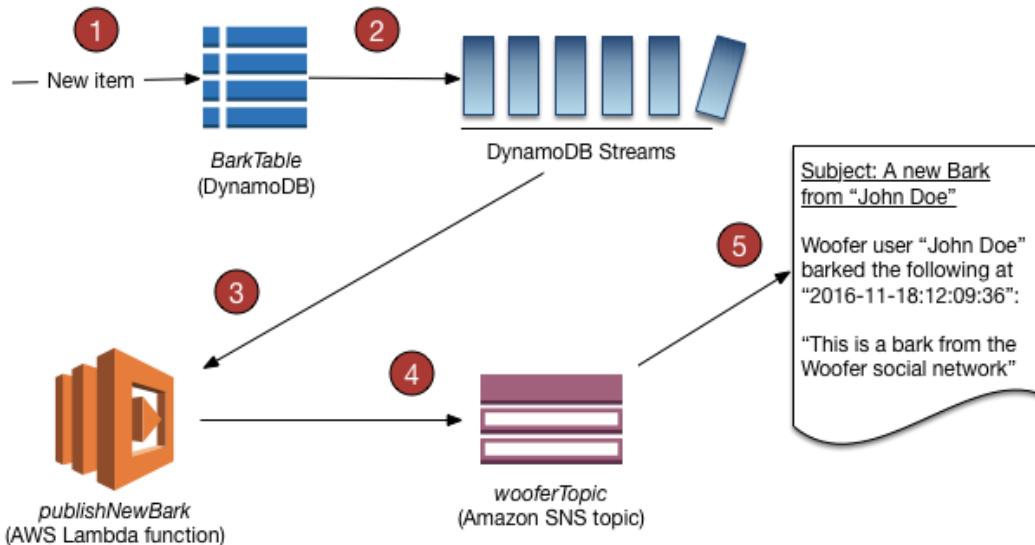
Tutorial: Processing New Items in a DynamoDB Table

Topics

- [Step 1: Create a DynamoDB Table With a Stream Enabled \(p. 542\)](#)
- [Step 2: Create a Lambda Execution Role \(p. 543\)](#)
- [Step 3: Create an Amazon SNS Topic \(p. 544\)](#)
- [Step 4: Create and Test a Lambda Function \(p. 545\)](#)
- [Step 5: Create and Test a Trigger \(p. 547\)](#)

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. If you have not done so already, follow the instructions in the [AWS Command Line Interface User Guide](#) install and configure the AWS CLI.

Step 1: Create a DynamoDB Table With a Stream Enabled

In this step, you will 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 will have a stream enabled. Later in this tutorial, you will create a trigger by associating an AWS Lambda function with the stream.

1. Type 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 will need them for the other steps in this tutorial.

Step 2: Create a Lambda Execution Role

In this step, you will create an IAM role (*WooferLambdaRole*) and assign permissions to it. This role will be used by the Lambda function that you will create in [Step 4: Create and Test a Lambda Function \(p. 545\)](#).

You will also create a policy for the role. The policy will contain all of the permissions that the Lambda function will need 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. Type 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": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:region:accountID:function:publishNewBark*"
        },
        {
            "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:table/BarkTable/stream/*"
    },
    {
        "Effect": "Allow",
        "Action": [
            "sns:Publish"
        ],
        "Resource": [
            "*"
        ]
    }
]
}

```

The policy has four statements, to allow *WooferLambdaRole* to do the following:

- Execute a Lambda function (*publishNewBark*). You will create the function later in this tutorial.
 - Access CloudWatch Logs. The Lambda function will write diagnostics to CloudWatch Logs at runtime.
 - Read data from the DynamoDB stream for *BarkTable*.
 - Publish messages to Amazon SNS.
4. Type 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 will create an Amazon SNS topic (*wooferTopic*) and subscribe an email address to it. Your Lambda function will use this topic to publish new barks from Woofer users.

1. Type the following command to create a new Amazon SNS topic:

```
aws sns create-topic --name wooferTopic
```

2. Type 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 will send a confirmation message to your email address. Click the **Confirm subscription** link in that message to complete the subscription process.

Step 4: Create and Test a Lambda Function

In this step, you will 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: 'Woofie user ' + who + ' barked the following at ' + when + ': \n\n ' + what,
                TopicArn: 'arn:aws:sns:region:accountID:woofieTopic'
            };
            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 type the following command to do this:

```
zip publishNewBark.zip publishNewBark.js
```

3. When you create the Lambda function, you specify the ARN for *WooferLambdaRole*, which you created in [Step 2: Create a Lambda Execution Role \(p. 543\)](#). Type 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/WoofeLambdaRole"
...

```

Now type the following command to create the Lambda function. (Replace `roleARN` with the ARN for `WoofeLambdaRole`.)

```

aws lambda create-function \
--region us-east-1 \
--function-name publishNewBark \
--zip-file fileb://publishNewBark.zip \
--role roleARN \
--handler publishNewBark.handler \
--timeout 5 \
--runtime nodejs4.3

```

4. Now you will test `publishNewBark` to verify that it works. To do this, you will provide input that resembles a real record from DynamoDB Streams.

Create a file named `payload.json` with the following contents.

```

{
    "Records": [
        {
            "eventID": "7de3041dd709b024af6f29e4fa13d34c",
            "eventName": "INSERT",
            "eventVersion": "1.1",
            "eventSource": "aws:dynamodb",
            "awsRegion": "us-west-2",
            "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": "1302160000000001596893679",
                "SizeBytes": 112,
                "StreamViewType": "NEW_IMAGE"
            },
            "eventSourceARN": "arn:aws:dynamodb:us-east-1:123456789012:table/BarkTable/
stream/2016-11-16T20:42:48.104"
        }
    ]
}

```

Type the following command to test the *publishNewBark* function:

```
aws lambda invoke --function-name publishNewBark --payload file://payload.json  
output.txt
```

If the test was successful, you will see the following output:

```
{  
    "StatusCode": 200  
}
```

In addition, the `output.txt` file will contain this 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 \(p. 545\)](#), you tested the Lambda function to ensure that it ran correctly. In this step, you will create a *trigger* by associating the Lambda function (*publishNewBark*) with an event source (the *BarkTable* stream).

1. When you create the trigger, you will need to specify the ARN for the *BarkTable* stream. Type 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  
...
```

2. Type the following command to create the trigger. (Replace `streamARN` with the actual stream ARN.)

```
aws lambda create-event-source-mapping \  
    --region us-east-1 \  
    --function-name publishNewBark \  
    --event-source streamARN \  
    --batch-size 1 \  
    --starting-position TRIM_HORIZON
```

API Version 2012-08-10

3. You will now test the trigger. Type 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.

4. Go to 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.

Best Practices

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 executed 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 \(p. 545\)](#) 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 executes the handler for the first time. The Lambda function shown in [Step 4: Create and Test a Lambda Function \(p. 545\)](#) 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 executes, AWS Lambda may 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. (Note that 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 will allow 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.

For more information, see [Best Practices for Working with AWS Lambda Functions](#) in the AWS Lambda Developer Guide.

In-Memory Acceleration with DAX

Topics

- [Use Cases for DAX \(p. 549\)](#)
- [Usage Notes \(p. 550\)](#)
- [Concepts \(p. 550\)](#)
- [DAX Cluster Components \(p. 553\)](#)
- [Creating a DAX Cluster \(p. 557\)](#)
- [DAX and DynamoDB Consistency Models \(p. 564\)](#)
- [Using the DAX Client in an Application \(p. 570\)](#)
- [Managing DAX Clusters \(p. 590\)](#)
- [DAX Access Control \(p. 596\)](#)
- [DAX API Reference \(p. 608\)](#)

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 Amazon DynamoDB, and thus 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 over-provision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

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:

- 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 e-commerce 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 data 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). Alternatively, you can offload the activity from your application to a DAX cluster, and reduce the amount of read capacity units you'd 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 of the read capacity in a DynamoDB table, which 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:

- Applications that require strongly consistent reads (or cannot tolerate eventually consistent reads).
- Applications that do not require microsecond response times for reads, or that 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.

Usage Notes

- DAX is available in the following AWS regions:
 - US East (N. Virginia)
 - US West (N. California)
 - US West (Oregon)
 - EU (Ireland)
 - Asia Pacific (Tokyo)
- DAX supports applications written in Java or Node.js, using the DAX clients for those programming languages.
- DAX does not support Transport Layer Security (TLS).
- DAX is only available for the EC2-VPC platform. (There is no support for the EC2-Classic platform.)

Concepts

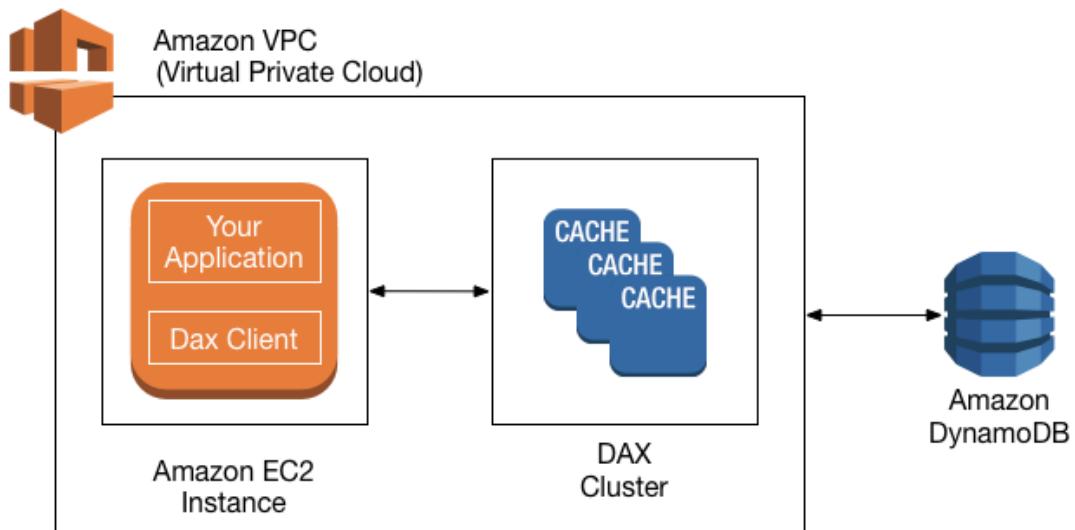
DAX is designed to run within an Amazon Virtual Private Cloud environment (Amazon VPC). The Amazon VPC service defines a virtual network that closely resembles a traditional data center. With an Amazon VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security 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 2013-12-04, 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 your default VPC without having to perform any additional configuration steps.

For more information, see [Your Default VPC and 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 will run within your default VPC.

To run your application, you launch an Amazon EC2 instance into your Amazon VPC, and 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.

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 \(p. 553\)](#).

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, so that incoming requests are evenly distributed across all of the nodes in the cluster.

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 attempts 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 \(p. 554\)](#).

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 will need to access DynamoDB directly rather than using DAX.

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 attempts 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 setting (TTL), 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, so DAX will send 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 \(p. 590\)](#).)

DAX also maintains a least recently used list (LRU) for the item cache. The LRU list keeps track of 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 will evict older items (even if they have not expired yet) to make room for new items. The LRU algorithm is always enabled for the item cache, and is not user-configurable.

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 attempts 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 item 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 \(p. 590\)](#).)

DAX also maintains a least recently used list (LRU) for the query cache. The LRU list keeps track of 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 will evict 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.

DAX Cluster Components

A DAX cluster is comprised of AWS infrastructure components. This section describes these components, and how they work together.

Topics

- [Nodes \(p. 553\)](#)
- [Clusters \(p. 554\)](#)
- [Regions and Availability Zones \(p. 555\)](#)
- [Parameter Groups \(p. 555\)](#)
- [Security Groups \(p. 555\)](#)
- [Cluster ARN \(p. 555\)](#)
- [Cluster Endpoint \(p. 555\)](#)
- [Node Endpoints \(p. 556\)](#)
- [Subnet Groups \(p. 556\)](#)
- [Events \(p. 556\)](#)
- [Maintenance Window \(p. 556\)](#)

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 will increase the overall read throughput of the cluster.
- By using a larger node type. Larger node types provide more capacity and can increase throughput. (Note that you must create a new cluster with the new node type.)

Valid node types are as follows:

- `dax.r3.large`
- `dax.r3.xlarge`
- `dax.r3.2xlarge`
- `dax.r3.4xlarge`
- `dax.r3.8xlarge`

Every node within a cluster is of the same node type, and runs the same DAX caching software.

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.

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 do not 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 will spread the load evenly across all of 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 will still be able to 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 should become unavailable.

A DAX cluster can support up to ten nodes per cluster (the primary node, plus a maximum of nine read replicas).

Important

A DAX cluster can be deployed with a single node. While this configuration might be acceptable for development or test workloads, a one-node cluster is not fault-tolerant and we do not

recommend a one-node cluster for production use. If this single node encounters software or hardware errors, the cluster can become unavailable or lose cached data.

For production usage, we strongly recommend using DAX with at least three nodes (the primary node and at least two read replicas), where the nodes are placed in different Availability Zones.

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, you need to ensure that you launch your DAX cluster in the correct region. If you have DynamoDB tables in other regions, you will need to 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 are able to achieve the greatest possible fault tolerance.

Important

Do not place all of your cluster's nodes in a single Availability Zone. In this configuration, your DAX cluster will become unavailable in case of an Availability Zone failure.

For production usage, we strongly recommend using DAX with at least three nodes (the primary node and at least two read replicas), where the nodes are placed in different Availability Zones.

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, thereby guaranteeing that all of the nodes in that cluster are configured in exactly the same way.

Security Groups

A DAX cluster runs in an Amazon VPC environment—a virtual network that is dedicated to your AWS account, and is isolated from other Amazon 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 ingress rule to your security group, your applications running within your VPC can access the DAX cluster.

Cluster ARN

Every DAX cluster is assigned an Amazon Resource Identifier (ARN). The ARN format is:

```
arn:aws:dax:region:accountID:cache/clusterName
```

You use the cluster ARN in an IAM policy to define permissions for DAX API actions. For more information, see [DAX Access Control \(p. 596\)](#).

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 host names and port numbers of individual nodes in the cluster. Your application automatically "knows" all of the nodes in the cluster, even if you add or remove read replicas.

Here is an example of a cluster endpoint:

```
myDAXcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111
```

Node Endpoints

Each of the individual nodes in a DAX cluster has its own host name and port number. Here is an example of a node endpoint:

```
myDAXcluster-a.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111
```

Your application can access a node directly, 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 keeping 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 Virtual Private Cloud (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 Virtual Private Cloud (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 SNS topic, so that 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 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
eu-west-1	EU (Ireland) Region	22:00–06:00 UTC
us-east-1	US East (N. Virginia) Region	03:00–11:00 UTC

Region Code	Region Name	Maintenance Window
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 you have requested should occur.

Creating a DAX Cluster

This section walks you through first-time setup and usage of DAX in your default 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 have created your DAX cluster, you will be able to access it from an Amazon EC2 instance running in the same Amazon VPC. You will then be able to use your DAX cluster with an application program. (For more information, see [Using the DAX Client in an Application \(p. 570\)](#).)

Topics

- [Creating a DAX Service Role \(p. 557\)](#)
- [AWS CLI \(p. 558\)](#)
- [AWS Management Console \(p. 562\)](#)

Creating a DAX Service Role

In order for your DAX cluster to access DynamoDB tables on your behalf, you will need to create a service role. A *service role* is an IAM role that authorizes an AWS service to act on your behalf. The service role will allow DAX to access your DynamoDB tables, as if you were accessing those tables yourself. You need to create the service role before you can create the DAX cluster.

If you are using the AWS Management 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 [Step 2: Create a DAX Cluster \(p. 563\)](#) in [AWS Management Console \(p. 562\)](#).

If you are using the AWS CLI, you will need to specify a DAX service role that you have created previously. Otherwise, you will need to create a new service role beforehand. For more information, see [Step 1: Create a Service Role for DAX \(p. 559\)](#) in [AWS CLI \(p. 558\)](#).

Permissions Required for Service Role Creation

The AWS-managed *AdministratorAccess* policy provides all of the permissions needed for creating a DAX cluster, and for creating a service role. If your IAM user has *AdministratorAccess* attached, then no further action is needed.

Otherwise, you will need to add the following permissions to your IAM policy so that your IAM user can create the service role:

- `iam:CreateRole`
- `iam:CreatePolicy`

- `iam:AttachRolePolicy`
- `iam:PassRole`

You should attach these permissions to the user that is attempting to perform the action.

Note

The `iam:CreateRole`, `iam:CreatePolicy`, `iam:AttachPolicy` 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: 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 encounter error messages. The following table shows these messages, and how to correct the problems.

If you see this error message...	Do the following:
User: arn:aws:iam:: <code>accountID</code> :user/ <code>userName</code> is not authorized to perform: <code>iam:CreateRole</code> on resource: arn:aws:iam:: <code>accountID</code> :role/service-role/ <code>roleName</code>	Add <code>iam:CreateRole</code> to your user policy.
User: arn:aws:iam:: <code>accountID</code> :user/ <code>userName</code> is not authorized to perform: <code>iam:CreatePolicy</code> on resource: policy <code>policyName</code>	Add <code>iam:CreatePolicy</code> to your user policy.
User: arn:aws:iam:: <code>accountID</code> :user/ <code>userName</code> is not authorized to perform: <code>iam:AttachRolePolicy</code> on resource: role <code>daxServiceRole</code>	Add <code>iam:AttachRolePolicy</code> to your user policy.

For more information about IAM policies required for DAX cluster administration, see [DAX Access Control \(p. 596\)](#).

AWS CLI

Topics

- [Step 1: Create a Service Role for DAX \(p. 559\)](#)
- [Step 2: Create a Subnet Group \(p. 560\)](#)
- [Step 3: Create a DAX Cluster \(p. 561\)](#)
- [Step 4: Configure Security Group Inbound Rules \(p. 561\)](#)

This section describes how to create a DAX cluster using the AWS Command Line Interface (AWS CLI). If you have not already done so, you will need to install and configure the AWS CLI. To do this, go to the AWS Command Line Interface User Guide and follow these instructions:

- [Installing the AWS CLI](#)
- [Configuring the AWS CLI](#)

Important

To manage DAX clusters with the AWS CLI, please install or upgrade to version 1.11.110 or higher.

Note

All of the AWS CLI examples use the `us-west-2` region and fictitious account IDs.

Step 1: Create a Service Role for DAX

Before you can create a DAX cluster, you will need to create a service role for it. A *service role* is an IAM role that authorizes an AWS service to act on your behalf. The service role will allow DAX to access your DynamoDB tables, as if you were accessing those tables yourself.

In this step, you will create an IAM policy, and then attach that policy to an IAM role. This will enable you to assign the role to a DAX cluster so that it can perform DynamoDB operations on your behalf.

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 DAXServiceRole \  
    --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:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:accountID:*"  
            ]  
        }  
    ]  
}
```

Replace `accountID` with your AWS account ID. To find your AWS account ID, go to the upper right-hand portion of the AWS Management Console and choose your login ID. Your AWS account ID appears in the drop-down menu. (In the ARN, `accountID` must be a twelve-digit number. Do not use hyphens or any other punctuation.)

4. Create an IAM policy for the service role:

```
aws iam create-policy \  
  --policy-name DAXServicePolicy \  
  --policy-document file://service-role-policy.json
```

In the output, take note of the ARN for the policy you created. For example:

```
arn:aws:iam::123456789012:policy/DAXServicePolicy
```

5. Attach the policy to the service role:

```
aws iam attach-role-policy \  
  --role-name DAXServiceRole \  
  --policy-arn arn
```

Replace *arn* with the actual role ARN from the previous step.

Step 2: Create a Subnet Group

Note

If you have 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 2013-12-04, then you already have a default VPC in each AWS region. For more information, see [Your Default VPC and 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 will be deployed to the subnets within the subnet group.

1. To determine the identifier for your default VPC, type the following command:

```
aws ec2 describe-vpcs
```

In the output, take note of the identifier for your default VPC. For example:

```
vpc-12345678
```

2. Determine the subnets IDs associated with your default VPC:

```
aws ec2 describe-subnets \  
  --filters "Name=vpc-id,Values=vpcID" \  
  --query "Subnets[*].SubnetId"
```

Replace *vpcID* with your actual VPC ID. For example: vpc-12345678

In the output, take note of 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
```

Step 3: Create a DAX Cluster

Follow this procedure to create a DAX cluster in your default Amazon VPC:

1. Get the Amazon Resource Name (ARN) for your service role:

```
aws iam get-role \
--role-name DAXServiceRole \
--query "Role.Arn" --output text
```

In the output, take note of the service role ARN. For example:

```
arn:aws:iam::123456789012:role/DaxServiceRole
```

2. You are now ready to create your DAX cluster:

```
aws dax create-cluster \
--cluster-name mydaxcluster \
--node-type dax.r3.large \
--replication-factor 3 \
--iam-role-arn roleARN \
--subnet-group my-subnet-group \
--region us-west-2
```

Replace *roleARN* with the ARN from the previous step.

All of the nodes in the cluster will be of type *dax.r3.large* (--node-type). There will be three nodes (--replication-factor)—one primary node and two replicas.

To view the cluster status, type the following command:

```
aws dax describe-clusters
```

The status is shown in the output. For example: "Status": "creating"

Note

Creating the cluster will take several minutes. When the cluster is ready, its status changes to available.

In the meantime, you can proceed to [Step 4: Configure Security Group Inbound Rules \(p. 561\)](#) and follow the instructions there.

Step 4: Configure Security Group Inbound Rules

The nodes in your DAX cluster use the default security group for your Amazon VPC. For the default security group, you will need to authorize inbound traffic on TCP port 8111. This will allow 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 will need to perform this procedure for that group instead.

1. To determine the default security group identifier, type the following command:

```
aws ec2 describe-security-groups \
--filters Name=vpc-id,Values=vpcID,Name=group-name,Values=default \
--query "SecurityGroups[*].{GroupName:GroupName,GroupId:GroupId}"
```

Replace *vpcID* with your actual VPC ID (from [Step 2: Create a Subnet Group \(p. 560\)](#)).

In the output, take note of the security group identifier. For example: sg-01234567

2. Now do this:

```
aws ec2 authorize-security-group-ingress \
--group-id sgID --protocol tcp --port 8111
```

Replace *sgID* with your actual security group identifier.

AWS Management Console

This section describes how to create a DAX cluster using the AWS Management Console.

Topics

- [Step 1: Create a Subnet Group \(p. 562\)](#)
- [Step 2: Create a DAX Cluster \(p. 563\)](#)
- [Step 3: Configure Security Group Inbound Rules \(p. 563\)](#)

Step 1: Create a Subnet Group

Note

If you have 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 2013-12-04, then you already have a default VPC in each AWS region. For more information, see [Your Default VPC and 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 will be deployed to the subnets within the subnet group.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **DAX**.
3. Choose **Create subnet group**.
4. In the **Create subnet group** window, do the following:
 - a. **Name**—type a short name for the subnet group.
 - b. **Description**—type 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, so that DAX can deploy the cluster nodes into multiple Availability Zones. If an Availability Zone becomes unavailable, DAX will automatically fail over to a surviving Availability Zone, and your DAX cluster will continue to function without interruption.

When the settings are as you want them, click **Create subnet group**.

Step 2: Create a DAX Cluster

Follow this procedure to create a DAX cluster in your default Amazon VPC:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, beneath the **DAX** heading, choose **Clusters**.
3. Choose **Create cluster**.
4. In the **Create cluster** window, do the following:
 - a. **Cluster name**—type a short name for your DAX cluster.
 - b. **Cluster description**—type a description for the cluster.
 - c. **Node type**—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 **2** (one primary and one read replica) and **10** (one primary and nine read replicas).
 - e. **IAM role**—choose **Create new**, and enter the following information:
 - **IAM role name**—type a name for an IAM role. For example: *DAXServiceRole*. The console will create a new IAM role, and your DAX cluster assume this role at runtime.
 - **IAM policy name**—type a name for an IAM policy. For example: *DAXServicePolicy*. The console will create a new IAM policy, and attach the policy to the IAM role.
 - **IAM role policy**—choose **Read/Write**. This will allow the DAX cluster to perform read and write operations in DynamoDB.
 - **Target DynamoDB table**—choose **All tables**.
 - f. **Subnet group**—choose the subnet group that you created in [Step 1: Create a Subnet Group \(p. 562\)](#).
 - g. **Security Groups**—choose **default**.
5. When the settings are as you want them, choose **Launch cluster**.

On the **Clusters** screen, your DAX cluster will be listed with a status of *Creating*.

Note

Creating the cluster will take several minutes. When the cluster is ready, its status changes to *Available*.

In the meantime, you can proceed to [Step 3: Configure Security Group Inbound Rules \(p. 563\)](#) and follow the instructions there.

Step 3: Configure Security Group Inbound Rules

Your DAX cluster uses TCP port 8111 for communication, so you will need to authorize inbound traffic on that port. This will allow 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 will need to perform this procedure for that group instead.

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. In the **Actions** menu, choose **Edit inbound rules**.
4. Choose **Add Rule**, and enter the following information:
 - **Port Range**—type **8111**

Source—type **default**, and then choose the identifier for your default security group.

When the settings are as you want them, choose **Save**.

DAX and DynamoDB Consistency Models

DAX is a write-through caching service, designed to simplify the process of adding a cache to Amazon 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 application behaves as you expect.

In many use cases, the way that your application uses DAX will affect the consistency of data within the DAX cluster, as well as the consistency of data between DAX and DynamoDB.

Topics

- [Consistency Among DAX Cluster Nodes \(p. 564\)](#)
- [DAX Item Cache Behavior \(p. 565\)](#)
- [DAX Query Cache Behavior \(p. 566\)](#)
- [Strongly Consistent Reads \(p. 567\)](#)
- [Negative Caching \(p. 567\)](#)
- [Strategies for Writes \(p. 568\)](#)

Consistency Among DAX Cluster Nodes

To achieve high availability for your application, we recommend that you provision your DAX cluster with at least two nodes (ideally three or more), and place those nodes in multiple availability zones within a region.

When your DAX cluster is running, it will replicate the data among all of the nodes in the cluster (assuming that you have provisioned more than one node). Consider an application that performs a successful `updateItem` using DAX. This causes the item cache in the primary node to be modified with the new value; that value will then be replicated to all of the other nodes in the cluster. This replication is eventually consistent, and usually takes less than one second to complete.

In this scenario, it is 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 will all be consistent when the update has been fully replicated throughout all of the nodes in the cluster. (Note that 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 in such a way that it can tolerate eventually consistent data.

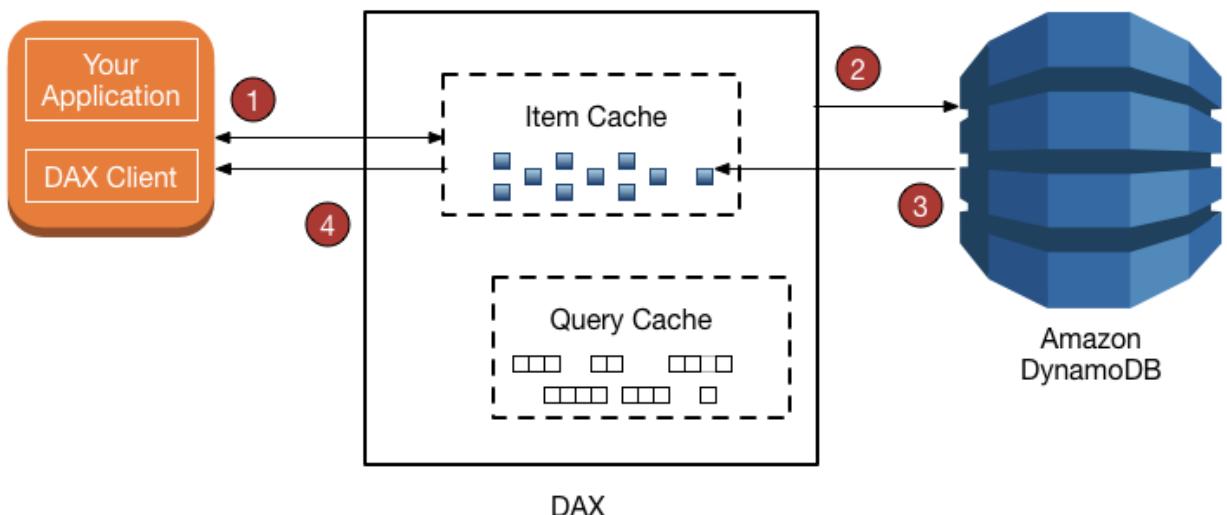
DAX Item Cache Behavior

Every DAX cluster has two distinct caches—an item cache and a query cache. (For more information, see [Concepts \(p. 550\)](#).) This section addresses the consistency implications of reading from and writing to the DAX item cache.

Consistency of Reads

With Amazon DynamoDB, the `GetItem` operation performs an eventually consistent read by default. If you use `GetItem` with the DynamoDB client, and then attempt to read the same item immediately afterward, you might see the data as it appeared prior to the update. This is due to propagation delay across all of 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 proceeds as follows:



1. The DAX client issues a `GetItem` request. DAX attempts 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 of the other nodes in the cluster.

The item will remain in the DAX item cache, subject to the TTL setting and LRU algorithm for the cache (see [Concepts \(p. 550\)](#)). However, during this period, DAX will not re-read the item from DynamoDB. If someone else updates the item using a DynamoDB client, bypassing DAX entirely, then a `GetItem` request using the DAX client will yield different results from the same `GetItem` request using the DynamoDB

client. In this scenario, DAX and DynamoDB will hold inconsistent values for the same key until the TTL for the DAX item has expired.

If an application modifies data in an underlying DynamoDB table, bypassing DAX, the application will need 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`, and `BatchWriteItem`). When you use these operations with the DAX client, the items are modified in both DAX and DynamoDB. DAX will update the items in its item cache, regardless of the TTL value for these items.

For example, suppose you issue a `GetItem` request from the DAX client to read an item from the *ProductCatalog* table. (The partition key is *Id*; 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, let us assume that the TTL is ten minutes. Three 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 ten minutes will 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 allows you to issue writes directly, so that your writes are immediately reflected in the item cache, and thus have the writes be reflected directly in the item cache. You do not need to manage cache invalidation logic, because DAX handles it for you.

DAX supports the following write operations: `PutItem`, `UpdateItem`, `DeleteItem`, and `BatchWriteItem`. When you send one of these requests to DAX, it does the following:

- 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.

If a write to DynamoDB fails for any reason, including throttling, then the item will not be cached in DAX and the exception for the failure will be 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 do not 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 do not 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 cannot "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 where an application is working with a table named `DocumentRevisions`, 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` is 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 will be 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 upon updates to individual items. The `PutItem` operation from step 2 will only be 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 is able to tolerate inconsistent results between the query cache and the item cache.

Strongly Consistent Reads

To perform a strongly consistent read 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 cannot serve strongly consistent reads by itself, because it is 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.

Negative Caching

DAX supports negative cache entries, in both the item cache and the query cache. A *negative cache entry* occurs when DAX cannot 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 will cause DAX to read the corresponding item from the underlying DynamoDB table. If the item does not exist in DynamoDB, then DAX will store an empty item in its item cache, and then return the empty item to the application. Now suppose the application sends another `GetItem` request for the same item. DAX will find the empty item in the item cache, and return it to the application immediately. It does not consult DynamoDB at all.

A negative cache entry will remain in the DAX item cache until its item TTL has expired, LRU is invoked, or until 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 does not contain a cached result, then DAX sends the request to DynamoDB. If there are no matching items in the result set, then DAX stores an empty result set in the query cache, and returns the empty result set to the application. Subsequent `Query` or `Scan` requests will 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 will be 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 \(p. 569\)](#).)

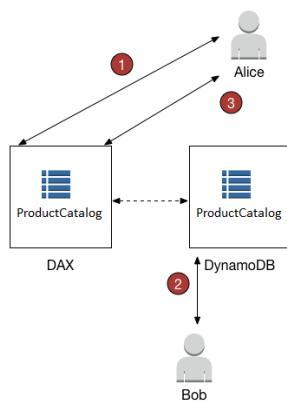
For write-intensive applications (such as those that perform bulk data loading), it might not be desirable to write all of the data through DAX because only a very small percentage of that data is ever read by the application. When you write large amounts of data through DAX, it must invoke its LRU algorithm 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 will not re-read the item from DynamoDB.

Write-Through

The DAX item cache implements a write-through policy (see [How DAX Processes Writes \(p. 566\)](#)). 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 will no longer be 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 will see 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 do not 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.

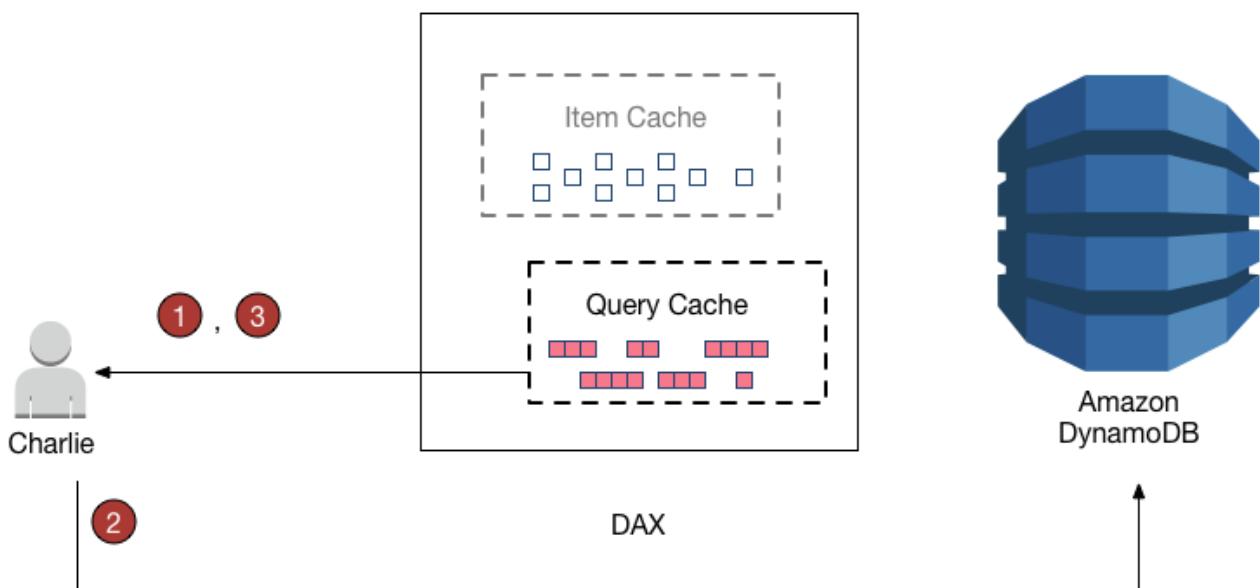
In this scenario, Alice and Bob will see different representations of the same *ProductCatalog* item. This will be the case until DAX evicts the item from the item cache, or until another user updates the same item again using DAX.

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 will reduce write latency; however, the item cache will not remain in sync with the data in DynamoDB.

If you decide to use a write-around strategy, remember that DAX will populate 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).

Consider a user (Charlie) who wants to work with 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, stores the results in the DAX query cache, and then returns the results to Charlie. The result set will remain 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 do not reflect his `PutItem` operation. This will be the case until DAX evicts the result set from the query cache.

Using the DAX Client in an Application

To leverage DAX from an application, you use the DAX client for your programming language. The DAX client is designed for minimal disruption to your existing DynamoDB applications—you only need to make a few simple modifications to the code.

This section demonstrates how to launch an Amazon EC2 instance in your default Amazon VPC, connect to the instance, and run a sample application.

This section also provides information on how to modify your existing application so that it can leverage your DAX cluster.

Note

The DAX clients are available at the following URL:

- <http://dax-sdk.s3-website-us-west-2.amazonaws.com>

At this time, DAX clients are available for Java and Node.js.

Tutorial: Running a Sample Application

Topics

- [Step 1: Launch an Amazon EC2 Instance \(p. 570\)](#)
- [Step 2: Create an IAM User and Policy \(p. 571\)](#)
- [Step 3: Configure Your Amazon EC2 Instance \(p. 572\)](#)
- [Step 4a: \(Java\) Run the Sample Application \(p. 573\)](#)
- [Step 4b: \(Node.js\) Run the Sample Application \(p. 581\)](#)

Note

In order to complete this tutorial, you must have a DAX cluster running in your default VPC. If you have not yet created a DAX cluster, see [Creating a DAX Cluster \(p. 557\)](#) for instructions.

Step 1: Launch an Amazon EC2 Instance

When your DAX cluster is available, you can launch an Amazon EC2 instance in your default Amazon VPC. You will then be able to install and run DAX client software on that instance.

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. Note that 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. 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 will need it for the next step ([Step 3: Configure Your Amazon EC2 Instance \(p. 572\)](#)).

Note

It will take a few minutes for your Amazon EC2 instance to become available. In the meantime, you can proceed to [Step 2: Create an IAM User and Policy \(p. 571\)](#) and follow the instructions there.

Step 2: Create an IAM User and Policy

In this step, you will create an IAM user with a policy that grants access to your DAX cluster and to DynamoDB. You will then be able to run applications that interact with your DAX cluster.

1. Open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, choose **Users**.
3. Choose **Add user**.
4. On the **Details** page, enter the following information:
 - **User name**—type a unique name. For example: MyDAXUser
 - **Access type**—choose **Programmatic access**.

Choose **Next: Permissions**.

5. On the **Permissions** page, choose **Attach existing policies directly**, and then choose **Create policy**.
6. On the **Create Policy** page, select **Create Your Own Policy**.
7. On the **Review Policy** page, enter the following information:
 - **Policy Name**—type a unique name. For example: `MyDAXUserPolicy`.
 - **Description**—type a short description for the policy
 - **Policy Document**—copy and paste the following document:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dax:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        },  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        }  
    ]  
}
```

Choose **Create Policy**.

8. Return to the **Permissions** page. In the list of policies, choose **Refresh**. To narrow the list of policies, choose **Filter | Customer managed**. Choose the IAM policy you created in the previous step (for example: `MyDAXUserPolicy`), and then choose **Next: Review**.
9. On the **Review** page, choose **Create user**.
10. On the **Complete** page, go to the **Secret access key** and choose **Show**. After you do this, copy both the **Access key ID** and **Secret access key**. You will need both of these identifiers for [Step 3: Configure Your Amazon EC2 Instance \(p. 572\)](#).

Step 3: Configure Your Amazon EC2 Instance

When your Amazon EC2 instance is available, you can log into it 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.

1. If you haven't already done so, 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. For 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 \(p. 570\)](#)).

The login ID is `ec2-user`. No password is required.

3. After you have logged in to your EC2 instance, you will need to configure your AWS credentials as shown below. Enter your AWS access key ID and secret key (from [Step 2: Create an IAM User and Policy \(p. 571\)](#)), 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/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]:
```

Step 4a: (Java) Run the Sample Application

Note

This section covers the DAX client for Java. If you prefer using Node.js, skip this section and go to [Step 4b: \(Node.js\) Run the Sample Application \(p. 581\)](#).

To help you test DAX functionality, we have provided a Java program that you can run on your Amazon EC2 instance. The program consists of three source files:

- `TryDax.java`—the entry point for the program.
- `TryDaxHelper.java`—utility methods for DynamoDB and DAX clients, and for creating a test table and data.
- `TryDaxTests.java`—`GetItem`, `Query` and `Scan` activities for the test table.

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
```

4. Set your `CLASSPATH` variable:

```
export SDKVERSION=sdkVersion
```

```
export CLASSPATH=.:./DaxJavaClient-latest.jar:aws-java-sdk-$SDKVERSION/lib/aws-java-sdk-$SDKVERSION.jar:aws-java-sdk-$SDKVERSION/third-party/lib/*
```

Replace *sdkVersion* with the actual version number of the AWS SDK for Java. For example: 1.11.112

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. Compile the code as follows:

```
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. You will now run the program again, but this time the `GetItem`, `Query` and `Scan` operations will be 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 in the console. For example:

```
mycluster.frfx8h.clustercfg.dax.amazonaws.com:8111
```

- **Using the AWS CLI**—type the following command:

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output. For example:

```
{
    "Port": 8111,
    "Address": "mycluster.frfx8h.clustercfg.dax.amazonaws.com"
}
```

Now run the program again—but this time, specify the cluster endpoint as a command line parameter:

```
java TryDax mycluster.frfx8h.clustercfg.dax.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.

For more information about this program, see the following sections:

- [TryDax.java \(p. 576\)](#)
- [TryDaxHelper.java \(p. 577\)](#)
- [TryDaxTests.java \(p. 579\)](#)

TryDax.java

The `TryDax.java` file contains the `main` method. If you run the program with no command line parameters, it creates a DynamoDB client and uses that client for all API operations. If you specify a 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. For example:

- Use the DAX client instead of the DynamoDB client (see [Step 4a: \(Java\) Run the Sample Application \(p. 573\)](#)).
- 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 \(p. 577\)](#))
- 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 do not want to create and delete the table each time you run the program).

Note

To run this program, you must include both the DAX Java client and the AWS SDK for Java in your classpath. See [Step 4a: \(Java\) Run the Sample Application \(p. 573\)](#) for an example of setting your `CLASSPATH` variable.

```
import com.amazonaws.services.dynamodbv2.document.DynamoDB;

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 DynamoDB and 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. For example:

- 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 do not want to create and delete the table each time you run the program).

Note

To run this program, you must include both the DAX Java client and the AWS SDK for Java in your classpath. See [Step 4a: \(Java\) Run the Sample Application \(p. 573\)](#) for an example of setting your `CLASSPATH` variable.

```
import java.util.Arrays;

import com.amazonaws.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 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. For example:

- 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 must include both the DAX Java client and the AWS SDK for Java in your classpath. See [Step 4a: \(Java\) Run the Sample Application \(p. 573\)](#) for an example of setting your `CLASSPATH` variable.

```

import java.util.HashMap;
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));  
    }  
}
```

Step 4b: (Node.js) Run the Sample Application

Note

This section covers the DAX client for Node.js. If you prefer using Java, skip this section and go to [Step 4a: \(Java\) Run the Sample Application \(p. 573\)](#).

To help you test DAX functionality, we have provided some Node.js programs that you can run on your Amazon EC2 instance.

1. Set up Node.js on your Amazon EC2 instance:

- a. Install node version manager (nvm):

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.32.0/install.sh | bash
```

- b. Activate nvm:

```
. ~/.nvm/nvm.sh
```

- c. Use nvm to install Node.js:

```
nvm install 4.8.4
```

- d. 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 v4.8.4
```

2. Install the DAX Node.js client using the node package manager (nvm):

```
npm install https://dax-sdk.s3-us-west-2.amazonaws.com/javascript/amazon-dax-client-latest.tgz
```

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
```

4. Run the following Node.js programs:

```
node 01-create-table.js  
node 02-write-data.js
```

The first program creates a DynamoDB table named *TryDaxTable*. The second program writes data to the table.

5. 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.

6. In the previous step, you ran the programs against the DynamoDB endpoint. You will now run the programs again, but this time the `GetItem`, `Query` and `Scan` operations will be 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 in the console. For example:

```
mycluster.frfx8h.clustercfg.dax.amazonaws.com:8111
```

- **Using the AWS CLI**—type the following command:

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output. For example:

```
{  
    "Port": 8111,  
    "Address": "mycluster.frfx8h.clustercfg.dax.amazonaws.com"  
}
```

Now run the programs again—but this time, specify the cluster endpoint as a command line parameter:

```
node 03-getitem-test.js mycluster.frfx8h.clustercfg.dax.amazonaws.com:8111  
node 04-query-test.js mycluster.frfx8h.clustercfg.dax.amazonaws.com:8111  
node 05-scan-test.js mycluster.frfx8h.clustercfg.dax.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.

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 \(p. 583\)](#)
- [02-write-data.js \(p. 583\)](#)

- [03-getitem-test.js \(p. 584\)](#)
- [04-query-test.js \(p. 585\)](#)
- [05-scan-test.js \(p. 586\)](#)
- [06-delete-table.js \(p. 587\)](#)

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 <= 10; 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 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
            }
        };

        ddbClient.get(params, function(err, data) {
            if (err) {
                console.error("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 dynamodb = new AWS.DynamoDB(); //low-level client
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 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
    }
}

```

```

        ":skval1":sk1,
        ":skval2":sk2
    }
};

for (var i = 0; i < iterations; i++) {
    var startTime = new Date().getTime();

    ddbClient.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 client = new AWS.DynamoDB.DocumentClient()

if (process.argv.length > 2) {
    var dax = new AmazonDaxClient({endpoints: [process.argv[2]], region: region})
    client = new AWS.DynamoDB.DocumentClient({service: dax });
}

var tableName = "TryDaxTable";

var iterations = 5;

var params = {
    TableName: tableName
};

for (var i = 0; i < iterations; i++) {
    var startTime = new Date().getTime();

    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 are done 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));
    }
});

```

Modifying an Existing Application to Use DAX

If you already have a Java application that uses DynamoDB, you will need to modify it so that it can access your DAX cluster. The DAX Java client is very similar to the DynamoDB low-level client included in the AWS SDK for Java, so you do not need to rewrite your entire application.

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, you replace the DynamoDB client with a DAX client:

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDaxClient;
import com.amazonaws.services.dynamodbv2.ClientConfig;
import com.amazonaws.services.dynamodbv2.ClusterDaxClient;
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
        ClientConfig daxConfig = new ClientConfig()

.withEndpoints("mydaxcluster.2cmrwl.clustercfg.dax.usel.cache.amazonaws.com:8111");
        AmazonDaxClient client = new ClusterDaxClient(daxConfig);

        /*
        ** ...
        ** 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 below:

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClient;
import com.amazon.dax.client.dynamodbv2.ClientConfig;
import com.amazon.dax.client.dynamodbv2.ClusterDaxClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
```

```

public class GetMusicItemWithDocumentApi {

    public static void main(String[] args) throws Exception {
        ClientConfig daxConfig = new ClientConfig()
            .withEndpoints("mydaxcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111");
        AmazonDaxClient client = new ClusterDaxClient(daxConfig);

        // Document client wrapper
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");

        try {
            System.out.println("Attempting to read the item...");
            System.out.println(table.getItem(
                "Artist", "No One You Know",
                "SongTitle", "Scared of My Shadow"));
            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 very 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.HashMap;
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();
}
```

Managing DAX Clusters

This section addresses some of the common management tasks for DAX clusters.

Topics

- [IAM Permissions for Managing a DAX Cluster \(p. 591\)](#)
- [Customizing DAX Cluster Settings \(p. 592\)](#)
- [Configuring TTL Settings \(p. 594\)](#)
- [Tagging Support for DAX \(p. 595\)](#)

- [AWS CloudTrail Integration \(p. 596\)](#)
- [Deleting a DAX Cluster \(p. 596\)](#)

IAM Permissions for Managing a DAX Cluster

When you administer a DAX cluster using the AWS Management Console or the 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 (see [Amazon DynamoDB Accelerator](#) in the Amazon DynamoDB API Reference).

Note

For more detailed information about managing IAM permissions, see the following:

- IAM and creating DAX clusters: [Creating a DAX Cluster \(p. 557\)](#).
- IAM and DAX data plane operations: [DAX Access Control \(p. 596\)](#).

For the DAX management APIs, you cannot scope API actions to a specific resource. The `Resource` element must be set to `"*"`. (Note that 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 will fail with a "Not Authorized" exception, because the underlying `DescribeClusters` API call cannot 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 *will* 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, encompassing 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",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:DefineKeySchema",
                "dax:Endpoints"
            ],
            "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": [
                    "*"
                ]
            }
        ]
    }
}
```

Customizing DAX Cluster Settings

When you create a DAX cluster, the following default settings are used:

- Automatic cache eviction enabled with TTL of 5 minutes
- No preference for availability zones
- No preference for maintenance windows
- Notifications disabled

You cannot change these settings on a DAX cluster that is currently running. However, for new clusters, you can customize the settings at creation time. To do this in the AWS Management Console, deselect **Use default settings** to modify the following settings:

- **Network and Security**—allows you to run individual DAX cluster nodes in different Availability Zones (AZs) within the current AWS region. If you choose **No Preference**, the nodes will be distributed among AZs 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 time-to-live (TTL) behavior.
- **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 will be selected at random from an 8-hour block of time per region. (For more information, see [Maintenance Window \(p. 556\)](#).)

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 on setting up and subscribing to an Amazon SNS topic, see [Getting Started with Amazon Simple Notification Service](#) in the Amazon Simple Notification Service Developer Guide.)

Scaling a DAX Cluster

There are two options available for scaling a DAX cluster. The first option is *read scaling*, where you add read replicas to the cluster. The second option is to select different node types.

Read Scaling

With read scaling, you can improve throughput by adding more read replicas to the cluster. A single DAX cluster supports up to 9 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. When the status changes to `available`, the cluster is ready for use again.

Node Types

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. (Note that all of the nodes in a DAX cluster must be of the same type.)

You cannot 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 \(p. 553\)](#).

You can create a new DAX cluster using the AWS Management Console or the AWS CLI. (For the latter, use the `--node-type` parameter to specify the node type.)

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 \(p. 552\)](#) and [Query Cache \(p. 553\)](#)

The default time to live (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 in the AWS Management 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 will be set to 10 minutes and the query cache TTL will be 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 modify-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 modify-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 cannot 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.

Note

For more information, see [Tagging for DynamoDB \(p. 314\)](#).

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, beneath the **DAX** heading, 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**.

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"
```

To 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, you 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 of the changes that have been made at the cluster level, and changes to cluster components such as nodes, subnet groups, and parameter groups. For more information, see [Logging DynamoDB Operations by Using AWS CloudTrail \(p. 658\)](#).

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 from the AWS Management Console or from the AWS CLI. Here is an example:

```
aws dax delete-cluster --cluster-name mydaxcluster
```

DAX Access Control

DAX is designed to work together with DynamoDB, to seamlessly add a caching layer to your applications. However, DAX and DynamoDB are separate AWS services. Even though both services use AWS Identity and Access Management (IAM) to implement their respective security policies, 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.

In this section, we describe the access control mechanisms provided by DAX, and provide 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 [Authentication and Access Control 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 [Authentication and Access Control 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:*"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
            ]  
        }  
    ]  
}
```

This policy allows DAX to perform all DynamoDB API actions on a DynamoDB table named *Books*, which is in the *us-west-2* region and is owned by AWS account ID 123456789012.

IAM Policy to Allow DAX Cluster Access

Once you have created a DAX cluster, you need to grant permissions to an IAM user so that the user can access the DAX cluster.

For example, suppose you want to grant access to *DAXCluster01* to an IAM 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 above. You would then attach the policy to Alice.

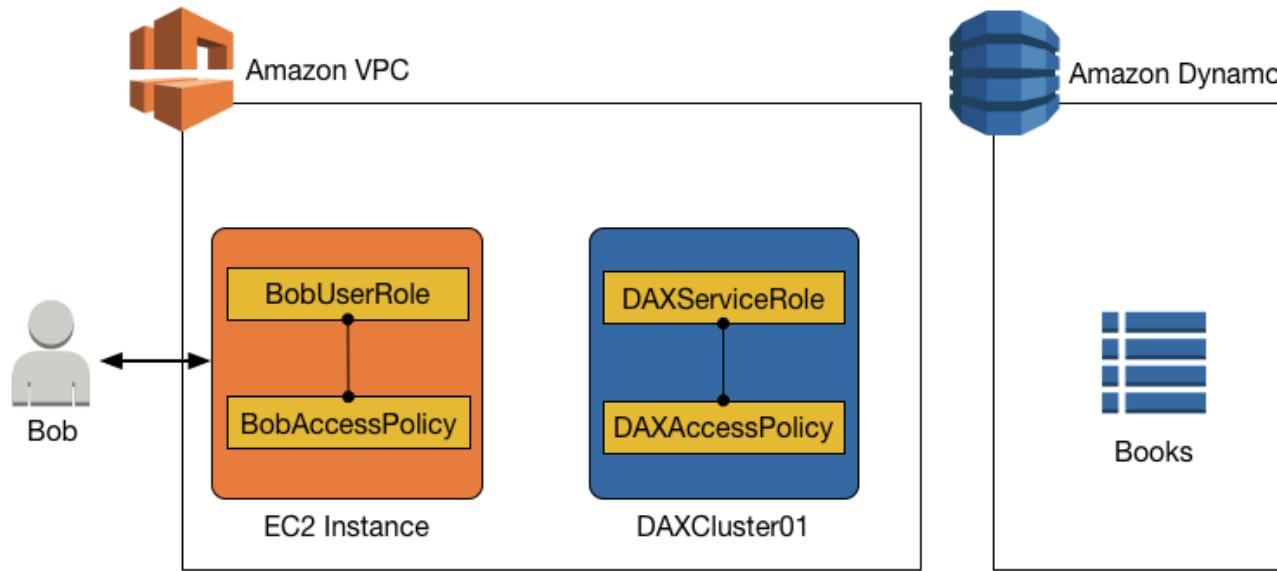
Note

Instead of attaching the policy to an IAM 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, in conjunction 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

To help further your understanding of IAM policies for use with DAX, we present a common scenario. (We will refer to this scenario throughout the rest of this section.) The following diagram shows a high-level overview of the scenario:

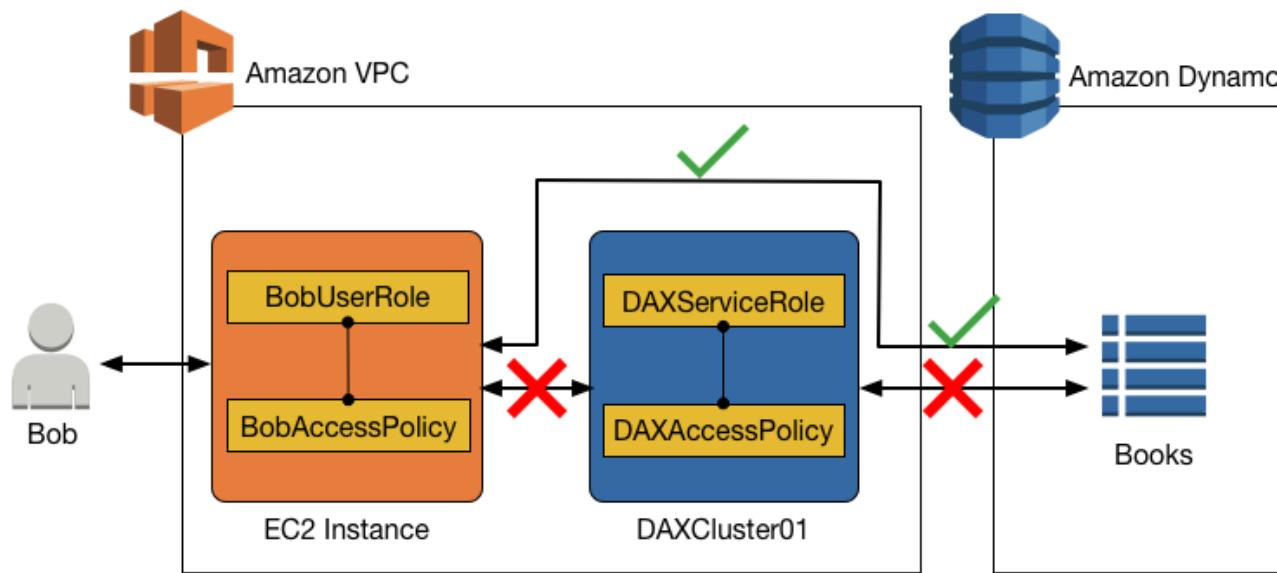


In this scenario, there are the following entities:

- An IAM 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 privileges 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"  
            ],  
            "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.

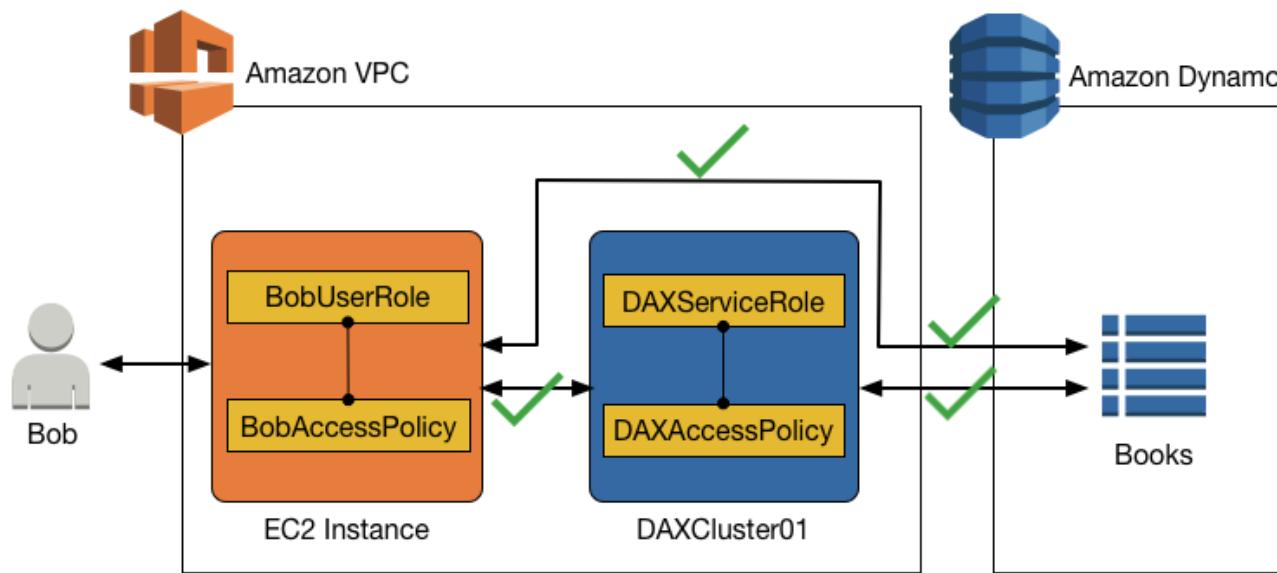
Read-Write Access to DynamoDB (Only)

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"  
            ],  
            "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`

In addition, there are four other DAX-specific actions that do not correspond to any DynamoDB APIs:

- `dax:DefineAttributeList`
- `dax:DefineAttributeListId`
- `dax:DefineKeySchema`
- `dax:Endpoints`

You must specify all four of these actions in any IAM policy that allows access to a DAX cluster. These actions are specific to the low-level DAX data transport protocol. Your application does not need to concern itself with these actions—they are only used in IAM policies.

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*) would confer this access:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:DefineKeySchema",
                "dax:Endpoints"
            ],
            "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 DynamoDB access (`DynamoDBAccessStmt`). These statements would 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* would confer this access:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:DefineKeySchema",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:Endpoints"
            ],
            "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"
            ],
            "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*, 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:DefineKeySchema",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:Endpoints"
            ],
            "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"
            ],
            "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:

```

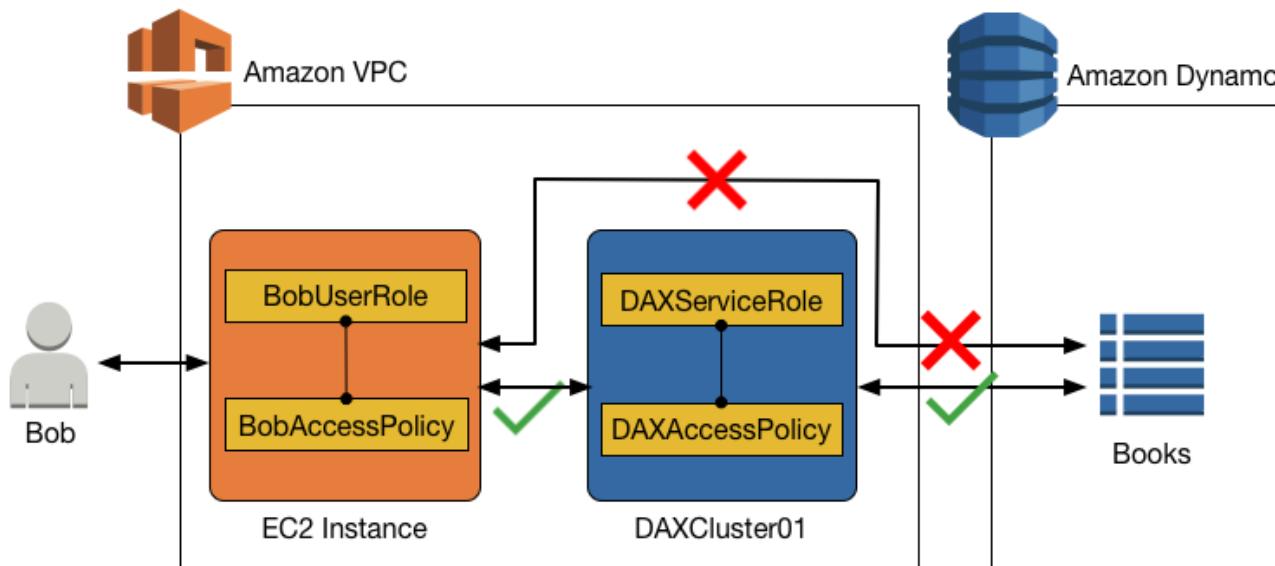
{
    "Version": "2012-10-17",
    "Statement": [
        {

```

```
"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 are configuring 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 will gain 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, then the use of DAX can subvert those policies. In the policy below, 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:DefineKeySchema",  
                "dax:DefineAttributeList",  
                "dax:DefineAttributeListId",  
                "dax:Endpoints"  
            ],  
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"  
        }  
    ]  
}
```

Note that in this access policy, there are no permissions to access DynamoDB directly.

Together with *BobAccessPolicy*, the following *DAXAccessPolicy* would give *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"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

As shown in this example, 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. You must also ensure that giving a user access to a DAX cluster does not subvert previously established access control policies.

DAX API Reference

For more information, see [Amazon DynamoDB Accelerator](#) in the Amazon DynamoDB API Reference.

Authentication and Access Control for Amazon DynamoDB

Access to Amazon DynamoDB requires credentials. Those credentials must have permissions to access AWS resources, such as an Amazon DynamoDB table or an Amazon Elastic Compute Cloud (Amazon EC2) instance. The following sections provide details on how you can use [AWS Identity and Access Management \(IAM\)](#) and DynamoDB to help secure access to your resources.

- [Authentication \(p. 609\)](#)
- [Access Control \(p. 610\)](#)

Authentication

You can access AWS as any of the following types of identities:

- **AWS account root user** – When you first create an AWS account, you begin with a single 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 do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the [best practice of using the root user only to create your first IAM user](#). Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.
- **IAM user** – An [IAM user](#) is an identity within your AWS account that has specific custom permissions (for example, permissions to create a table in DynamoDB). You can use an IAM user name and password to sign in to secure AWS webpages like the [AWS Management Console](#), [AWS Discussion Forums](#), or the [AWS Support Center](#).

In addition to a user name and password, you can also generate [access keys](#) for each user. You can use these keys when you access AWS services programmatically, either through [one of the several SDKs](#) or by using the [AWS Command Line Interface \(CLI\)](#). The SDK and CLI tools use the access keys to cryptographically sign your request. If you don't use AWS tools, you must sign the request yourself. DynamoDB supports *Signature Version 4*, a protocol for authenticating inbound API requests. For more

information about authenticating requests, see [Signature Version 4 Signing Process](#) in the *AWS General Reference*.

- **IAM role** – An [IAM role](#) is an IAM identity that you can create in your account that has specific permissions. It is similar to an *IAM user*, but it is not associated with a specific person. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources. IAM roles with temporary credentials are useful in the following situations:
 - **Federated user access** – Instead of creating an IAM user, you can use existing user identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an [identity provider](#). For more information about federated users, see [Federated Users and Roles](#) in the *IAM User Guide*.
 - **AWS service access** – You can use an IAM role in your account to grant an AWS service permissions to access your account's resources. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see [Creating a Role to Delegate Permissions to an AWS Service](#) in the *IAM User Guide*.
 - **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 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 Roles for Applications on Amazon EC2](#) in the *IAM User Guide*.

Access Control

You can have valid credentials to authenticate your requests, but unless you have permissions you cannot create or access Amazon DynamoDB resources. For example, you must have permissions to create an Amazon DynamoDB table.

The following sections describe how to manage permissions for Amazon DynamoDB. We recommend that you read the overview first.

- [Overview of Managing Access \(p. 610\)](#)
- [Using Identity-Based Policies \(IAM Policies\) \(p. 614\)](#)
- [DynamoDB API Permissions Reference \(p. 621\)](#)
- [Using Conditions \(p. 625\)](#)

Overview of Managing Access Permissions to Your Amazon DynamoDB Resources

Every AWS resource is owned by an AWS account, and permissions to create or access a resource are governed by permissions policies. An account administrator can attach permissions policies to IAM

identities (that is, users, groups, and roles), and some services (such as AWS Lambda) also support attaching permissions policies to resources.

Note

An *account administrator* (or administrator user) is a user with administrator privileges. For more information, see [IAM Best Practices](#) in the *IAM User Guide*.

When granting permissions, you decide who is getting the permissions, the resources they get permissions for, and the specific actions that you want to allow on those resources.

Topics

- [Amazon DynamoDB Resources and Operations \(p. 611\)](#)
- [Understanding Resource Ownership \(p. 611\)](#)
- [Managing Access to Resources \(p. 612\)](#)
- [Specifying Policy Elements: Actions, Effects, and Principals \(p. 613\)](#)
- [Specifying Conditions in a Policy \(p. 613\)](#)

Amazon DynamoDB Resources and Operations

In DynamoDB, the primary resources are *tables*. DynamoDB also supports additional resource types, *indexes* and *streams*. However, you can create indexes and streams only in the context of an existing DynamoDB table. These are referred to as *subresources*.

These resources and subresources have unique Amazon Resource Names (ARNs) associated with them, as shown in the following table.

Resource Type	ARN Format
Table	<code>arn:aws:dynamodb:<i>region</i>:<i>account-id</i>:table/<i>table-name</i></code>
Index	<code>arn:aws:dynamodb:<i>region</i>:<i>account-id</i>:table/<i>table-name</i>/index/<i>index-name</i></code>
Stream	<code>arn:aws:dynamodb:<i>region</i>:<i>account-id</i>:table/<i>table-name</i>/stream/<i>stream-label</i></code>

DynamoDB provides a set of operations to work with DynamoDB resources. For a list of available operations, see [Amazon DynamoDB Actions](#).

Understanding Resource Ownership

The AWS account owns the resources that are created in the account, regardless of who created the resources. Specifically, the resource owner is the AWS account of the [principal entity](#) (that is, the root account, an IAM user, or an IAM role) that authenticates the resource creation request. The following examples illustrate how this works:

- If you use the root account credentials of your AWS account to create a table, your AWS account is the owner of the resource (in DynamoDB, the resource is a table).
- If you create an IAM user in your AWS account and grant permissions to create a table to that user, the user can create a table. However, your AWS account, to which the user belongs, owns the table resource.
- If you create an IAM role in your AWS account with permissions to create a table, anyone who can assume the role can create a table. Your AWS account, to which the user belongs, owns the table resource.

Managing Access to Resources

A *permissions policy* describes who has access to what. The following section explains the available options for creating permissions policies.

Note

This section discusses using IAM in the context of DynamoDB. It doesn't provide detailed information about the IAM service. For complete IAM documentation, see [What Is IAM?](#) in the *IAM User Guide*. For information about IAM policy syntax and descriptions, see [AWS IAM Policy Reference](#) in the *IAM User Guide*.

Policies attached to an IAM identity are referred to as *identity-based* policies (IAM policies) and policies attached to a resource are referred to as *resource-based* policies. DynamoDB supports only identity-based policies (IAM policies).

Topics

- [Identity-Based Policies \(IAM Policies\) \(p. 612\)](#)
- [Resource-Based Policies \(p. 613\)](#)

Identity-Based Policies (IAM Policies)

You can attach policies to IAM identities. For example, you can do the following:

- **Attach a permissions policy to a user or a group in your account** – To grant a user permissions to create an Amazon DynamoDB resource, such as a table, you can attach a permissions policy to a user or group that the user belongs to.
- **Attach a permissions policy to a role (grant cross-account permissions)** – You can attach an identity-based permissions policy to an IAM role to grant cross-account permissions. For example, the administrator in account A can create a role to grant cross-account permissions to another AWS account (for example, account B) or an AWS service as follows:
 1. Account A administrator creates an IAM role and attaches a permissions policy to the role that grants permissions on resources in account A.
 2. Account A administrator attaches a trust policy to the role identifying account B as the principal who can assume the role.
 3. Account B administrator can then delegate permissions to assume the role to any users in account B. Doing this allows users in account B to create or access resources in account A. The principal in the trust policy can also be an AWS service principal if you want to grant an AWS service permissions to assume the role.

For more information about using IAM to delegate permissions, see [Access Management](#) in the *IAM User Guide*.

The following is an example policy that grants permissions for one DynamoDB action (`dynamodb>ListTables`). The wildcard character (*) in the `Resource` value means that you can use this action to obtain the names of all the tables owned by the AWS account in the current AWS region.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ListTables",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb>ListTables"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

```
        "Resource": "*"
    ]
}
```

For more information about using identity-based policies with DynamoDB, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 614\)](#). For more information about users, groups, roles, and permissions, see [Identities \(Users, Groups, and Roles\)](#) in the *IAM User Guide*.

Resource-Based Policies

Other services, such as Amazon S3, also support resource-based permissions policies. For example, you can attach a policy to an S3 bucket to manage access permissions to that bucket. DynamoDB doesn't support resource-based policies.

Specifying Policy Elements: Actions, Effects, and Principles

For each DynamoDB resource, the service defines a set of API operations. To grant permissions for these API operations, DynamoDB defines a set of actions that you can specify in a policy. Some API operations can require permissions for more than one action in order to perform the API operation. For more information about resources and API operations, see [Amazon DynamoDB Resources and Operations \(p. 611\)](#) and [DynamoDB Actions](#).

The following are the most basic policy elements:

- **Resource** – You use an Amazon Resource Name (ARN) to identify the resource that the policy applies to. For more information, see [Amazon DynamoDB Resources and Operations \(p. 611\)](#).
- **Action** – You use action keywords to identify resource operations that you want to allow or deny. For example, you can use `dynamodb:Query` to allow the user permissions to perform the DynamoDB `Query` operation.
- **Effect** – You specify the effect, either allow or deny, when the user requests the specific action. If you don't explicitly grant access to (allow) a resource, access is implicitly denied. You can also explicitly deny access to a resource, which you might do to make sure that a user cannot access it, even if a different policy grants access.
- **Principal** – In identity-based policies (IAM policies), the user that the policy is attached to is the implicit principal. For resource-based policies, you specify the user, account, service, or other entity that you want to receive permissions (applies to resource-based policies only). DynamoDB doesn't support resource-based policies.

To learn more about IAM policy syntax and descriptions, see [AWS IAM Policy Reference](#) in the *IAM User Guide*.

For a list showing all of the Amazon DynamoDB API operations and the resources that they apply to, see [DynamoDB API Permissions: Actions, Resources, and Conditions Reference \(p. 621\)](#).

Specifying Conditions in a Policy

When you grant permissions, you can use the access policy language to specify the conditions when a policy should take effect. For example, you might want a policy to be applied only after a specific date. For more information about specifying conditions in a policy language, see [Condition](#) in the *IAM User Guide*.

To express conditions, you use predefined condition keys. There are AWS-wide condition keys and DynamoDB-specific keys that you can use as appropriate. For a complete list of AWS-wide keys, see

Available Keys for Conditions in the *IAM User Guide*. For a complete list of DynamoDB-specific keys, see [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 625\)](#).

Using Identity-Based Policies (IAM Policies) for Amazon DynamoDB

This topic provides examples of identity-based policies that demonstrate how an account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on Amazon DynamoDB resources.

Important

We recommend that you first review the introductory topics that explain the basic concepts and options available to manage access to your Amazon DynamoDB resources. For more information, see [Overview of Managing Access Permissions to Your Amazon DynamoDB Resources \(p. 610\)](#).

The sections in this topic cover the following:

- [Permissions Required to Use the Amazon DynamoDB Console \(p. 614\)](#)
- [AWS Managed \(Predefined\) Policies for Amazon DynamoDB \(p. 615\)](#)
- [Customer Managed Policy Examples \(p. 615\)](#)

The following shows 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 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` region, which is owned by the AWS account specified by `account-id`. The *Amazon Resource Name (ARN)* in the `Resource` value specifies the table to which the permissions apply.

Permissions Required to Use the Amazon DynamoDB Console

For a user to work with the DynamoDB console, that user must have a minimum set of permissions that allows the user to work with the DynamoDB resources for their AWS account. In addition to these DynamoDB permissions, the console requires permissions from the following services:

- 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` managed policy to the user, as described in [AWS Managed \(Predefined\) Policies for Amazon DynamoDB \(p. 615\)](#).

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the Amazon DynamoDB API.

AWS Managed (Predefined) Policies for Amazon DynamoDB

AWS addresses many 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 what 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 by using the AWS Management Console.
- **AmazonDynamoDBFullAccess** – Grants full access to DynamoDB resources by using the AWS Management Console.
- **AmazonDynamoDBFullAccesswithDataPipeline** – Grants full access to DynamoDB resources, including export and import using AWS Data Pipeline, by using AWS Management Console.

Note

You can review these permissions policies by signing in to the IAM console and searching for specific policies there.

You can also create your own custom IAM policies to allow permissions for DynamoDB actions and resources. You can attach these custom policies to the IAM users or groups that require those permissions.

Customer Managed Policy Examples

In this section, you can find example user policies that grant permissions for various DynamoDB actions. These policies work when you are using AWS SDKs or the AWS CLI. When you are using the console, you need to grant additional permissions specific to the console, which is discussed in [Permissions Required to Use the Amazon DynamoDB Console \(p. 614\)](#).

Note

All examples use the us-west-2 region and contain fictitious account IDs.

Examples

- [Example 1: Allow a User to Perform Any DynamoDB Actions on a Table \(p. 616\)](#)
- [Example 2: Allow Read-only Access on Items in a Table \(p. 616\)](#)
- [Example 3: Allow Put, Update, and Delete Operations on a Specific Table \(p. 616\)](#)
- [Example 4: Allow Access to a Specific Table and All of Its Indexes \(p. 617\)](#)

- [Example 5: Set Up Permissions Policies for Separate Test and Production Environments \(p. 617\)](#)
- [Example 6: Prevent a User from Purchasing Reserved Capacity Offerings \(p. 619\)](#)
- [Example 7: Allow Read Access for a DynamoDB Stream Only \(Not for the Table\) \(p. 620\)](#)
- [Example 8: Allow an AWS Lambda Function to Process DynamoDB Stream Records \(p. 620\)](#)

Example 1: Allow a User to Perform Any DynamoDB Actions on a Table

The following permissions policy grants permissions for all DynamoDB actions on a table. The ARN value specified in the `Resource` identifies a table in a specific region.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Note

If you replace the table name in the resource ARN (`Books`) with a wildcard character (*) , you allow any DynamoDB actions on *all* tables in the account. Carefully consider the security implications if you decide to do this.

Example 2: Allow Read-only Access on Items in a Table

The following permissions policy grants permissions for the `GetItem` and `BatchGetItem` DynamoDB actions only and thereby sets read-only access to a table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:.GetItem",
                "dynamodb:BatchGetItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Example 3: Allow Put, Update, and Delete Operations on a Specific Table

The following permissions policy grants permissions for the `PutItem`, `UpdateItem`, and `DeleteItem` actions on a specific DynamoDB table.

```
{
    "Version": "2012-10-17",
```

```

    "Statement": [
        {
            "Sid": "PutUpdateDeleteOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}

```

Example 4: Allow Access to a Specific Table and All of Its Indexes

The following permissions policy grants permissions for all of the DynamoDB actions on a table (`Book`) and all of the table's indexes. For more information about how indexes work, see [Improving Data Access with Secondary Indexes \(p. 446\)](#).

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AccessAllIndexesOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
            ]
        }
    ]
}

```

Example 5: Set Up Permissions Policies for Separate Test and Production Environments

Suppose you have separate test and production environments where each environment maintains its own version of a table named `ProductCatalog`. If you create these `ProductCatalog` tables from the same AWS account, testing work might affect the production environment because of the way that permissions are set up (for example, the limits on concurrent create and delete actions are set at the AWS account level). As a result, each action in the test environment reduces the number of actions that are available in your production environment. There is also a risk that the code in your test environment might accidentally access tables in the production environment. To prevent these issues, consider creating separate AWS accounts for your production and test environments.

Suppose further that you have two developers, Bob and Alice, who are testing the `ProductCatalog` table. Instead of creating a separate AWS account for every developer, your developers can share the same test 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 `Bob_ProductCatalog`. In this case, you can create IAM users Alice and Bob in the AWS account that you created for the test environment. You can then grant permissions to these users to perform DynamoDB actions on the tables that they own.

To grant these 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 all DynamoDB actions on the `Alice_ProductCatalog` table:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnAliceTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
Alice_ProductCatalog"
        }
    ]
}
```

Then, you can create a similar policy with a different resource (`Bob_ProductCatalog` table) for user Bob.

- 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 user Bob 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": "AllAPIActionsOnUserSpecificTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
#{aws:username}_ProductCatalog"
        },
        {
            "Sid": "AdditionalPrivileges",
            "Effect": "Allow",
            "Action": [
                "dynamodb>ListTables",
                "dynamodb>DescribeTable",
                "cloudwatch: *",
                "sns: *"
            ],
            "Resource": "*"
        }
    ]
}
```

Note

When using IAM policy variables, you must explicitly specify the 2012-10-17 version of the access policy language in the policy. The default version of the access policy language (2008-10-17) does not support policy variables.

Note that, instead of identifying a specific table as a resource, you can use a wildcard character (*) to grant permissions on all tables where the name is prefixed with the name of the IAM user that is making the request, as shown following:

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_*"
```

Example 6: Prevent a User from Purchasing Reserved Capacity Offerings

DynamoDB customers can purchase reserved capacity, as described at [Amazon DynamoDB Pricing](#). With 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 have the same levels of access.

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 operations to display reserved capacity information and to make purchases. You cannot call these operations from an application program, because they are only accessible 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 offerings and current purchases 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:/*"  
        }  
    ]  
}
```

Example 7: Allow Read Access for a DynamoDB Stream Only (Not for the Table)

When you enable DynamoDB Streams on a table, it captures information about every modification to data items in the table. For more information, see [Capturing Table Activity with DynamoDB Streams \(p. 518\)](#).

In some cases, you might want to prevent an application from reading data from a DynamoDB table, while still allowing access to that table's stream. For example, you can configure AWS Lambda to poll the 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 creates a policy that grants users permissions to access the streams on a table named `GameScores`. The final wildcard character (*) in the ARN matches any stream ID 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 permits access to the streams on the `GameScores` table, but not to the table itself.

Example 8: Allow an AWS Lambda Function to Process DynamoDB Stream Records

If you want certain actions to be performed based on new events in a DynamoDB stream, you can write an AWS Lambda function that is triggered by these new events. For more information about using Lambda with stream events, see [DynamoDB Streams and AWS Lambda Triggers \(p. 541\)](#). A Lambda function such as this needs permissions to read data from the DynamoDB stream.

To grant permissions to Lambda, you use the permissions policy that is associated with the Lambda function's IAM role (execution role), which you specify when you create the Lambda function.

For example, you can associate the following permissions policy with the execution role to grant Lambda permissions to perform the DynamoDB Streams actions listed.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowLambdaFunctionInvocation",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "AllAPIAccessForDynamoDBStreams",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb>ListStreams"
            ],
            "Resource": "*"
        }
    ]
}
```

For more information, see [AWS Lambda Permission Model](#) in the *AWS Lambda Developer Guide*.

DynamoDB API Permissions: Actions, Resources, and Conditions Reference

When you are setting up [Access Control \(p. 610\)](#) and writing a permissions policy that you can attach to an IAM identity (identity-based policies), you can use the following list as a reference. The list includes 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 [Available Keys](#) 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 \(p. 625\)](#).

Note

To specify an action, use the `dynamodb:` prefix followed by the API operation name (for example, `dynamodb>CreateTable`).

DynamoDB API Permissions: Actions, Resources, and Condition Keys Reference

[BatchGetItem](#)

Action(s): `dynamodb:BatchGetItem`

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[BatchWriteItem](#)

Action(s): dynamodb:BatchWriteItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[CreateTable](#)

Action(s): dynamodb:CreateTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[DeleteItem](#)

Action(s): dynamodb:DeleteItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[DeleteTable](#)

Action(s): dynamodb>DeleteTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[DescribeReservedCapacity](#)

Action(s): dynamodb:DeleteTable

Resource:

`arn:aws:dynamodb:region:account-id:*`

[DescribeReservedCapacityOfferings](#)

Action(s): dynamodb:DescribeReservedCapacityOfferings

Resource:

`arn:aws:dynamodb:region:account-id>*`

[DescribeStream](#)

Action(s): dynamodb:DescribeStream

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label`

or

`arn:aws:dynamodb:region:account-id:table/table-name/stream/*`

[DescribeTable](#)

Action(s): dynamodb:DescribeTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[GetItem](#)

Action(s): dynamodb:GetItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[GetRecords](#)

Action(s): dynamodb:GetRecords

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label`

or

`arn:aws:dynamodb:region:account-id:table/table-name/stream/*`

[GetShardIterator](#)

Action(s): dynamodb:GetShardIterator

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label`

or

arn:aws:dynamodb:**region:account-id:table/table-name/stream/***

ListStreams

Action(s): dynamodb>ListStreams

Resource:

arn:aws:dynamodb:**region:account-id:table/table-name/stream/***

or

arn:aws:dynamodb:**region:account-id:table/*/stream/***

ListTables

Action(s): dynamodb>ListTables

Resource:

*

PurchaseReservedCapacityOfferings

Action(s): dynamodb>PurchaseReservedCapacityOfferings

Resource:

arn:aws:dynamodb:**region:account-id:***

PutItem

Action(s): dynamodb:PutItem

Resource:

arn:aws:dynamodb:**region:account-id:table/table-name**

or

arn:aws:dynamodb:**region:account-id:table/***

Query

Action(s): dynamodb:Query

Resource:

To query a table:arn:aws:dynamodb:**region:account-id:table/table-name**

or:

arn:aws:dynamodb:**region:account-id:table/table-name**

To query an index:

arn:aws:dynamodb:**region:account-id:table/table-name/index/index-name**

or:

arn:aws:dynamodb:**region:account-id:table/table-name/index/***

Scan

Action(s): dynamodb:Scan

Resource:

To scan a table:

`arn:aws:dynamodb:region:account-id:table/table-name`

or:

`arn:aws:dynamodb:region:account-id:table/table-name`

To scan an index:

`arn:aws:dynamodb:region:account-id:table/table-name/index/index-name`

or:

`arn:aws:dynamodb:region:account-id:table/table-name/index/*`

UpdateItem

Action(s): dynamodb:UpdateItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

UpdateTable

Action(s): dynamodb:UpdateTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

Related Topics

- [Access Control \(p. 610\)](#)
- [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 625\)](#)

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 [Access Control \(p. 610\)](#)). 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 to 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.

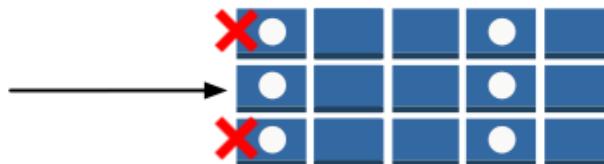
Note

Some AWS services also support tag-based conditions; however, DynamoDB does not.

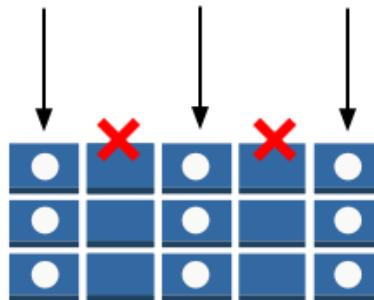
Permissions Use Case

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 are 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

IAM 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 \(p. 635\)](#).

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": {
                "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 \(p. 635\)](#).
- `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+0OFF.

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.

Note

Condition keys are case-sensitive.

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. Note that the key name <code>LeadingKeys</code> is plural, even if the key is used with single-item actions. In addition, note that 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 same way and have the same meaning as the parameters for certain DynamoDB API actions, as shown following: <ul style="list-style-type: none"> • <code>AttributesToGet</code> Used by: <code>BatchGetItem</code> , <code>.GetItem</code> , <code>Query</code> , <code>Scan</code>

DynamoDB Condition Key	Description
	<ul style="list-style-type: none"> • <code>AttributeUpdates</code> Used by: <code>UpdateItem</code> • <code>Expected</code> Used by: <code>DeleteItem</code>, <code>PutItem</code>, <code>UpdateItem</code> • <code>Item</code> Used by: <code>PutItem</code> • <code>ScanFilter</code> 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 \(p. 635\)](#).

Note

Each of the examples in the following section sets the `Effect` clause to `Allow` and specify 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 `User_ID` 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 a whitelist approach, which allows access to a named set of attributes. You can write an equivalent policy that denies access to other attributes instead (that is, a blacklist).

approach). We don't recommend this *blacklist* approach because users can determine the names of these blacklisted attributes by repeatedly issuing requests for all possible attribute names, eventually finding an attribute that they aren't allowed to access. To avoid this, follow the *principle of least privilege*, as explained in Wikipedia at http://en.wikipedia.org/wiki/Principle_of_least_privilege, and use a *whitelist* approach to enumerate all of the allowed values, rather than specifying the denied attributes.

This policy doesn't permit `PutItem`, `DeleteItem`, or `BatchWriteItem`, because 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, we can remove `UpdateItem` from the list of allowed actions. Because none of the remaining actions accept `ReturnValues`, we can remove `ReturnValues` from the condition. We can also change `StringEqualsIfExists` to `StringEquals` because the `Select` parameter always has a value (`ALL_ATTRIBUTES`, unless otherwise specified).
- To allow only write actions, we can remove everything except `UpdateItem` from the list of allowed actions. Because `UpdateItem` does not use the `Select` parameter, we can remove `Select` from the condition. We 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": [
                        "Score"
                    ]
                }
            }
        }
    ]
}
```

```
        "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": {
            "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 whitelisted attributes can be requested. For write requests, the application cannot set `ReturnValues` to `ALL_OLD` or `ALL_NEW` and it cannot perform conditional write operations based on any other attributes.

Related Topics

- [Access Control \(p. 610\)](#)
- [DynamoDB API Permissions: Actions, Resources, and Conditions Reference \(p. 621\)](#)

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 IAM 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 [Web Identity Federation Playground](#) is an interactive website that lets you walk through the process of authenticating via Login with Amazon, Facebook, or Google, getting temporary security credentials, and then using those credentials to make a request to AWS.
- The entry [Web Identity Federation using the AWS SDK for .NET](#) on the AWS .NET Development blog walks through how to use web identity federation with Facebook and 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 SDK for iOS](#) and the [AWS SDK for Android](#) contain sample apps. These apps 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 web identity federation can be used with DynamoDB, let's revisit the *GameScores* table that was introduced in [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 625\)](#). Here is the primary key for *GameScores*:

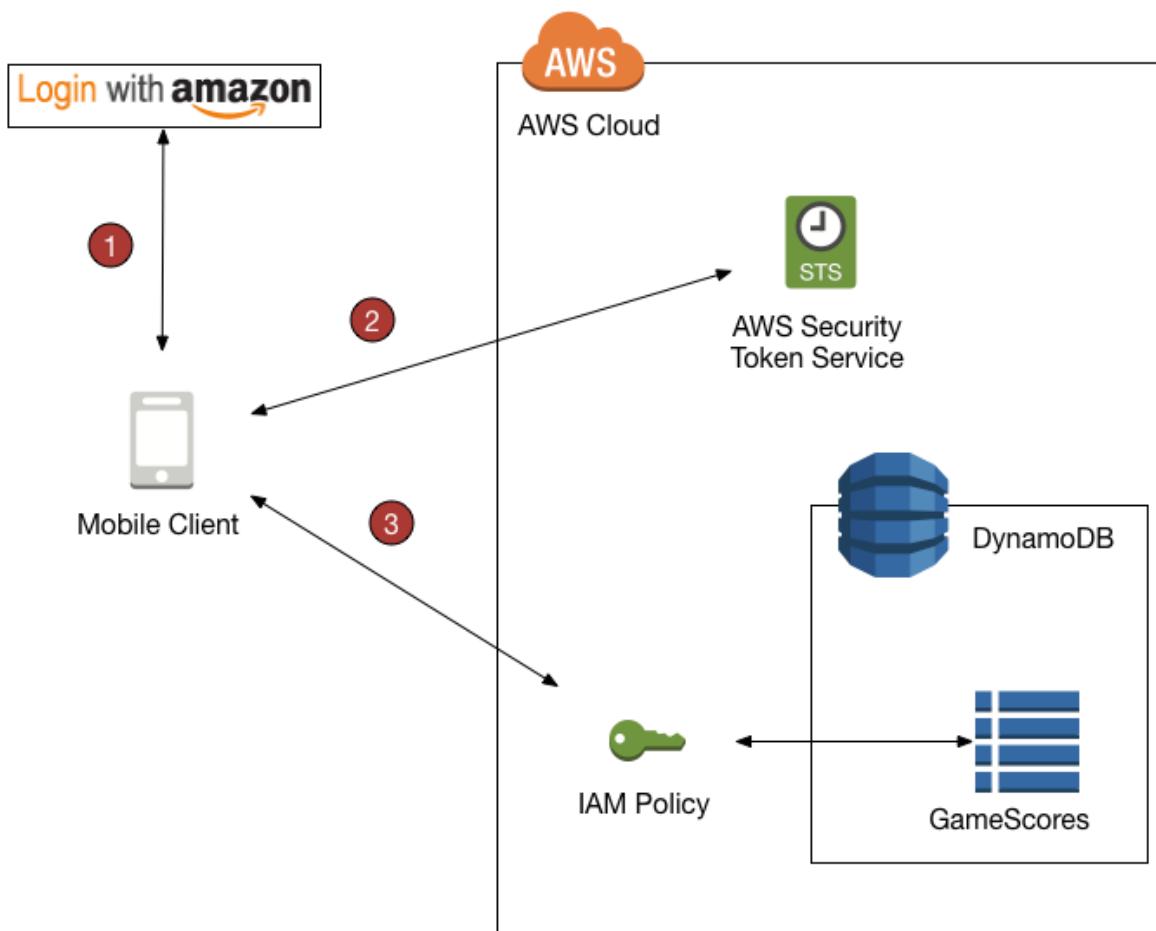
Table Name	Primary Key Type	Partition Key Name and Type	Sort Key Name and Type
GameScores (<u>UserId</u> , <u>GameTitle</u> , ...)	Composite	Attribute Name: UserId Type: String	Attribute Name: GameTitle Type: String

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 leverage 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, we will give this role a name of *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, he signs in to his 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 \(p. 625\)](#):

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAccessToOnlyItemsMatchingUserID",
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/GameScores/*"
      ],
      "Condition": {
        "StringEquals": {
          "dynamodb:partitionKey": "#ID"
        }
      }
    }
  ]
}
```

```

        "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:

- 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
- 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 will need to obtain an app ID from each provider.
- Create one or more IAM roles.** You will 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 will 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 the following pages:

- [How to Authenticate Users \(AWS Mobile SDK for iOS\)](#)
- [How to Authenticate Users \(AWS Mobile SDK for Android\)](#)

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 will then generate 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. Choose the **Access control** tab.

5. Choose the identity provider, actions, and attributes for the policy.

When the settings are as you want them, click **Create policy**. The generated policy appears.

6. Click **Attach policy instructions**, 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 will honor 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 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 (3600 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>AROAJU4SA2VW5S2RF2YMG: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 will require 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 will no longer be valid.
- The `AssumedRoleId` field contains the name of a session-specific IAM role that has been assumed by the app. The app will honor 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.AGJZDKHJKAUUSW6C44CHPEXAMPLE
<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 \(p. 629\)](#).

For more information about how AWS Security Token Service generates temporary access credentials, see [Requesting Temporary Security Credentials](#) in *IAM User Guide*.

Monitoring DynamoDB

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon 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 Amazon 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 Amazon DynamoDB performance in your environment, by measuring performance at various times and under different load conditions. As you monitor Amazon 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 limits of a table.
- System errors, so you can determine if any requests resulted in an error.

Topics

- [Monitoring Tools \(p. 643\)](#)
- [Monitoring with Amazon CloudWatch \(p. 644\)](#)

- [Logging DynamoDB Operations by Using AWS CloudTrail \(p. 658\)](#)

Monitoring Tools

AWS provides tools that you can use to monitor Amazon 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 Amazon 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 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 \(p. 644\)](#).
- **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 [Using 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 Amazon DynamoDB involves manually monitoring those items that the CloudWatch alarms don't cover. The Amazon 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.

- Amazon 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 Amazon DynamoDB into readable, near real-time metrics. These statistics are recorded for a period of two weeks, so that you can access historical information for a better perspective on how your web application or service is performing. By default, Amazon DynamoDB metric data is sent to CloudWatch automatically. For more information, see [What Is Amazon CloudWatch?](#) in the *Amazon CloudWatch User Guide*.

Topics

- [Amazon DynamoDB Metrics and Dimensions \(p. 644\)](#)
- [How Do I Use Amazon DynamoDB Metrics? \(p. 655\)](#)
- [Creating CloudWatch Alarms to Monitor Amazon DynamoDB \(p. 656\)](#)

Amazon 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 Amazon 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**.
3. Select the **DynamoDB** namespace.

To view metrics (CLI)

- At a command prompt, use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/DynamoDB"
```

CloudWatch displays the following metrics for Amazon DynamoDB:

Amazon DynamoDB Dimensions and Metrics

The metrics and dimensions that Amazon DynamoDB sends to Amazon CloudWatch are listed here.

DynamoDB Metrics

The following metrics are available from Amazon DynamoDB. Note that DynamoDB only sends metrics to CloudWatch when they have a non-zero value. For example, the `UserErrors` metric is incremented whenever a request generates an HTTP 400 status code. If no HTTP 400 errors were encountered during a time period, CloudWatch will not provide metrics for `UserErrors` during that period.

Note

Amazon CloudWatch aggregates the following DynamoDB metrics at one-minute intervals:

- `ConditionalCheckFailedRequests`
- `ConsumedReadCapacityUnits`
- `ConsumedWriteCapacityUnits`

- `ReadThrottleEvents`
- `ReturnedBytes`
- `ReturnedItemCount`
- `ReturnedRecordsCount`
- `SuccessfulRequestLatency`
- `SystemErrors`
- `TimeToLiveDeletedItemCount`
- `ThrottledRequests`
- `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 table, each metric has a list of Valid Statistics that is applicable to that metric.

Metric	Description
<code>ConditionalCheckFailedRequests</code>	<p>The number of failed attempts to perform conditional writes. The <code>PutItem</code>, <code>UpdateItem</code>, and <code>DeleteItem</code> operations let you provide a logical condition that must evaluate to true before the operation can proceed. If this condition evaluates to false, <code>ConditionalCheckFailedRequests</code> is incremented by one.</p> <p>Note A failed conditional write will result in an HTTP 400 error (Bad Request). These events are reflected in the <code>conditionalCheckFailedRequests</code> metric, but not in the <code>UserErrors</code> metric.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> • <code>Maximum</code> • <code>Average</code> • <code>SampleCount</code> • <code>Sum</code>
<code>ConsumedReadCapacityUnits</code>	<p>The number of read capacity units consumed over the specified time period, so you can track how much of your provisioned 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 Provisioned Throughput in Amazon DynamoDB.</p>

Metric	Description
	<p>Note Use the <code>Sum</code> statistic to calculate the consumed throughput. For example, get the <code>Sum</code> value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average <code>ConsumedReadCapacityUnits</code> per second (recognizing that this average will not highlight any large but brief spikes in read activity that occurred during that minute). You can compare the calculated value to the provisioned throughput value you provide DynamoDB.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> – Minimum number of read capacity units consumed by any individual request to the table or index. • <code>Maximum</code> – Maximum number of read capacity units consumed by any individual request to the table or index. • <code>Average</code> – Average per-request read capacity consumed. • <code>Sum</code> – Total read capacity units consumed. This is the most useful statistic for the <code>ConsumedReadCapacityUnits</code> metric. • <code>SampleCount</code> – Number of requests to DynamoDB that consumed read capacity.

Metric	Description
ConsumedWriteCapacityUnits	<p>The number of write capacity units consumed over the specified time period, so you can track how much of your provisioned 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 Provisioned Throughput in Amazon DynamoDB.</p> <p>Note Use the <code>Sum</code> statistic to calculate the consumed throughput. For example, get the <code>Sum</code> value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average <code>ConsumedWriteCapacityUnits</code> per second (recognizing that this average will 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 you provide DynamoDB.</p> <p>Units: <code>Count</code></p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> – Minimum number of write capacity units consumed by any individual request to the table or index. • <code>Maximum</code> – Maximum number of write capacity units consumed by any individual request to the table or index. • <code>Average</code> – Average per-request write capacity consumed. • <code>Sum</code> – Total write capacity units consumed. This is the most useful statistic for the <code>ConsumedWriteCapacityUnits</code> metric. • <code>SampleCount</code> – Number of requests to DynamoDB that consumed write capacity.

Metric	Description
OnlineIndexConsumedWriteCapacity	<p>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.</p> <p>You can adjust the write capacity of the index using the <code>UpdateTable</code> operation, even while the index is still being built.</p> <p>Note that the <code>ConsumedWriteCapacityUnits</code> metric for the index does not include the write throughput consumed during index creation.</p> <p>Units: <code>Count</code></p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> • <code>Maximum</code> • <code>Average</code> • <code>SampleCount</code> • <code>Sum</code>
OnlineIndexPercentageProgress	<p>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.</p> <p>Units: <code>Count</code></p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> • <code>Maximum</code> • <code>Average</code> • <code>SampleCount</code> • <code>Sum</code>

Metric	Description
OnlineIndexThrottleEvents	<p>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.</p> <p>You can adjust the write capacity of the index using the <code>UpdateTable</code> operation, even while the index is still being built.</p> <p>Note that the <code>WriteThrottleEvents</code> metric for the index does not include any throttle events that occur during index creation.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum
ProvisionedReadCapacityUnits	<p>The number of provisioned read capacity units for a table or a global secondary index.</p> <p>The <code>TableName</code> dimension returns the <code>ProvisionedReadCapacityUnits</code> for the table, but not for any global secondary indexes. To view <code>ProvisionedReadCapacityUnits</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum – Lowest setting for provisioned read capacity. If you use <code>UpdateTable</code> to increase read capacity, this metric shows the lowest value of provisioned <code>ReadCapacityUnits</code> during this time period. • Maximum – Highest setting for provisioned read capacity. If you use <code>UpdateTable</code> to decrease read capacity, this metric shows the highest value of provisioned <code>ReadCapacityUnits</code> during this time period. • Average – Average provisioned read capacity. The <code>ProvisionedReadCapacityUnits</code> 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.

Metric	Description
ProvisionedWriteCapacityUnits	<p>The number of provisioned write capacity units for a table or a global secondary index</p> <p>The <code>TableName</code> dimension returns the <code>ProvisionedWriteCapacityUnits</code> for the table, but not for any global secondary indexes. To view <code>ProvisionedWriteCapacityUnits</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> – Lowest setting for provisioned write capacity. If you use <code>UpdateTable</code> to increase write capacity, this metric shows the lowest value of provisioned <code>WriteCapacityUnits</code> during this time period. • <code>Maximum</code> – Highest setting for provisioned write capacity. If you use <code>UpdateTable</code> to decrease write capacity, this metric shows the highest value of provisioned <code>WriteCapacityUnits</code> during this time period. • <code>Average</code> – Average provisioned write capacity. The <code>ProvisionedWriteCapacityUnits</code> 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	<p>Requests to DynamoDB that exceed the provisioned read capacity units for a table or a global secondary index.</p> <p>A single request can result in multiple events. For example, a <code>BatchGetItem</code> that reads 10 items is processed as ten <code>GetItem</code> events. For each event, <code>ReadThrottleEvents</code> is incremented by one if that event is throttled. The <code>ThrottledRequests</code> metric for the entire <code>BatchGetItem</code> is not incremented unless <i>all ten</i> of the <code>GetItem</code> events are throttled.</p> <p>The <code>TableName</code> dimension returns the <code>ReadThrottleEvents</code> for the table, but not for any global secondary indexes. To view <code>ReadThrottleEvents</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>SampleCount</code> • <code>Sum</code>

Metric	Description
ReturnedBytes	<p>The number of bytes returned by <code>GetRecords</code> operations (Amazon DynamoDB Streams) during the specified time period.</p> <p>Units: Bytes</p> <p>Dimensions: Operation, StreamLabel, TableName</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum
ReturnedItemCount	<p>The number of items returned by <code>Query</code> or <code>Scan</code> operations during the specified time period.</p> <p>Note that the number of items <i>returned</i> is not necessarily the same as the number of items that were evaluated. For example, suppose you requested a <code>Scan</code> on a table that had 100 items, but specified a <code>FilterExpression</code> that narrowed the results so that only 15 items were returned. In this case, the response from <code>Scan</code> would contain a <code>ScanCount</code> of 100 and a <code>Count</code> of 15 returned items.</p> <p>Units: Count</p> <p>Dimensions: TableName, Operation</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum
ReturnedRecordsCount	<p>The number of stream records returned by <code>GetRecords</code> operations (Amazon DynamoDB Streams) during the specified time period.</p> <p>Units: Count</p> <p>Dimensions: Operation, StreamLabel, TableName</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
SuccessfulRequestLatency	<p>Successful requests to DynamoDB or Amazon DynamoDB Streams during the specified time period. <code>SuccessfulRequestLatency</code> can provide two different kinds of information:</p> <ul style="list-style-type: none"> The elapsed time for successful requests (<code>Minimum</code>, <code>Maximum</code>, <code>Sum</code>, or <code>Average</code>). The number of successful requests (<code>sampleCount</code>). <p><code>SuccessfulRequestLatency</code> reflects activity only within DynamoDB or Amazon DynamoDB Streams, and does not take into account network latency or client-side activity.</p> <p>Units: <code>Milliseconds</code></p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> <code>Minimum</code> <code>Maximum</code> <code>Average</code> <code>SampleCount</code>
SystemErrors	<p>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.</p> <p>Units: <code>Count</code></p> <p>Dimensions: All dimensions</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> <code>Sum</code> <code>SampleCount</code>
TimeToLiveDeletedItemCount	<p>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.</p> <p>Units: <code>Count</code></p> <p>Dimensions: <code>TableName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> <code>Sum</code>

Metric	Description
ThrottledRequests	<p>Requests to DynamoDB that exceed the provisioned throughput limits on a resource (such as a table or an index).</p> <p><code>ThrottledRequests</code> 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 <code>ThrottledRequests</code> is incremented by one.</p> <p>Note In a batch request (<code>BatchGetItem</code> OR <code>BatchWriteItem</code>), <code>ThrottledRequests</code> is only incremented if <i>every</i> request in the batch is throttled. If any individual request within the batch is throttled, one of the following metrics is incremented:</p> <ul style="list-style-type: none"> • <code>ReadThrottleEvents</code> – For a throttled <code>GetItem</code> event within <code>BatchGetItem</code>. • <code>WriteThrottleEvents</code> – For a throttled <code>PutItem</code> or <code>DeleteItem</code> event within <code>BatchWriteItem</code>. <p>To gain insight into which event is throttling a request, compare <code>ThrottledRequests</code> with the <code>ReadThrottleEvents</code> and <code>WriteThrottleEvents</code> for the table and its indexes.</p> <p>Note A throttled request will result in an HTTP 400 status code. All such events are reflected in the <code>ThrottledRequests</code> metric, but not in the <code>UserErrors</code> metric.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount

Metric	Description
UserErrors	<p>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, attempting to update a nonexistent table, or an incorrect request signature.</p> <p>All such events are reflected in the <code>UserErrors</code> metric, except for the following:</p> <ul style="list-style-type: none"> • <i>ProvisionedThroughputExceededException</i> – See the <code>ThrottledRequests</code> metric in this section. • <i>ConditionalCheckFailedException</i> – See the <code>ConditionalCheckFailedRequests</code> metric in this section. <p><code>UserErrors</code> represents the aggregate of HTTP 400 errors for DynamoDB or Amazon DynamoDB Streams requests for the current region and the current AWS account.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount
WriteThrottleEvents	<p>Requests to DynamoDB that exceed the provisioned write capacity units for a table or a global secondary index.</p> <p>A single request can result in multiple events. For example, a <code>PutItem</code> 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 <code>WriteThrottleEvents</code> metric is incremented by one if that event is throttled. For single <code>PutItem</code> requests, if any of the events are throttled, <code>ThrottledRequests</code> is also incremented by one. For <code>BatchWriteItem</code>, the <code>ThrottledRequests</code> metric for the entire <code>BatchWriteItem</code> is not incremented unless all of the individual <code>PutItem</code> OR <code>DeleteItem</code> events are throttled.</p> <p>The <code>TableName</code> dimension returns the <code>writeThrottleEvents</code> for the table, but not for any global secondary indexes. To view <code>writeThrottleEvents</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndexName</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount

Dimensions for DynamoDB Metrics

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.

Dimension	Description
GlobalSecondaryIndexName	This dimension limits the data to a global secondary index on a table. If you specify <code>GlobalSecondaryIndexName</code> , you must also specify <code>TableName</code> .
Operation	<p>This dimension limits the data to one of the following DynamoDB operations:</p> <ul style="list-style-type: none"> • <code>PutItem</code> • <code>DeleteItem</code> • <code>UpdateItem</code> • <code>GetItem</code> • <code>BatchGetItem</code> • <code>Scan</code> • <code>Query</code> • <code>BatchWriteItem</code> <p>In addition, you can limit the data to the following Amazon DynamoDB Streams operation:</p> <ul style="list-style-type: none"> • <code>GetRecords</code>
StreamLabel	This dimension limits the data to a specific stream label. It is used with metrics originating from Amazon DynamoDB Streams <code>GetRecords</code> 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 Amazon DynamoDB Metrics?

The metrics reported by Amazon 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 Kinesis Firehose .

How can I?	Relevant Metrics
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 limits of a table?	<code>ThrottledRequests</code> is incremented by one if any event within a request exceeds a provisioned throughput limit. 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?	<p>You can monitor <code>SystemErrors</code> to determine if any requests resulted in a HTTP 500 (server error) code. Typically, this metric should be equal to zero. If it isn't, then you might want to investigate.</p> <p>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.</p>

Creating CloudWatch Alarms to Monitor Amazon 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.

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.

```
Prompt>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.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name ReadCapacityUnitsLimitAlarm --state-reason "initializing" --state-value OK
```

```
Prompt>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 limits 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.

```
Prompt>aws cloudwatch put-metric-alarm \
    --alarm-name RequestsExceedingThroughputAlarm \
    --alarm-description "Alarm when my requests are exceeding provisioned throughput limits of a table" \
    --namespace AWS/DynamoDB \
    --metric-name ThrottledRequests \
    --dimensions Name=TableName,Value=myTable \
    --statistic Sum \
    --threshold 0 \
    --comparison-operator GreaterThanThreshold \
    --period 300 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:requests-exceeding-throughput
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --state-reason "initializing" --state-value OK
```

```
Prompt>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.

```
Prompt>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 \
    --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.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
"initializing" --state-value OK
```

```
Prompt>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 CloudTrail, a service that captures low-level API requests made by or on behalf of DynamoDB in your AWS account and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures calls made from the DynamoDB console or from the DynamoDB low-level API. Using the information collected by CloudTrail, you can determine what request was made to DynamoDB, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

DynamoDB Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, low-level API calls made to DynamoDB actions are tracked in log files. DynamoDB records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

The following API actions are supported:

Amazon DynamoDB

- [CreateTable](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [DescribeTimeToLive](#)
- [ListTables](#)
- [UpdateTable](#)
- [UpdateTimeToLive](#)
- [DescribeReservedCapacity](#)

- [DescribeReservedCapacityOfferings](#)
- [PurchaseReservedCapacityOfferings](#)

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](#)

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the [userIdentity](#) field in the [CloudTrail Event Reference](#).

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon SNS notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see [Configuring Amazon SNS Notifications](#).

You can also aggregate DynamoDB log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see [Receiving CloudTrail Log Files from Multiple Regions](#).

Understanding DynamoDB Log File Entries

CloudTrail log files contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the low-level DynamoDB API calls.

The following example shows a CloudTrail log.

```
{"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": "2015-05-01T07:24:55Z", "eventSource": "dynamodb.amazonaws.com", "eventName": "CreateTable", "awsRegion": "us-west-2", "sourceIPAddress": "192.0.2.0", "userAgent": "console.aws.amazon.com", "requestParameters": { "provisionedThroughput": { "writeCapacityUnits": 10, "readCapacityUnits": 10 } }, "tableName": "Music", "keySchema": [ { "attributeName": "Artist", "keyType": "HASH" }, { "attributeName": "SongTitle", "keyType": "RANGE" } ], "attributeDefinitions": [ { "attributeType": "S", "attributeName": "Artist" }, { "attributeType": "S", "attributeName": "SongTitle" } ] } ] }
```

```

        }
    ],
},
"responseElements": {"tableDescription": {
    "tableName": "Music",
    "attributeDefinitions": [
        {
            "attributeType": "S",
            "attributeName": "Artist"
        },
        {
            "attributeType": "S",
            "attributeName": "SongTitle"
        }
    ],
    "itemCount": 0,
    "provisionedThroughput": {
        "writeCapacityUnits": 10,
        "numberOfDecreasesToday": 0,
        "readCapacityUnits": 10
    },
    "creationDateTime": "May 1, 2015 7:24:55 AM",
    "keySchema": [
        {
            "attributeName": "Artist",
            "keyType": "HASH"
        },
        {
            "attributeName": "SongTitle",
            "keyType": "RANGE"
        }
    ],
    "tableStatus": "CREATING",
    "tableSizeBytes": 0
}},
"requestID": "KAVGJR1Q0I5VHF8FS8V809EV7FVV4KONSO5AEMVJF66Q9ASUAAJG",
"eventID": "a8b5f864-480b-43bf-bc22-9b6d77910a29",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
},
{
    "eventVersion": "1.03",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
        "accountId": "444455556666",
        "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": "2015-05-04T02:43:11Z",
    "eventSource": "dynamodb.amazonaws.com",

```

```

        "eventName": "DescribeTable",
        "awsRegion": "us-west-2",
        "sourceIPAddress": "192.0.2.0",
        "userAgent": "console.aws.amazon.com",
        "requestParameters": {"tableName": "Music"},
        "responseElements": null,
        "requestID": "DISTSH6DQRLCC74L48Q51LRBFVV4KQNSO5AEMVJF66Q9ASUAAJG",
        "eventID": "c07befa7-f402-4770-8c1b-1911601ed2af",
        "eventType": "AwsApiCall",
        "apiVersion": "2012-08-10",
        "recipientAccountId": "111122223333"
    },
    {
        "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": "2015-05-04T02:14:52Z",
        "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": "AALNP0J2L244N5O15PKISJ1KUFVV4KQNSO5AEMVJF66Q9ASUAAJG",
"eventID": "eb834e01-f168-435f-92c0-c36278378b6e",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
},
{
    "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": "AKIAI44QH8DHBEEXAMPLE",
                "arn": "arn:aws:iam::444455556666:role/admin-role",
                "accountId": "444455556666",
                "userName": "bob"
            }
        }
    },
    "eventTime": "2015-05-04T02:42:20Z",
    "eventSource": "dynamodb.amazonaws.com",
    "eventName": "ListTables",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "console.aws.amazon.com",
    "requestParameters": null,
    "responseElements": null,
    "requestID": "3BGHST50VHLMTPUMAUTA1RF4M3VV4KQNSO5AEMVJF66Q9ASUAAJG",
    "eventID": "bd5bf4b0-b8a5-4bec-9edf-83605bd5e54e",
    "eventType": "AwsApiCall",
    "apiVersion": "2012-08-10",
    "recipientAccountId": "111122223333"
},
{
    "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": "AKIAI44QH8DHBEEXAMPLE",
        "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": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
"eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
}
]}
]
```

Best Practices for DynamoDB

Use this section to quickly find recommendations for maximizing performance and minimizing throughput costs.

Table Best Practices

DynamoDB tables are distributed across multiple partitions. For best results, design your tables and applications so that read and write activity is spread evenly across all of the items in your tables, and avoid I/O "hot spots" that can degrade performance.

- [Design For Uniform Data Access Across Items In Your Tables \(p. 667\)](#)
- [Understand Partition Behavior \(p. 669\)](#)
- [Use Burst Capacity Sparingly \(p. 673\)](#)
- [Distribute Write Activity During Data Upload \(p. 673\)](#)
- [Understand Access Patterns for Time Series Data \(p. 674\)](#)
- [Cache Popular Items \(p. 674\)](#)
- [Consider Workload Uniformity When Adjusting Provisioned Throughput \(p. 675\)](#)
- [Test Your Application At Scale \(p. 676\)](#)

Item Best Practices

DynamoDB items are limited in size (see [Limits in DynamoDB \(p. 731\)](#)). However, there is no limit on the number of items in a table. Rather than storing large data attribute values in an item, consider one or more of these application design alternatives.

- [Use One-to-Many Tables Instead Of Large Set Attributes \(p. 677\)](#)
- [Compress Large Attribute Values \(p. 677\)](#)
- [Store Large Attribute Values in Amazon S3 \(p. 677\)](#)

- [Break Up Large Attributes Across Multiple Items \(p. 678\)](#)

Query and Scan Best Practices

Sudden, unexpected read activity can quickly consume the provisioned read capacity of a table or a global secondary index. In addition, such activity can be inefficient if it is not evenly spread across table partitions.

- [Performance Considerations for Scans \(p. 679\)](#)
- [Avoid Sudden Spikes in Read Activity \(p. 679\)](#)
- [Take Advantage of Parallel Scans \(p. 681\)](#)

Local Secondary Index Best Practices

A local secondary index lets you define an alternative sort key for your data. You can query a local secondary index in the same way that you query a table. Before using local secondary indexes, you should be aware of the inherent tradeoffs in terms of provisioned throughput costs, storage costs, and query efficiency.

- [Use Indexes Sparingly \(p. 682\)](#)
- [Choose Projections Carefully \(p. 682\)](#)
- [Optimize Frequent Queries To Avoid Fetches \(p. 683\)](#)
- [Take Advantage of Sparse Indexes \(p. 683\)](#)
- [Watch For Expanding Item Collections \(p. 683\)](#)

Global Secondary Index Best Practices

Global Secondary Indexes let you define alternative partition key and sort key attributes for your data. These attributes don't have to be the same as the table's partition key and sort key. You can query a global secondary index in the same way that you query a table. As with local secondary indexes, global secondary indexes also present tradeoffs that you need to consider when designing your applications.

- [Choose a Key That Will Provide Uniform Workloads \(p. 684\)](#)
- [Take Advantage of Sparse Indexes \(p. 684\)](#)
- [Use a Global Secondary Index For Quick Lookups \(p. 685\)](#)
- [Create an Eventually Consistent Read Replica \(p. 685\)](#)

Best Practices for Tables

Topics

- [Design For Uniform Data Access Across Items In Your Tables \(p. 667\)](#)
- [Understand Partition Behavior \(p. 669\)](#)
- [Use Burst Capacity Sparingly \(p. 673\)](#)
- [Distribute Write Activity During Data Upload \(p. 673\)](#)

- [Understand Access Patterns for Time Series Data \(p. 674\)](#)
- [Cache Popular Items \(p. 674\)](#)
- [Consider Workload Uniformity When Adjusting Provisioned Throughput \(p. 675\)](#)
- [Test Your Application At Scale \(p. 676\)](#)

This section covers some best practices for working with tables.

Design For Uniform Data Access Across Items In Your Tables

The optimal usage of a table's provisioned throughput depends on these factors:

- The primary key selection.
- The workload patterns on individual items.

The primary key uniquely identifies each item in a table. The primary key can be simple (partition key) or composite (partition key and sort key).

When it stores data, DynamoDB divides a table's items into multiple partitions, and distributes the data primarily based upon the partition key value. Consequently, to achieve the full amount of request throughput you have provisioned for a table, keep your workload spread evenly across the partition key values. Distributing requests across partition key values distributes the requests across partitions.

For example, if a table has a very small number of heavily accessed partition key values, possibly even a single very heavily used partition key value, request traffic is concentrated on a small number of partitions – potentially only one partition. If the workload is heavily unbalanced, meaning that it is disproportionately focused on one or a few partitions, the requests will not achieve the overall provisioned throughput level. To get the most out of DynamoDB throughput, create tables where the partition key has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible.

This does not mean that you must access all of the partition key values to achieve your throughput level; nor does it mean that the percentage of accessed partition key values needs to be high. However, be aware that when your workload accesses more distinct partition key values, those requests will be spread out across the partitioned space in a manner that better utilizes your allocated throughput level. In general, you will utilize your throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values in a table grows.

Choosing a Partition Key

The following table compares some common partition key schemas for provisioned throughput efficiency:

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 (e.g. day, hour, minute)	Bad

Partition key value	Uniformity
Device ID, where each device accesses data at relatively similar intervals	Good
Device ID, where even if there are a lot of devices being tracked, one is by far more popular than all the others.	Bad

If a single table has only a very 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 will be written to that same partition key value.

If the table will fit entirely into a single partition (taking into consideration growth of your data over time), and if your application's read and write throughput requirements do not exceed the read and write capabilities of a single partition, then your application should not encounter any unexpected throttling as a result of partitioning.

However, if you anticipate scaling beyond a single partition, then you should architect your application so that it can use more of the table's full provisioned throughput.

Randomizing Across Multiple Partition Key Values

One way to increase the write throughput of this application would be to randomize the writes across multiple partition key values. Choose a random number from a fixed set (for example, 1 to 200) and concatenate it as a suffix to the date. This will yield partition key values such as 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 all of the partition key values; this will yield better parallelism and higher overall throughput.

To read all of the items for a given day, you would need to obtain all of the items for each suffix. For example, you would first issue a `Query` request for the partition key value 2014-07-09.1, then another `Query` for 2014-07-09.2, and so on through 2014-07-09.200. Finally, your application would need to merge the results from all of the `Query` requests.

Using a Calculated Value

A randomizing strategy can greatly improve write throughput; however, it is 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 are able to calculate based upon something that's intrinsic to the item.

Continuing with our example, suppose that each item has an `orderId`. Before your application writes the item to the table, it can calculate a partition key suffix based upon the order ID. The calculation should result in a number between 1 and 200 that is fairly evenly distributed given any set of names (or user IDs.)

A simple calculation would 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 as a suffix. With this strategy, the writes are spread evenly across the partition key values, and thus across the partitions. You can easily perform a `GetItem` operation on a particular item, because you can calculate the partition key value you need when you want to retrieve a specific `orderId` value.

To read all of the items for a given day, you would still need to query each of the $2014-07-09.N$ keys (where N is 1 to 200), and your application would need to merge all of the results. However, you will avoid having a single "hot" partition key value taking all of the workload.

Understand Partition Behavior

DynamoDB manages table partitioning for you automatically, adding new partitions if necessary and distributing provisioned throughput capacity evenly across them.

You can estimate the number of partitions that DynamoDB will initially allocate for your table, and compare that estimate against your scale and access patterns. You can also estimate the number of additional partitions that DynamoDB will allocate in response to increased storage or provisioned throughput requirements. These estimates can help you determine the best table design for your application needs.

Note

The following details about partition sizes and throughput are subject to change.

Initial Allocation of Partitions

When you create a new table, DynamoDB allocates the table's partitions according to the provisioned throughput settings that you specify.

A single partition can support a maximum of 3,000 read capacity units or 1,000 write capacity units. When you create a new table, the initial number of partitions can be expressed as follows:

$$(\text{readCapacityUnits} / 3,000) + (\text{writeCapacityUnits} / 1,000) = \text{initialPartitions (rounded up)}$$

For example, suppose that you created a table with 1,000 read capacity units and 500 write capacity units. In this case, the initial number of partitions would be:

$$(1,000 / 3,000) + (500 / 1,000) = 0.8333 \rightarrow 1$$

Therefore, a single partition could accommodate all of the table's provisioned throughput requirements.

However, if you had created the table with 1,000 read capacity units and 1,000 write capacity units, then a single partition would not be able to support the specified throughput capacity:

$$(1,000 / 3,000) + (1,000 / 1,000) = 1.333 \rightarrow 2$$

In this case, the table would require two partitions, each with 500 read capacity units and 500 write capacity units.

Subsequent Allocation of Partitions

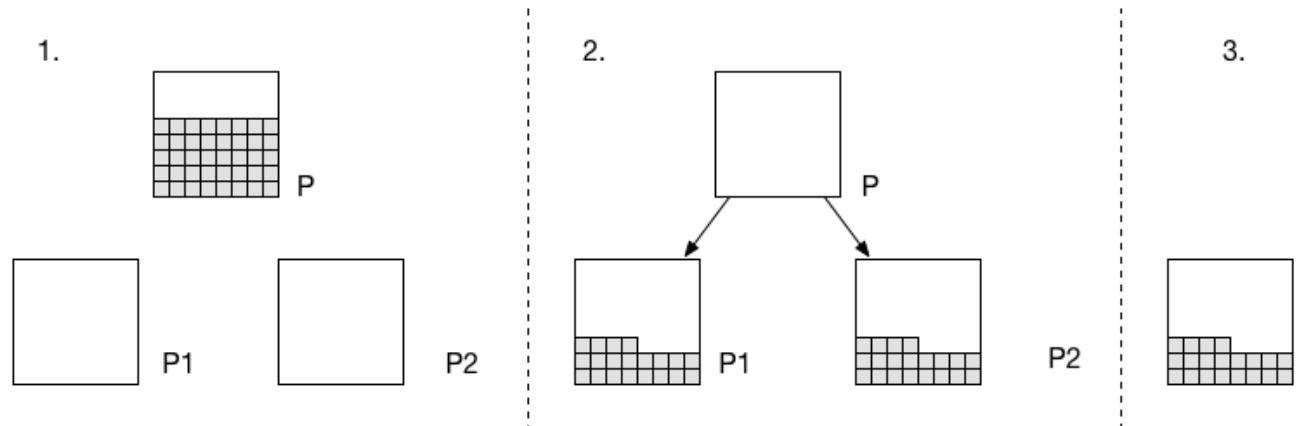
A single partition can hold approximately 10 GB of data, and can support a maximum of 3,000 read capacity units or 1,000 write capacity units.

If necessary, DynamoDB can allocate additional partitions to your table by *splitting* an existing partition. Suppose that one of a table's partitions (**P**) exceeded its limit for storage (10 GB). In this case, DynamoDB would split the partition as follows:

1. Allocate two new partitions (**P1** and **P2**).
2. Distribute the data from **P** evenly across **P1** and **P2**.

3. Deallocate P from the table.

The following diagram shows how DynamoDB performs a partition split. The large squares represent partitions, and the small squares represent data items in the table.



During a partition split, DynamoDB evenly distributes the data from the old partition to the two new partitions (the data in other partitions are not affected). The old partition's provisioned throughput capacity is then evenly distributed across the two new partitions (see [Throughput Capacity Per Partition \(p. 672\)](#)).

Note that DynamoDB performs partition splits automatically, in the background. The table remains fully available for read and write activity at your specified throughput levels.

A partition split can occur in response to:

- Increased provisioned throughput settings
- Increased storage requirements

Increased Provisioned Throughput Settings

If you increase a table's provisioned throughput and the table's current partitioning scheme cannot accommodate your new requirements, DynamoDB will *double* the current number of partitions.

For example, suppose that you created a new table with 5,000 read capacity units and 2,000 write capacity units. Using the information from [Initial Allocation of Partitions \(p. 669\)](#), you can determine that this new table will require four partitions:

$$(5,000 / 3,000) + (2,000 / 1,000) = 3.6667 \rightarrow 4$$

Each of the four partitions can accommodate 1,250 reads per second (5,000 read capacity units / 4 partitions) and 500 writes per second (2,000 write capacity units / 4 partitions).

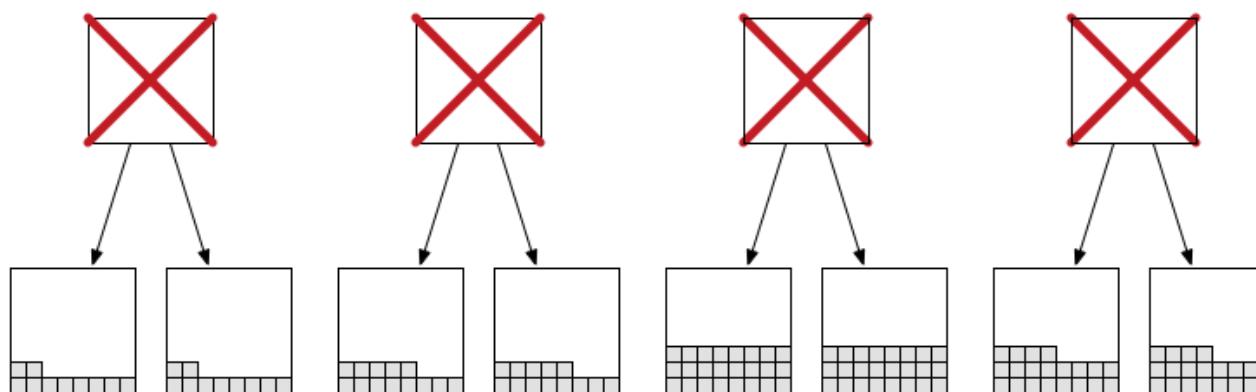
Now suppose that you increased the table's read capacity units from 5,000 to 8,000. The existing four partitions would not be able to support this requirement. In response (see [Subsequent Allocation of Partitions \(p. 669\)](#)), DynamoDB would double the number of partitions to eight ($4 * 2 = 8$). Each of the resulting partitions would be able to accommodate 1,000 reads per second (8,000 read capacity units / 8 partitions) and 250 writes per second (2,000 write capacity units / 8 partitions).

The following diagram shows the original four partitions in the table, and the resulting partition scheme after DynamoDB doubles the number of partitions. The large squares represent partitions, and the small squares represent data items in the table.

Four partitions with 1,250 read capacity units and 500 write capacity units each



Eight partitions with 1,000 read capacity units and 250 write capacity units each



Increased Storage Requirements

If an existing partition fills up with data, DynamoDB will split that partition. The result will be two partitions, with the data from the old partition divided equally between the new partitions.

The table described in [Increased Provisioned Throughput Settings \(p. 670\)](#) has eight partitions, so its maximum capacity would be approximately 80 GB, as shown following:

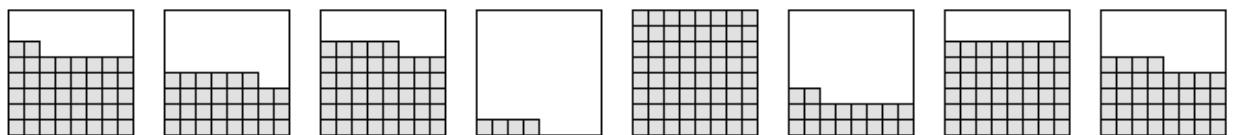
$$8 \text{ partitions} * 10 \text{ GB} = 80 \text{ GB}$$

If one of these partitions were to fill to capacity, DynamoDB would respond by splitting that partition, for a total of nine partitions and an overall capacity of 90 GB, as shown following:

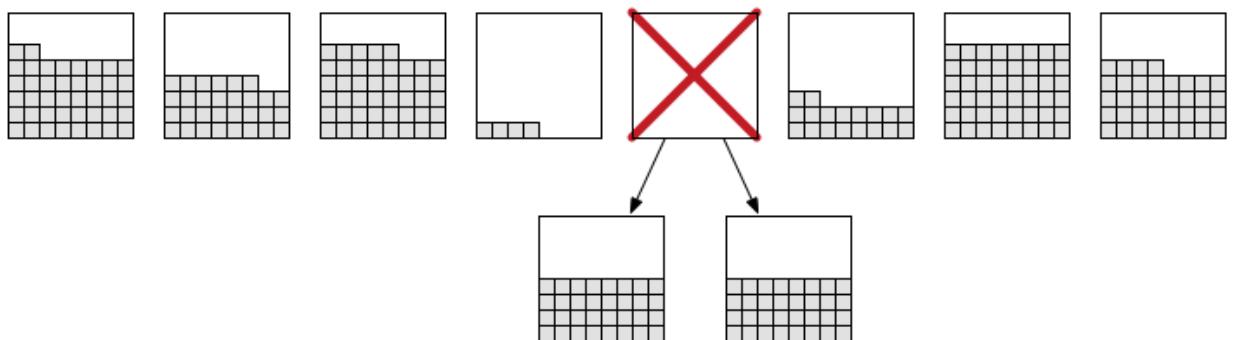
$$9 \text{ partitions} * 10 \text{ GB} = 90 \text{ GB}$$

The following diagram shows one of the original partitions filling to capacity, and the resulting partition scheme after DynamoDB splits that partition. The large squares represent partitions, and the small squares represent data items in the table.

1.



2.



Throughput Capacity Per Partition

If you estimate the number of partitions in your table, you can determine the approximate throughput capacity per partition. Suppose that you wanted to create a table with 5,000 read capacity units and 2,000 write capacity units. DynamoDB would allocate four partitions for the new table:

$$(5,000 / 3,000) + (2,000 / 1,000) = 3.6667 \rightarrow 4$$

You could determine the amount of read and write capacity per partition as follows:

```
5,000 read capacity units / 4 partitions = 1,250 read capacity units per partition
2,000 write capacity units / 4 partitions = 500 write capacity units per partition
```

Now suppose that one of these four partitions were to fill to capacity. DynamoDB would split that partition, resulting in five partitions allocated to the table. The read and write capacity per partition would then be distributed as follows:

```
1,250 read capacity units / 2 partitions = 625 read capacity units per child partition
500 write capacity units / 2 partitions = 250 write capacity units per child partition
```

As a result:

- Three of the five partitions would each have 1,250 read capacity units and 500 write capacity units.
- The other two partitions would each have 625 read capacity units and 250 write capacity units.

Note that as the number of partitions in a table increases, each partition has fewer read and write capacity units available to it. (However, the total provisioned throughput for the table remains the same.)

Use Burst Capacity Sparingly

DynamoDB provides some flexibility in the per-partition throughput provisioning. When you are not fully utilizing a partition's throughput, DynamoDB retains a portion of your unused capacity for later *bursts* of throughput usage. DynamoDB currently retains up to five 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 very quickly—even faster than the per-second provisioned throughput capacity that you've defined for your table. However, do not design your application so that it depends on burst capacity being available at all times: DynamoDB can and does use burst capacity for background maintenance and other tasks without prior notice.

Note

In the future, these details of burst capacity may change.

Distribute Write Activity During Data Upload

There are times when you load data from other data sources into DynamoDB. Typically, DynamoDB partitions your table data on multiple servers. When uploading data to a table, you get better performance if you upload data to all the allocated servers simultaneously. For example, suppose you want to upload user messages to a DynamoDB table. You might design a table that uses a composite primary key in which UserID is the partition key and the MessageID is the sort key. When uploading data from your source, you might tend to read all message items for a specific user and upload these items to DynamoDB as shown in the sequence in the following table.

UserID	MessageID
U1	1
U1	2
U1	...
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 its items, before going to the next partition key value and doing the same. Behind the scenes, DynamoDB is partitioning the data in your tables across multiple servers. To fully utilize all of the throughput capacity that has been provisioned for your tables, you need to distribute your workload across your partition key values. In this case, by directing an uneven amount of upload work toward items all with the same partition key value, you may not be able to fully utilize all of the resources DynamoDB has provisioned for your table. You can distribute your upload work by uploading one item from each partition key value first. Then you repeat the pattern for the next set of sort key values for all the items until you upload all the data as shown in the example upload sequence in the following table:

UserID	MessageID
U1	1
U2	1
U3	1
...
U1	2
U2	2
U3	2
...	...

Every upload in this sequence uses a different partition key value, keeping more DynamoDB servers busy simultaneously and improving your throughput performance.

Understand Access Patterns for Time Series Data

For each table that you create, you specify the throughput requirements. DynamoDB allocates resources to handle your throughput requirements with sustained low latency. When you design your application and tables, you should consider your application's access pattern to make the most efficient use of your table's resources.

Suppose you design a table to track customer behavior on your site, such as URLs that they click. You might design the table with a composite primary key consisting of Customer ID as the partition key and date/time as the sort key. In this application, customer data grows indefinitely over time; however, the applications might show uneven access pattern across all the items in the table where the latest customer data is more relevant and your application might access the latest items more frequently and as time passes these items are less accessed, eventually the older items are rarely accessed. If this is a known access pattern, you could take it into consideration when designing your table schema. Instead of storing all items in a single table, you could use multiple tables to store these items. For example, you could create tables to store monthly or weekly data. For the table storing data from the latest month or week, where data access rate is high, request higher throughput and for tables storing older data, you could dial down the throughput and save on resources.

You can save on resources by storing "hot" items in one table with higher throughput settings, and "cold" items in another table with lower throughput settings. You can remove old items by simply deleting the tables. You can optionally backup these tables to other storage options such as Amazon Simple Storage Service (Amazon S3). Deleting an entire table is significantly more efficient than removing items one-by-one, which essentially doubles the write throughput as you do as many delete operations as put operations.

Cache Popular Items

Some items in a table might be more popular than others. For example, consider the *ProductCatalog* table that is described in [Creating Tables and Loading Sample Data \(p. 280\)](#), and suppose that this table contains millions of different products. Some products might be very popular among customers, so those items would be consistently accessed more frequently than the others. As a result, the distribution of read activity on *ProductCatalog* would be highly skewed toward those popular items.

One solution would be to cache these reads at the application layer. Caching is a technique that is used in many high-throughput applications, offloading read activity on hot items to the cache rather than to

the database. Your application can cache the most popular items in memory, or use a product such as [ElastiCache](#) to do the same.

Continuing with the *ProductCatalog* example, when a customer requests an item from that table, the application would first consult the cache to see if there is a copy of the item there. If so, it is a cache hit; otherwise, it is a cache miss. When there is a cache miss, the application would need to read the item from DynamoDB and store a copy of the item in the cache. Over time, the cache misses would decrease as the cache fills with the most popular items; applications would not need to access DynamoDB at all for these items.

A caching solution can mitigate the skewed read activity for popular items. In addition, since it reduces the amount of read activity against the table, caching can help reduce your overall costs for using DynamoDB.

Consider Workload Uniformity When Adjusting Provisioned Throughput

As the amount of data in your table grows, or as you provision additional read and write capacity, DynamoDB automatically spreads the data across multiple partitions. If your application doesn't require as much throughput, you simply decrease it using the `UpdateTable` operation, and pay only for the throughput that you have provisioned.

For applications that are designed for use with uniform workloads, DynamoDB's partition allocation activity is not noticeable. A temporary non-uniformity in a workload can generally be absorbed by the bursting allowance, as described in [Use Burst Capacity Sparingly \(p. 673\)](#). However, if your application must accommodate non-uniform workloads on a regular basis, you should design your table with DynamoDB's partitioning behavior in mind (see [Understand Partition Behavior \(p. 669\)](#)), and be mindful when increasing and decreasing provisioned throughput on that table.

If you reduce the amount of provisioned throughput for your table, DynamoDB will not decrease the number of partitions. Suppose that you created a table with a much larger amount of provisioned throughput than your application actually needed, and then decreased the provisioned throughput later. In this scenario, the provisioned throughput per partition would be less than it would have been if you had initially created the table with less throughput.

For example, consider a situation where you need to bulk-load 20 million items into a DynamoDB table. Assume that each item is 1 KB in size, resulting in 20 GB of data. This bulk-loading task will require a total of 20 million write capacity units. To perform this data load within 30 minutes, you would need to set the provisioned write throughput of the table to 11,000 write capacity units.

The maximum write throughput of a partition is 1000 write capacity units (see [Understand Partition Behavior \(p. 669\)](#)); therefore, DynamoDB will create 11 partitions, each with 1000 provisioned write capacity units.

After the bulk data load, your steady-state write throughput requirements might be much lower — for example, suppose that your applications only require 200 writes per second. If you decrease the table's provisioned throughput to this level, each of the 11 partitions will have around 20 write capacity units per second provisioned. This level of per-partition provisioned throughput, combined with DynamoDB's bursting behavior, might be adequate for the application.

However, if an application will require sustained write throughput above 20 writes per second per partition, you should either: (a) design a schema that requires fewer writes per second per partition key value, or (b) design the bulk data load so that it runs at a slower pace and reduces the initial throughput requirement. For example, suppose that it was acceptable to run the bulk import for over 3 hours, instead of just 30 minutes. In this scenario, only 1900 write capacity units per second needs to be provisioned, rather than 11,000. As a result, DynamoDB would create only two partitions for the table.

Test Your Application At Scale

Many tables begin with small amounts of data, but then grow larger as applications perform write activity. This growth can occur gradually, without exceeding the provisioned throughput settings you have defined for the table. As your table grows larger, DynamoDB automatically scales your table out by distributing the data across more partitions. When this occurs, the provisioned throughput that is allocated to each resulting partition is less than that which is allocated for the original partition(s).

Suppose that your application accesses the table's data across all of the partition key values, but in a non-uniform fashion (accessing a small number of partition key values more frequently than others). Your application might perform acceptably when there is not very much data in the table. However, as the table becomes larger, there will be more partitions and less throughput per partition. You might discover that your application is throttled when it attempts to use the same non-uniform access pattern that worked in the past.

To avoid problems with "hot" keys when your table becomes larger, make sure that you test your application design at scale. Consider the ratio of storage to throughput when running at scale, and how DynamoDB will allocate partitions to the table. (For more information, see [Understand Partition Behavior \(p. 669\)](#).)

If it isn't possible for you to generate a large amount of test data, you can create a table that has very high provisioned throughput settings. This will create a table with many partitions; you can then use `UpdateTable` to reduce the settings, but keep the same ratio of storage to throughput that you determined for running the application at scale. You now have a table that has the throughput-per-partition ratio that you expect after it grows to scale. Test your application against this table using a realistic workload.

Tables that store time series data can grow in an unbounded manner, and can cause slower application performance over time. With time series data, applications typically read and write the most recent items in the table more frequently than older items. If you can remove older time series data from your real-time table, and archive that data elsewhere, you can maintain a high ratio of throughput per partition.

For best practices with time series data, [Understand Access Patterns for Time Series Data \(p. 674\)](#).

Best Practices for Items

Topics

- [Use One-to-Many Tables Instead Of Large Set Attributes \(p. 677\)](#)
- [Compress Large Attribute Values \(p. 677\)](#)
- [Store Large Attribute Values in Amazon S3 \(p. 677\)](#)
- [Break Up Large Attributes Across Multiple Items \(p. 678\)](#)

When you are working with items in DynamoDB, you need to consider how to get the best performance, how to reduce provisioned throughput costs, and how to avoid throttling by staying within your read and write capacity units. If the items that you are handling exceed the maximum item size, as described in [Limits in DynamoDB \(p. 731\)](#), you need to consider how you will deal with the situation. This section offers best practices for addressing these considerations.

Use One-to-Many Tables Instead Of Large Set Attributes

If your table has items that store a large set type attribute, such as number set or string set, consider removing the attribute and breaking the table into two tables. To form one-to-many relationships between these tables, use the primary keys.

The Forum, Thread, and Reply tables in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section are good examples of these one-to-many relationships. For example, the Thread table has one item for each forum thread, and the Reply table stores one or more replies for each thread.

Instead of storing replies as items in a separate table, you could store both threads and replies in the same table. For each thread, you could store all replies in an attribute of string set type; however, keeping thread and reply data in separate tables is beneficial in several ways:

- If you store replies as items in a table, you can store any number of replies, because a DynamoDB table can store any number of items.

If you store replies as an attribute value in the Thread table, you would be constrained by the maximum item size, which would limit the number of replies that you could store. (See [Limits in DynamoDB \(p. 731\)](#))

- When you retrieve a Thread item, you pay less for provisioned throughput, because you are retrieving only the thread data and not all the replies for that thread.
- Queries allow you to retrieve only a subset of items from a table. By storing replies in a separate Reply table, you can retrieve only a subset of replies, for example, those within a specific date range, by querying the Reply table.

If you store replies as a set type attribute value, you would always have to retrieve all the replies, which would consume more provisioned throughput for data that you might not need.

- When you add a new reply to a thread, you add only an item to the Reply table, which incurs the provisioned throughput cost of only that single Reply item. If replies are stored in the Thread table, you incur the full cost of writing the entire Thread item including all replies whenever you add a single user reply to a thread.

Compress Large Attribute Values

You can compress large values before storing them in DynamoDB. Doing so can reduce the cost of storing and retrieving such data. Compression algorithms, such as GZIP or LZO, produce a binary output. You can then store this output in a Binary attribute type.

For example, the Reply table in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section stores messages written by forum users. These user replies might consist of very long strings of text, which makes them excellent candidates for compression.

For sample code that demonstrates how to compress long messages in DynamoDB, see:

- [Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API \(p. 385\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API \(p. 407\)](#)

Store Large Attribute Values in Amazon S3

DynamoDB currently limits the size of the items that you store in tables. For more information, see [Limits in DynamoDB \(p. 731\)](#). Your application, however, might need to store more data in an item than the

DynamoDB size limits permit. To work around this issue, you can store the large attributes as an object in Amazon Simple Storage Service (Amazon S3), and store the object identifier in your item. You can also use the object metadata support in Amazon S3 to store the primary key value of the corresponding item as Amazon S3 object metadata. This use of metadata can help with future maintenance of your Amazon S3 objects.

For example, consider the *ProductCatalog* table in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. Items in the *ProductCatalog* table store information about item price, description, authors for books, and dimensions for other products. If you wanted to store an image of each product, these images could be large. It might make sense to store the images in Amazon S3 instead of DynamoDB.

There are important considerations with this approach:

- Since DynamoDB does not support transactions that cross Amazon S3 and DynamoDB, your application will have to deal with failures and with cleaning up orphaned Amazon S3 objects.
- Amazon S3 limits length of object identifiers, so you must organize your data in a way that accommodates this and other Amazon S3 constraints. For more information, see the [Amazon Simple Storage Service Developer Guide](#).

Break Up Large Attributes Across Multiple Items

If you need to store more data in a single item than DynamoDB allows, you can store that data in multiple items as chunks of a larger "virtual item". To get the best results, store the chunks in a separate table that has a simple primary key (partition key), and use batch operations (`BatchGetItem` and `BatchWriteItem`) to read and write the chunks. This approach will help you to spread your workload evenly across table partitions.

For example, consider the *Forum*, *Thread* and *Reply* tables described in the [Creating Tables and Loading Sample Data \(p. 280\)](#) section. Items in the *Reply* table contain forum messages that were written by forum users. Due to the 400 KB item size limit in DynamoDB, the length of each reply is also limited. For large replies, instead of storing one item in the *Reply* table, break the reply message into chunks, and then write each chunk into its own separate item in a new *ReplyChunks* table with a simple primary key (partition key).

The primary key of each chunk would be a concatenation of the primary key of its "parent" reply item, a version number, and a sequence number. The sequence number determines the order of the chunks. The version number ensures that if a large reply is updated later, it will be updated atomically. In addition, chunks that were created before the update will not be mixed with chunks that were created after the update.

You would also need to update the "parent" reply item with the number of chunks, so that when you need to retrieve all the chunks for a reply, you will know how many chunks to look for.

As an illustration, here is how these items might appear in the *Reply* and *ReplyChunks* tables:

Reply

<u>Id</u>	<u>ReplyDateTime</u>	<u>Message</u>	<u>ChunkCount</u>	<u>ChunkVersion</u>
"DynamoDB#Thread1"	2012-03-15T20:42:54.023Z"		3	1
"DynamoDB#Thread2"	2012-03-21T20:41:23h10m27s"message"			

ReplyChunks

Id	Message
"DynamoDB#Thread1#2012-03-15T20:42:54.023Z#1#first part of long message text..."	
"DynamoDB#Thread1#2012-03-15T20:42:54.023Z#1#third part of long message text"	
"DynamoDB#Thread1#2012-03-15T20:42:54.023Z#1#second part of long message text..."	

Here are important considerations with this approach:

- Because DynamoDB does not support cross-item transactions, your application will need to deal with failure scenarios when writing multiple items and with inconsistencies between items when reading multiple items.
- If your application retrieves a large amount of data all at once, it can generate nonuniform workloads, which can cause unexpected throttling. This is especially true when retrieving items that share a partition key value.

Chunking large data items avoids this problem by using a separate table with a simple primary key (partition key), so that each large chunk is spread across the table.

A workable, but less optimal, solution would be to store each chunk in a table with a composite key, with the partition key being the primary key of the "parent" item. With this design choice, an application that retrieves all of the chunks of the same "parent" item would generate a nonuniform workload, with uneven I/O distribution across partitions.

Best Practices for Querying and Scanning Data

This section covers some best practices for `Query` and `Scan` operations.

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, then filters out values to provide the desired result, essentially adding the extra step of removing data from the result set. Avoid using a `Scan` operation on a large table or index with a filter that removes many results, if possible. 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, design your application to use `Scan` operations in a way that minimizes the impact on your request rate.

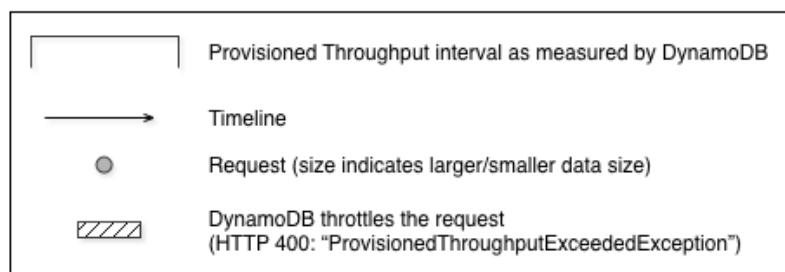
Avoid Sudden Spikes in Read Activity

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 were to 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.

Note that it is not just the sudden increase in capacity units the `Scan` uses that is a problem. It is also because the scan is 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 had been spread across multiple partitions, then the operation would not have throttled a specific partition.

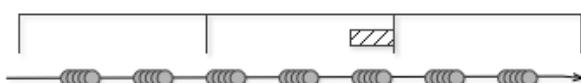
The following diagram illustrates the impact of a sudden spike of capacity unit usage by `Query` and `Scan` operations, and its impact on your other requests against the same table.



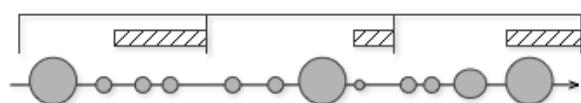
1. Good: Even distribution of requests and size



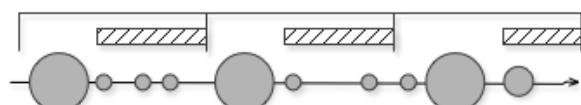
2. Not as Good: Frequent requests in bursts



3. Bad: A few random large requests



4. Bad: Large scan operations



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, if each item is 4 KB and you set the page size to 40 items, then a `Query` request would consume only 40 strongly consistent read operations or 20 eventually 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.

You should 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 \(p. 193\)](#).

Take 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 will have multiple workers that are all running `Scan` operations concurrently, which can very 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 utilized.
- 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 will probably 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 fifteen 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 will allow you to scan that number of segments. If, for example, 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.

You will need to monitor your parallel scans to optimize your provisioned throughput utilization, while also making sure that your other applications aren't starved of resources. Increase the value for

`TotalSegments` if you do not 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 Local Secondary Indexes

Topics

- [Use Indexes Sparingly \(p. 682\)](#)
- [Choose Projections Carefully \(p. 682\)](#)
- [Optimize Frequent Queries To Avoid Fetches \(p. 683\)](#)
- [Take Advantage of Sparse Indexes \(p. 683\)](#)
- [Watch For Expanding Item Collections \(p. 683\)](#)

This section covers some best practices for local secondary indexes.

Use Indexes Sparingly

Don't create local secondary indexes on attributes that you won't often query. Creating and maintaining multiple indexes makes sense for tables that are updated infrequently and are queried using many different criteria. Unused indexes, however, contribute to increased storage and I/O costs, and they do nothing for application performance.

Avoid indexing tables, such as those used in data capture applications, that experience heavy write activity. The cost of I/O operations required to maintain the indexes can be significant. If you need to index the data in such a table, copy the data to another table with any necessary indexes, and query it there.

Choose Projections Carefully

Because local secondary indexes consume storage and provisioned throughput, you should keep the size of the index as small as possible. Additionally, 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 will regularly request.

If you expect a lot of write activity on a table, compared to reads:

- Consider projecting fewer attributes, which will minimize the size of items written to the index. However, if these items are smaller than a single write capacity unit (1 KB), then there won't be any savings in terms of write capacity units. 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.
- If you know that some attributes of that table will rarely be used in queries, then there is no reason to project those attributes. If you subsequently update an attribute that is not in the index, you won't incur the extra cost of updating the index. You can still retrieve non-projected attributes in a Query, but at a higher provisioned throughput cost.

Specify `ALL` only if you want your queries to return the entire table item but you want to sort the table by a different sort key. Indexing all attributes will eliminate the need for table fetches, but in most cases it will double your costs for storage and write activity.

Optimize Frequent Queries To Avoid Fetches

To get the fastest queries with the lowest possible latency, project all of the attributes that you expect those queries to return. If you query an index, but the attributes that you request are not projected, DynamoDB will fetch the requested attributes from the table. To do so, DynamoDB must read the entire item from the table, which introduces latency and additional I/O operations.

If you only issue certain queries only occasionally, and you don't see the need to project all the requested attributes, keep in mind that these "occasional" queries can often turn into "essential" queries! You might regret not projecting those attributes after all.

For more information about table fetches, see [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 490\)](#).

Take Advantage of Sparse Indexes

For any item in a table, DynamoDB will only write a corresponding index entry if the index sort key value is present in the item. If the sort key does not appear in every table item, the index is said to be *sparse*.

Sparse indexes can be beneficial for queries on attributes that don't appear in most table items. For example, suppose that you have a *CustomerOrders* table that stores all of your orders. The key attributes for the table would be as follows:

- Partition key: CustomerId
- Sort key: OrderId

If you want to track only orders that are open, you can have an attribute named *IsOpen*. If you are waiting to receive an order, your application can define *IsOpen* by writing an "X" (or any other value) for that particular item in the table. When the order arrives, your application can delete the *IsOpen* attribute to signify that the order has been fulfilled.

To track open orders, you can create an index on *CustomerId* (partition key) and *IsOpen* (sort key). Only those orders in the table with *IsOpen* defined will appear in the index. Your application can then quickly and efficiently find the orders that are still open by querying the index. If you had thousands of orders, for example, but only a small number that are open, the application can query the index and return the *OrderId* of each open order. Your application will perform significantly fewer reads than it would take to scan the entire *CustomerOrders* table.

Instead of writing an arbitrary value into the *IsOpen* attribute, you can use a different attribute that will result in a useful sort order in the index. To do this, you can create an *OrderOpenDate* attribute and set it to the date on which the order was placed (and still delete the attribute once the order is fulfilled), and create the *OpenOrders* index with the schema *CustomerId* (partition key) and *OrderOpenDate* (sort key). This way when you query your index, the items will be returned in a more useful sort order.

Watch For Expanding Item Collections

An item collection is all the items in a table and its indexes that have the same partition key. An item collection cannot 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 will update any local secondary indexes that are affected. If the indexed attributes are defined in the table, the indexes will grow with the table.

When you create an index, think about how much data will be written to the index, and how much of that data will have the same partition key value. If you expect that the sum of table and index items for

a particular partition key value will exceed 10 GB, you should consider whether you could avoid creating the index.

If you cannot avoid creating the index, then you will need to anticipate the item collection size limit and take action before you exceed it. For strategies on working within the limit and taking corrective action, see [Item Collection Size Limit \(p. 493\)](#).

Best Practices for Global Secondary Indexes

Topics

- [Choose a Key That Will Provide Uniform Workloads \(p. 684\)](#)
- [Take Advantage of Sparse Indexes \(p. 684\)](#)
- [Use a Global Secondary Index For Quick Lookups \(p. 685\)](#)
- [Create an Eventually Consistent Read Replica \(p. 685\)](#)

This section covers some best practices for global secondary indexes.

Choose a Key That Will Provide Uniform Workloads

When you create a DynamoDB table, it's important to distribute the read and write activity evenly across the entire table. To do this, you choose attributes for the primary key so that the data is evenly spread across multiple partitions.

This same guidance is true for global secondary indexes. Choose partition keys and sort keys that have a high number of values relative to the number of items in the index. In addition, remember that global secondary indexes do not enforce uniqueness, so you need to understand the cardinality of your key attributes. *Cardinality* refers to the distinct number of values in a particular attribute, relative to the number of items that you have.

For example, suppose you have an Employee table with attributes such as *Name*, *Title*, *Address*, *PhoneNumber*, *Salary*, and *PayLevel*. Now suppose that you had a global secondary index named *PayLevelIndex*, with *PayLevel* as the partition key. Many companies only have a very small number of pay codes, often fewer than ten, even for companies with hundreds of thousands of employees. Such an index would not provide much benefit, if any, for an application.

Another problem with *PayLevelIndex* is the uneven distribution of distinct values. For example, there may be only a few top executives in the company, but a very large number of hourly workers. Queries on *PayLevelIndex* will not be very efficient because the read activity will not be evenly distributed across partitions.

Take Advantage of Sparse Indexes

For any item in a table, DynamoDB will only write a corresponding entry to a global secondary index if the index key value is present in the item. For global secondary indexes, this is the index partition key and its sort key (if present). If the index key value(s) do not appear in every table item, the index is said to be *sparse*.

You can use a sparse global secondary index to efficiently locate table items that have an uncommon attribute. To do this, you take advantage of the fact that table items that do not contain global secondary index attribute(s) are not indexed at all. For example, in the *GameScores* table, certain players might have earned a particular achievement for a game - such as "Champ" - but most players have not. Rather than scanning the entire *GameScores* table for Champs, you could create a global secondary index with a partition key of *Champ* and a sort key of *UserId*. This would make it easy to find all the Champs by querying the index instead of scanning the table.

Such a query can be very efficient, because the number of items in the index will be significantly fewer than the number of items in the table. In addition, the fewer table attributes you project into the index, the fewer read capacity units you will consume from the index.

Use a Global Secondary Index For Quick Lookups

You can create a global secondary index using any table attributes for the index partition key and sort key. You can even create an index that has exactly the same key attributes as that of the table, and project just a subset of non-key attributes.

One use case for a global secondary index with a duplicate key schema is for quick lookups of table data, with minimal provisioned throughput. If the table has a large number of attributes, and those attributes themselves are large, then every query on that table might consume a large amount of read capacity. If most of your queries do not require that much data to be returned, you can create a global secondary index with a bare minimum of projected attributes - including no projected attributes at all, other than the table's key. This lets you query a much smaller global secondary index, and if you really require the additional attributes, you can then query the table using the same key values.

Create an Eventually Consistent Read Replica

You can create a global secondary index that has the same key schema as the table, with some (or all) of the non-key attributes projected into the index. In your application, you can direct some (or all) read activity to this index, rather than to the table. This lets you avoid having to modify the provisioned read capacity on the table, in response to increased read activity. Note that there will be a short propagation delay between a write to the table and the time that the data appears in the index; your applications should expect eventual consistency when reading from the index.

You can create multiple global secondary indexes to support your application's characteristics. For example, suppose that you have two applications with very different read characteristics — a high-priority app that requires the highest levels of read performance, and a low-priority app that can tolerate occasional throttling of read activity. If both of these apps read from the same table, there is a chance that they could interfere with each other: A heavy read load from the low-priority app could consume all of the available read capacity for the table, which would in turn cause the high-priority app's read activity to be throttled. If you create two global secondary indexes — one with a high provisioned read throughput setting, and the other with a lower setting — you can effectively disentangle these two different workloads, with read activity from each application being redirected to its own index. This approach lets you tailor the amount of provisioned read throughput to each application's read characteristics.

In some situations, you might want to restrict the applications that can read from the table. For example, you might have an application that captures clickstream activity from a website, with frequent writes to a DynamoDB table. You might decide to isolate this table by preventing read access by most applications. (For more information, see [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 625\)](#).) However, if you have other apps that need to perform ad hoc queries on the data, you can create one or more global secondary indexes for that purpose. When you create the index(es), be sure to project only the attributes that your applications actually require. The apps can then read more data while consuming less provisioned read capacity, instead of having to read large items from the table. This can result in a significant cost savings over time.

DynamoDB Integration with Other AWS Services

Amazon DynamoDB is integrated with other AWS services, letting you automate repeating tasks or build applications that span multiple services. For example:

Topics

- [Amazon VPC Endpoints for DynamoDB \(p. 686\)](#)
- [Configure AWS Credentials in Your Files Using Amazon Cognito \(p. 693\)](#)
- [Loading Data From DynamoDB Into Amazon Redshift \(p. 694\)](#)
- [Processing DynamoDB Data With Apache Hive on Amazon EMR \(p. 695\)](#)
- [Exporting and Importing DynamoDB Data Using AWS Data Pipeline \(p. 722\)](#)

Amazon VPC Endpoints for 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.

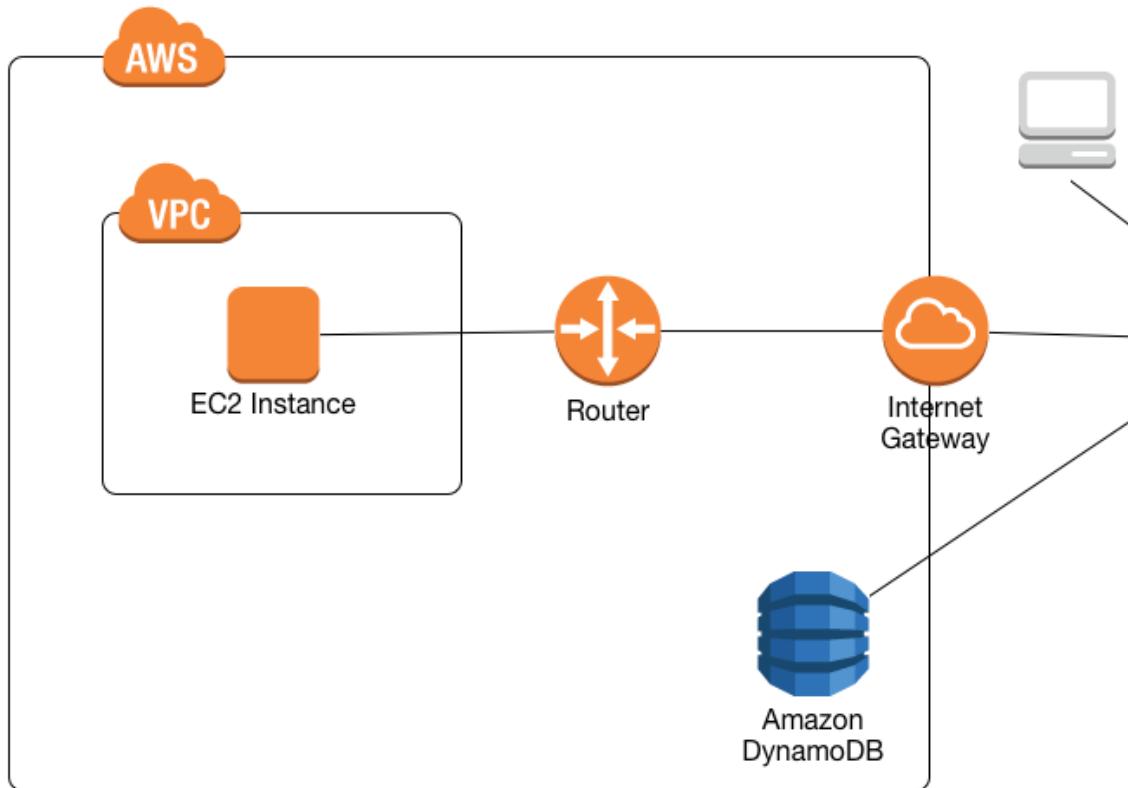
Note

If you created your AWS account after 2013-12-04, 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 your default VPC without having to perform any additional configuration steps.

For more information, see [Your Default VPC and Subnets](#) in the Amazon VPC User Guide.

In order 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 how an EC2 instance in a VPC accesses DynamoDB:

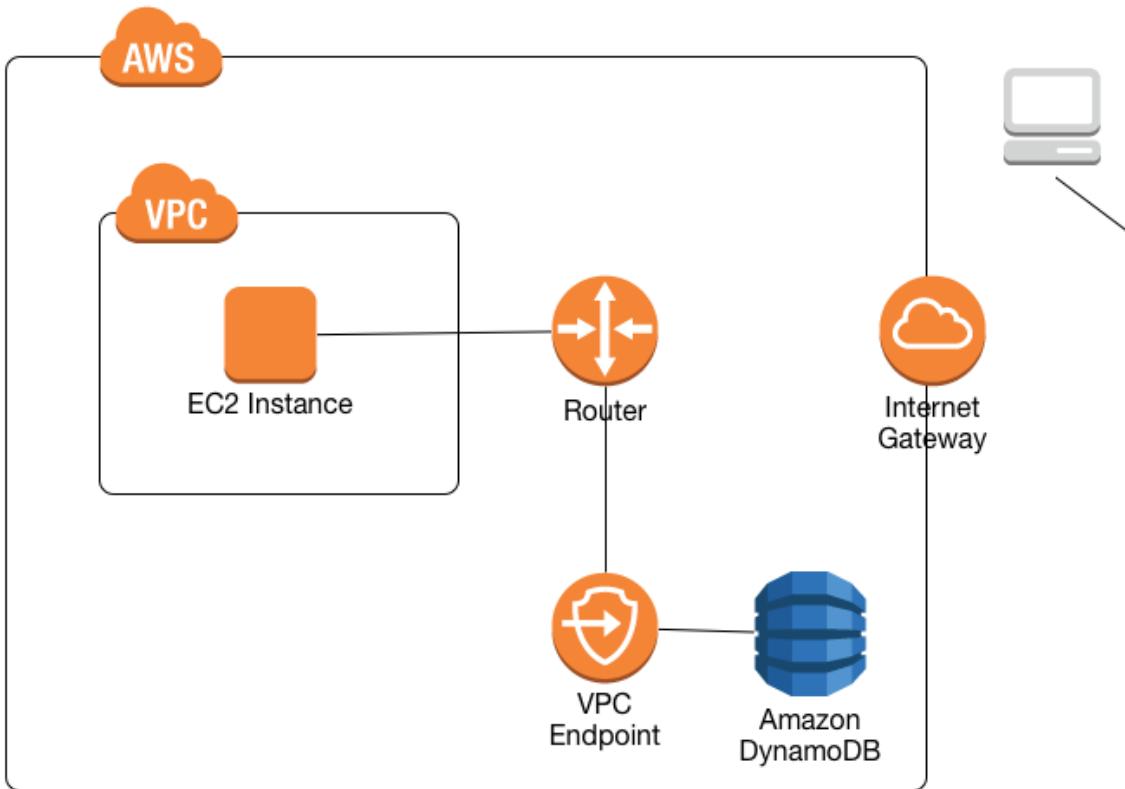


Many customers have legitimate privacy and security concerns about sending and receiving data across the public Internet. Customers can address these concerns by using a virtual private network (VPN) to route all DynamoDB network traffic through the customer's own corporate network infrastructure. However, 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 do not 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.

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 do not 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.



Tutorial: Using a VPC Endpoint for DynamoDB

Topics

- Step 1: Launch an Amazon EC2 Instance (p. 688)
- Step 2: Configure Your Amazon EC2 Instance (p. 690)
- Step 3: Create a VPC Endpoint for DynamoDB (p. 690)
- Step 4: (Optional) Clean Up (p. 692)

This section walks you through setting and using a VPC endpoint for DynamoDB.

Step 1: Launch an Amazon EC2 Instance

In this step, you launch an Amazon EC2 instance in your default Amazon VPC. You will then be able to 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. Note that 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 \(p. 690\)](#)).

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.
 - c. When the settings are as you want them, choose **Create**.
 - d. In the navigation pane, choose **Instances**.
 - e. Choose the Amazon EC2 instance that you launched in [Step 1: Launch an Amazon EC2 Instance \(p. 688\)](#).
 - 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. For 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 \(p. 688\)](#)).

The login ID is `ec2-user`. No password is required.

3. Configure your AWS credentials, as shown below. 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.).

2. 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.

3. 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.

4. Create the VPC endpoint. For the --vpc-id parameter, specify the VPC ID from the previous step.

```
aws ec2 create-vpc-endpoint --vpc-id vpc-0bbc736e --service-name com.amazonaws.us-east-1.dynamodb

{
    "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"
    }
}
```

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 Command Line Interface 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**.

Configure 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. Amazon Cognito uses 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 DynamoDB web service:

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`. Modify 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 the DynamoDB service 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 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 \(p. 696\)](#)
- [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 696\)](#)
- [Creating an External Table in Hive \(p. 703\)](#)
- [Processing HiveQL Statements \(p. 705\)](#)
- [Querying Data in DynamoDB \(p. 706\)](#)
- [Copying Data to and from Amazon DynamoDB \(p. 707\)](#)
- [Performance Tuning \(p. 718\)](#)

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:

- Master 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 master 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 \(p. 697\)](#)
- [Step 1: Create an Amazon EC2 Key Pair \(p. 697\)](#)
- [Step 2: Launch an Amazon EMR Cluster \(p. 697\)](#)
- [Step 3: Connect to the Master Node \(p. 698\)](#)
- [Step 4: Load Data into HDFS \(p. 699\)](#)
- [Step 5: Copy Data to DynamoDB \(p. 700\)](#)

- [Step 6: Query the Data in the DynamoDB Table \(p. 701\)](#)
- [Step 7: \(Optional\) Clean Up \(p. 702\)](#)

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 \(p. 46\)](#).
- An SSH client (Secure Shell). You use the SSH client to connect to the master 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 \(p. 697\)](#)

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 master 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-2 (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 master 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 \(p. 697\)](#)

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/elasticmapreduce/>.
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 Master Node \(p. 698\)](#)

Step 3: Connect to the Master Node

When the status of your Amazon EMR cluster changes to `Waiting`, you will be able to connect to the master 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 **Master public DNS** field. This is the public DNS name for the master 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 Master 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 master node.

After you connect to the master 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 \(p. 699\)](#)

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 http://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 data file contains a subset of data provided by the United States Board on Geographic Names (http://geonames.usgs.gov/domestic/download_data.htm).

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 \(p. 700\)](#)

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:State,p
);
```

You have now established a mapping between Hive and the Features table in DynamoDB.

5. 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.

6. 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 \(p. 701\)](#)

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:

1. 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 \(p. 702\)](#)

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/elasticmapreduce/>.
 - 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 \(p. 696\)](#), 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 \(p. 703\)](#)
- [Data Type Mappings \(p. 704\)](#)

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

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 \(p. 696\)](#)). 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 (p. 700)).

Topics

- [Using Aggregate Functions \(p. 706\)](#)
- [Using the GROUP BY and HAVING Clauses \(p. 706\)](#)
- [Joining Two DynamoDB tables \(p. 706\)](#)
- [Joining Tables from Different Sources \(p. 707\)](#)

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",
"YNAMODB.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 \(p. 696\)](#), 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 \(p. 708\)](#)
- [Copying Data Between DynamoDB and Amazon S3 \(p. 709\)](#)
- [Copying Data Between DynamoDB and HDFS \(p. 713\)](#)
- [Using Data Compression \(p. 717\)](#)
- [Reading Non-Printable UTF-8 Character Data \(p. 718\)](#)

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.

The examples in this section are written with the assumption you followed the steps in [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 696\)](#) and have an external table that is mastered in DynamoDB (*ddb_features*).

Example From Native Hive Table to DynamoDB

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 DynamoDB to Native Hive Table

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 \(p. 696\)](#), 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/elasticmapreduce/>.
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 that is mastered in DynamoDB (*ddb_features*).

Topics

- [Copying Data Using the Hive Default Format \(p. 709\)](#)
- [Copying Data with a User-Specified Format \(p. 710\)](#)
- [Copying Data Without a Column Mapping \(p. 711\)](#)
- [Viewing the Data in Amazon S3 \(p. 712\)](#)

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.

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 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"}^BClass^C{"s":"Summit"}^BElevation^C{"n":6135}^BLatitude^C{"n":32.3564729}
Name^C{"s":"Jones
Run"}^BState^C{"s":"PA"}^BClass^C{"s":Stream"}^BElevation^C{"n":1260}^BLatitude^C{"n":41.2120086}
Name^C{"s":"Sentinel
Dome"}^BState^C{"s":CA"}^BClass^C{"s":Summit"}^BElevation^C{"n":8133}^BLatitude^C{"n":37.7229821}
Name^C{"s":"Neversweet
Gulch"}^BState^C{"s":CA"}^BClass^C{"s":Valley"}^BElevation^C{"n":2900}^BLatitude^C{"n":41.6565269}
Name^C{"s":"Chacaloochee
Bay"}^BState^C{"s":AL"}^BClass^C{"s":Bay"}^BElevation^C{"n":0}^BLatitude^C{"n":30.6979676}^BId^C
```

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 master 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 master 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 master 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`

The examples in this section are written with the assumption you followed the steps in [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 696\)](#) and you have an external table that is mastered in DynamoDB (`ddb_features`).

Topics

- [Copying Data Using the Hive Default Format \(p. 713\)](#)
- [Copying Data with a User-Specified Format \(p. 714\)](#)
- [Copying Data Without a Column Mapping \(p. 715\)](#)
- [Accessing the Data in HDFS \(p. 716\)](#)

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":32.3564729}
Name^C{"s":"Jones
Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":1260}^BLatitude^C{"n":41.2120086}
Name^C{"s":"Sentinel
Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":8133}^BLatitude^C{"n":37.7229821}
Name^C{"s":"NeverSweet
Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":2900}^BLatitude^C{"n":41.6565269}
Name^C{"s":"Chacaloochee
Bay"}^BState^C{"s":"AL"}^BClass^C{"s":"Bay"}^BElevation^C{"n":0}^BLatitude^C{"n":30.6979676}^BId^C
```

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 master 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 master 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 master 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 master 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 \(p. 719\)](#)
- [Adjusting the Mappers \(p. 720\)](#)

- [Additional Topics \(p. 721\)](#)

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 bytes in the data you want to load, and then divide it by the size of one DynamoDB write capacity unit (1 KB). 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 \(p. 720\)](#).

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.

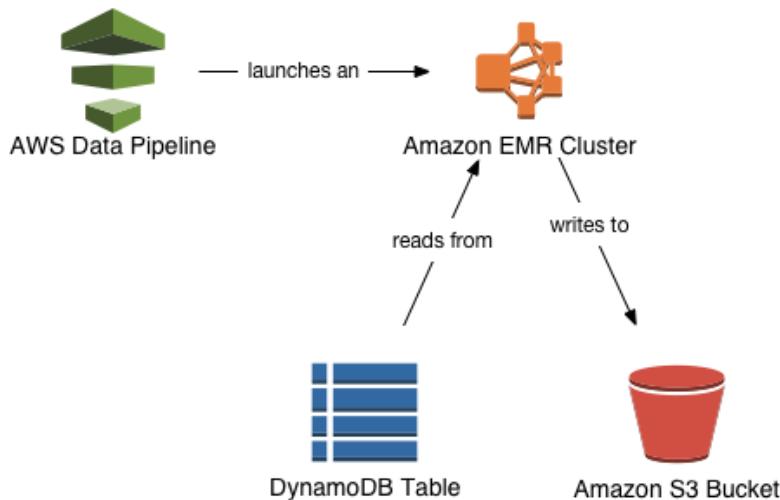
Exporting and Importing DynamoDB Data Using AWS Data Pipeline

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.

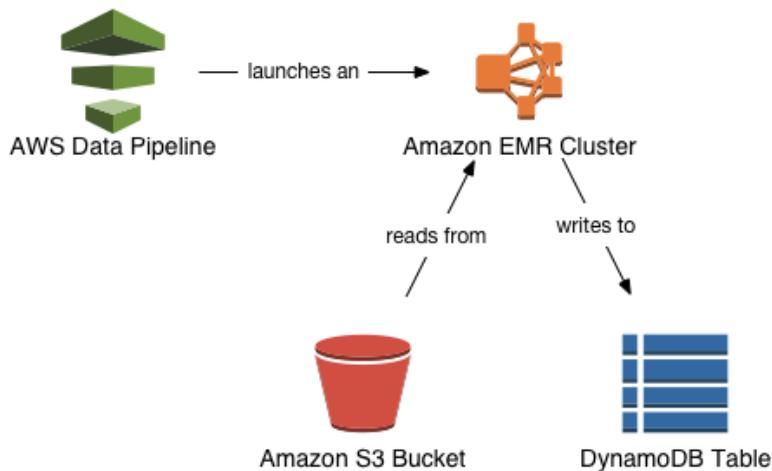
The ability to export and import data is useful in many scenarios. For example, suppose you want to maintain a baseline set of data, for testing purposes. You could put the baseline data into a DynamoDB table and export it to Amazon S3. Then, after you run an application that modifies the test data, 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.

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 performs reads and writes between DynamoDB to Amazon S3. The cluster configuration is one `m3.xlarge` instance master 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 IAM users or groups . These policies let you specify which users are allowed to import and export your DynamoDB data.

Important

The IAM user that performs the exports and imports must have an *active* AWS Access Key Id and Secret Key. For more information, see [Administering Access Keys for IAM Users](#) in the *IAM User Guide*.

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 have never used AWS Data Pipeline before, you will need to create *DataPipelineDefaultRole* and *DataPipelineDefaultResourceRole* yourself. Once you have created these roles, you can use them any time you want 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 \(p. 727\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 728\)](#).

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 New Role** and do the following:
 - a. In the **Role Name** field, type `DataPipelineDefaultRole` and then click **Next Step**.
 - b. In the **Select Role Type** panel, in the list of **AWS Service Roles**, go to **AWS Data Pipeline** and click **Select**.
 - c. In the **Attach Policy** panel, click the box next to the **AWSDataPipelineRole** policy, and then click **Next Step**.
 - d. In the **Review** panel, click **Create Role**.
4. Click **Create New Role** and do the following:
 - a. In the **Role Name** field, type `DataPipelineDefaultResourceRole` and then click **Next Step**.

- b. In the **Select Role Type** panel, in the list of **AWS Service Roles**, go to **Amazon EC2 Role for Data Pipeline** and click **Select**.
- c. In the **Attach Policy** panel, click the box next to the **AmazonEC2RoleforDataPipelineRole** policy, and then click **Next Step**.
- d. In the **Review** panel, click **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 \(p. 727\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 728\)](#).

Granting IAM Users and Groups Permission to Perform Export and Import Tasks

If you want to allow other IAM users 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 Using an AWS Managed Policy

The following procedure describes how to attach the AWS managed policy `AmazonDynamoDBFullAccesswithDataPipeline` to an IAM user. This managed policy provides full access to AWS Data Pipeline and to DynamoDB resources.

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 **Attach Policy**.
4. In the **Attach Policy** panel, select `AmazonDynamoDBFullAccesswithDataPipeline` and click **Attach Policy**.

Note

You can use a similar procedure to attach this managed policy to a 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 will need to create a customized IAM policy document. You can use the AWS managed policy `AmazonDynamoDBFullAccesswithDataPipeline` 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 an IAM 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 `AmazonDynamoDBFullAccesswithDataPipeline` 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:*
```

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...
]
```

- When the policy settings are as you want them, click **Create Policy**.

After you have created the policy, you can attach it to an IAM user.

- From the IAM Console Dashboard, click **Users** and select the user you want to modify.
- In the **Permissions** tab, click **Attach Policy**.
- 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 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 \(p. 724\)](#).

- 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. 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 file will be in a folder with the same name as your table, and the file will be named using the following format: `YYYY-MM-DD_HH.MM`. The internal format of this file is described at [Verify Data Export File](#) 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 [Verify Data Export File](#) 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 EU (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.

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. 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 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.
- You might not have the correct permissions for performing an export or import. For more information, see [Prerequisites to Export and Import Data \(p. 724\)](#).
- You might have reached a resource limit 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 limits, see [AWS Service Limits](#) 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 DynamoDB 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

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](#)

Limits in DynamoDB

This section describes current limits within Amazon DynamoDB (or no limit, in some cases). Each limit listed below applies on a per-region basis unless otherwise specified.

Topics

- [Capacity Units and Provisioned Throughput \(p. 731\)](#)
- [Tables \(p. 733\)](#)
- [Secondary Indexes \(p. 733\)](#)
- [Partition Keys and Sort Keys \(p. 733\)](#)
- [Naming Rules \(p. 734\)](#)
- [Data Types \(p. 734\)](#)
- [Items \(p. 735\)](#)
- [Attributes \(p. 735\)](#)
- [Expression Parameters \(p. 736\)](#)
- [DynamoDB Streams \(p. 736\)](#)
- [DynamoDB Accelerator \(DAX\) \(p. 737\)](#)
- [API-Specific Limits \(p. 737\)](#)

Capacity Units and Provisioned Throughput

Capacity Unit Sizes

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.

Provisioned Throughput Default Limits

For any table or global secondary index, the minimum settings for provisioned throughput are 1 read capacity unit and 1 write capacity unit.

An AWS account places some default limits on the throughput you can provision. These are the limits unless you request a higher amount. To request a service limit increase see <https://aws.amazon.com/support>.

- US East (N. Virginia) Region:
 - Per table – 40,000 read capacity units and 40,000 write capacity units
 - Per account – 80,000 read capacity units and 80,000 write capacity units
- All Other Regions:
 - Per table – 10,000 read capacity units and 10,000 write capacity units
 - Per account – 20,000 read capacity units and 20,000 write capacity units

Note

All the account's available throughput can be applied to a single table or across multiple tables.

The provisioned throughput limit includes the sum of the capacity of the table together with the capacity of all of its global secondary indexes.

In the AWS Management Console, you can see what your current provisioned capacity is in a given region and make sure you are not too close to the limits. If you increased your default limits, you can use the [DescribeLimits](#) operation to see the current limit values.

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 cannot exceed your per-account limits when you add provisioned capacity, and DynamoDB will not permit you to increase provisioned capacity extremely 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 limits, see the preceding section [Provisioned Throughput Default Limits \(p. 731\)](#).

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 do not take effect until the `UpdateTable` operation is complete. A dial down is allowed up to four times any time per day. A day is defined according to the GMT time zone. Additionally, if there was no decrease in the past four hours, an additional dial down is allowed, effectively bringing maximum number of decreases in a day to nine times (4 decreases in the first 4 hours, and 1 decrease for each of the subsequent 4 hour windows in a day).

Important

Table and GSIs dial down limits are decoupled, so any GSI(s) for a particular table have their own dial down limits. However, if a single request decreases the throughput for a table and a GSI, it will be rejected if either exceeds the current limits. A request will not be partially processed.

Example

A table with a GSI, in the first 4 hours of a day, can be modified as follows:

- Dial down the table's `WriteCapacityUnits` or `ReadCapacityUnits` (or both) 4 times.
- Dial down the GSI's `WriteCapacityUnits` or `ReadCapacityUnits` (or both) 4 times.

At the end of that same day the table and the GSI's throughput can potentially be decreased a total of 9 times each.

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.

Tables Per Account

For any AWS account, there is an initial limit of 256 tables per region.

To request a service limit increase see <https://aws.amazon.com/support>.

Secondary Indexes

Secondary Indexes Per Table

You can define a maximum of 5 local secondary indexes and 5 global secondary indexes per table.

Projected Secondary Index Attributes Per Table

You can project a total of up to 20 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 20. 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 local secondary indexes. With a local secondary index, there is a limit on item collection sizes: For every distinct partition key value, the total sizes of all table and index items cannot exceed 10 GB. This might constrain the number of sort keys per partition key value. For more information, see [Item Collection Size Limit \(p. 493\)](#).

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. Allowed characters are:

- A-Z
 - a-z
 - 0-9
 - _ (underscore)
 - - (hyphen)
 - . (dot)

Attribute Names

In general, an attribute name must be at least 1 character long, but no greater than 64 KB long.

The exceptions are listed below. 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`, then 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 \(p. 185\)](#).

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 the local secondary index entry corresponding to that item, including its key values and projected attributes.

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

An attribute value cannot be an empty String or 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 three bytes.

The maximum length of any single expression attribute name or expression attribute value is 255 bytes. For example, `#name` is five bytes; `:val` is four 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 \(p. 780\)](#).)

However, if you use a reserved word in an expression parameter, you must also specify `ExpressionAttributeNames`. For more information, see [Expression Attribute Names \(p. 342\)](#).

DynamoDB Streams

Simultaneous Readers of a Shard in DynamoDB Streams

Do not allow more than two processes to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling.

Maximum Write Capacity for a Table With a Stream Enabled

The following write capacity limits apply for tables with DynamoDB Streams enabled:

- US East (N. Virginia) Region:

- Per table – 40,000 write capacity units
- Per account – 80,000 write capacity units
- All Other Regions:
 - Per table – 10,000 write capacity units
 - Per account – 20,000 write capacity units

If you require a write capacity increase for a table with DynamoDB Streams enabled, go to <https://aws.amazon.com/support> and open a new Service Limit Increase case. Specify "Table Write Capacity Units" as the limit you want to increase.

DynamoDB Accelerator (DAX)

AWS Region Availability

DAX is available in the following AWS regions:

- US East (N. Virginia)
- US West (N. California)
- US West (Oregon)
- EU (Ireland)
- Asia Pacific (Tokyo)

Nodes

A DAX cluster consists of exactly 1 primary node, and between 0 and 9 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`

In general, you can have up to 10 `CreateTable`, `UpdateTable`, and `DeleteTable` requests running simultaneously (in any combination). In other words, the total number of tables in the `CREATING`, `UPDATING` or `DELETING` state cannot exceed 10.

The only exception is when you are creating a table with one or more secondary indexes. You can have up to 5 such requests running at a time; however, if the table or index specifications are complex, DynamoDB might temporarily reduce the number of concurrent requests below 5.

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.

DescribeLimits

`DescribeLimits` should only be called periodically. You can expect throttling errors if you call it more than once in a minute.

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.

DynamoDB Appendix

Topics

- [Example Tables and Data \(p. 739\)](#)
- [Creating Example Tables and Uploading Data \(p. 749\)](#)
- [DynamoDB Example Application Using AWS SDK for Python \(Boto\): Tic-Tac-Toe \(p. 764\)](#)
- [Amazon DynamoDB Storage Backend for Titan \(p. 780\)](#)
- [Reserved Words in DynamoDB \(p. 780\)](#)
- [Legacy Conditional Parameters \(p. 789\)](#)
- [Current Low-Level API Version \(2012-08-10\) \(p. 806\)](#)
- [Previous Low-Level API Version \(2011-12-05\) \(p. 807\)](#)

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">• <code>Id</code> (Number)
<i>Forum</i>	Simple primary key: <ul style="list-style-type: none">• <code>Name</code> (String)
<i>Thread</i>	Composite primary key: <ul style="list-style-type: none">• <code>ForumName</code> (String)• <code>Subject</code> (String)
<i>Reply</i>	Composite primary key: <ul style="list-style-type: none">• <code>Id</code> (String)

Table Name	Primary Key
	<ul style="list-style-type: none"> • 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>	<p>Composite primary key:</p> <ul style="list-style-type: none"> • PostedBy (String) • Message (String)

For more information about these tables, see [Use Case 1: Product Catalog \(p. 281\)](#) and [Use Case 2: Forum Application \(p. 281\)](#).

Sample Data Files

Topics

- [ProductCatalog Sample Data \(p. 740\)](#)
- [Forum Sample Data \(p. 745\)](#)
- [Thread Sample Data \(p. 745\)](#)
- [Reply Sample Data \(p. 747\)](#)

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: Load Data into Tables \(p. 283\)](#) in [Creating Tables and Loading Sample Data \(p. 280\)](#).

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": "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": "Amazon DynamoDB Thread"  
                    }  
                }  
            }  
        }  
    ]  
}
```

```
        "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 \(p. 749\)](#)
- [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 756\)](#)

In [Creating Tables and Loading Sample Data \(p. 280\)](#), 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 Sample Data \(p. 280\)](#). For step-by-step instructions to run this code using Eclipse, see [Java Code Samples \(p. 285\)](#).

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
// Licensed under the Apache License, Version 2.0.
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", "333-3333333333")
                .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")))
            .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 Sample Data \(p. 280\)](#). For step-by-step instructions to run this code in Visual Studio, see [.NET Code Samples \(p. 287\)](#).

```
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";
        bicycle4["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 AWS SDK for Python (Boto): Tic-Tac-Toe

Topics

- [Step 1: Deploy and Test Locally \(p. 765\)](#)
- [Step 2: Examine the Data Model and Implementation Details \(p. 768\)](#)
- [Step 3: Deploy in Production Using the DynamoDB Service \(p. 774\)](#)
- [Step 4: Clean Up Resources \(p. 780\)](#)

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 \(p. 765\)](#)** – 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 \(p. 768\)](#)** – 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 \(p. 774\)](#)** – 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 \(p. 780\)](#)** – 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 Required Packages \(p. 765\)](#)
- [1.2: Test the Game Application \(p. 766\)](#)

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 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, go to [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, go to [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 type 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 \(Downloadable Version\) \(p. 41\)](#).

4. Download the Tic-Tac-Toe application:

- a. Install Git. For instructions, go to [git Downloads](#).

- b. Execute 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:

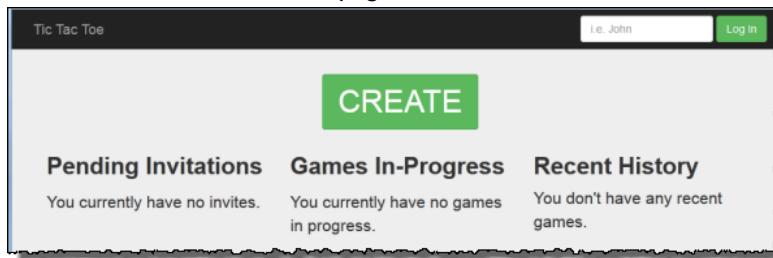
```
python.exe application.py --mode local --serverPort 5000 --port 8000
```

On Linux, you don't specify the .exe extension.

3. Open your web browser, and type the following:

```
http://localhost:5000/
```

The browser shows the home page:

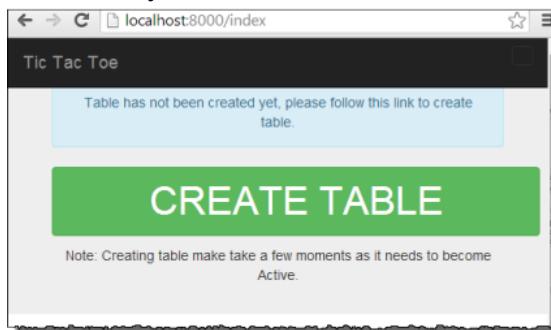


4. Type **user1** in the **Log in** box to log in as user1.

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. Type **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 type the following.

```
http://localhost:5000/
```

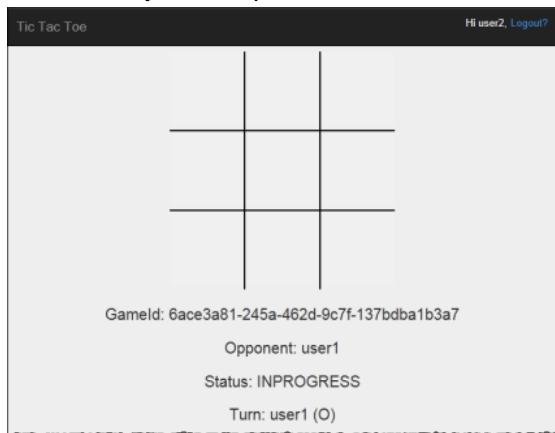
The browser passes information through cookies, so you should use incognito mode or private browsing so that your cookies don't carry over.

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 the move is valid, the square the user chose is available, and 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 five minutes, to check if the game state in DynamoDB has changed. If it has, the application updates the page with new information. After five 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 \(p. 768\)](#)
- [2.2: Application in Action \(Code Walkthrough\) \(p. 770\)](#)

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 \(p. 5\)](#).

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:

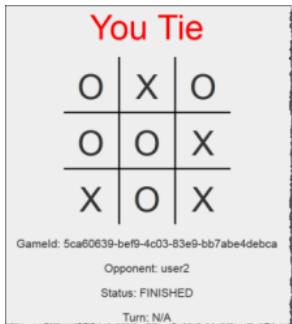
- HostId, the user who initiated the game.
- Opponent, the user who was invited to play.
- The user whose turn it is to play. The user who initiated the game plays first.
- The user who uses the O symbol on the board. The user who initiates the games uses the O symbol.

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:

The screenshot shows the Amazon DynamoDB Explore Table interface. The table is titled "Games". It has a single item with the following attributes:

Attribute	Type	Value
GameId (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 `o`, a `TopRight` attribute with the value `o`, and a `BottomRight` attribute with the value `x`. The attribute value is either `o` 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	Index Size (Bytes)*	Item Count*					
This table has no local 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*
hostStatusDate	HostId (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125
oppStatusDate	Opponent (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 `Opponent` 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 \(p. 446\)](#).

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.

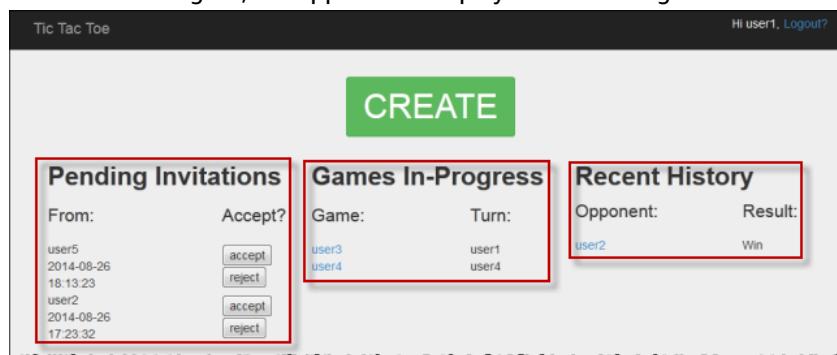
- **Game page** – This page shows the tic-tac-toe grid where users play.

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 return 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 remove 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.

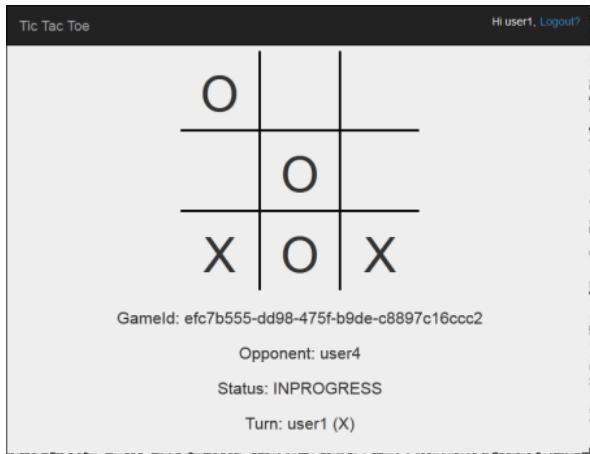
```
oppGamesInProgress = self.cm.getGamesTable().query(Opponent__eq=user,
                                                    StatusDate__beginswith=status,
                                                    index="OpponentId-StatusDate-index",
                                                    limit=10)
```

- 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 AWS 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 will update the game item accordingly.

Step 3: Deploy in Production Using the DynamoDB Service

Topics

- [3.1: Create an IAM Role for Amazon EC2 \(p. 775\)](#)
- [3.2: Create the Games Table in Amazon DynamoDB \(p. 776\)](#)
- [3.3: Bundle and Deploy Tic-Tac-Toe Application Code \(p. 776\)](#)
- [3.4: Set Up the AWS Elastic Beanstalk Environment \(p. 777\)](#)

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 Elastic Beanstalk, an easy-to-use service for deploying and scaling web applications and services. For more information, go to [Deploying a Flask Application to AWS Elastic Beanstalk](#).

Elastic Beanstalk will launch 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 will 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, go to [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 AWS region. For a list of available regions, go to [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 AWS IAM service. You will 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 on creating a bundle, go to [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 will launch an Amazon EC2 instance or instances and deploy 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 will allow the Amazon EC2 instance that is running your Tic-Tac-Toe application to assume the correct IAM 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": "  
                "arn:aws:dynamodb:  
                    <region>:  
                    <account>/Games  
            "  
        }  
    ]  
}
```

```
        "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, go to [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 will create the table for you. In this case, we will let the application create the Games table.

3.3: Bundle and Deploy 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 \(p. 765\)](#).

After you extract all files, note that you will have a `code` folder. To hand off this folder to Electric Beanstalk, you will bundle the contents of this folder as a `.zip` file. First, you need to add a configuration file to that folder. Your application will use 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=<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, go to [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 will 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, note that 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 will 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 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, we 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, go to the [Create a Bucket](#) and [Add an Object to a Bucket](#) topics in the *Amazon Simple Storage Service Getting Started 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 will launch one Amazon EC2 instance to deploy and run your Tic-Tac-Toe application.

1. Type 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
```

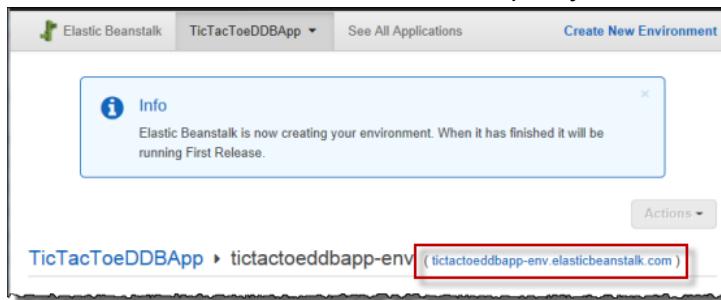
For more information about custom URLs, go to [Constructing a Launch Now URL](#) in the *AWS Elastic Beanstalk Developer Guide*. For the URL, note the following:

- You will need to 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**.

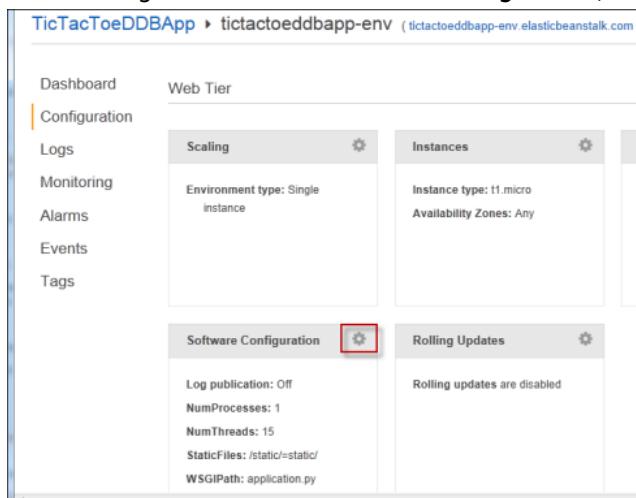
3. Note the URL for future reference. This URL opens your Tic-Tac-Toe application home page.



4. Configure the Tic-Tac-Toe application so it knows the location of the configuration file.

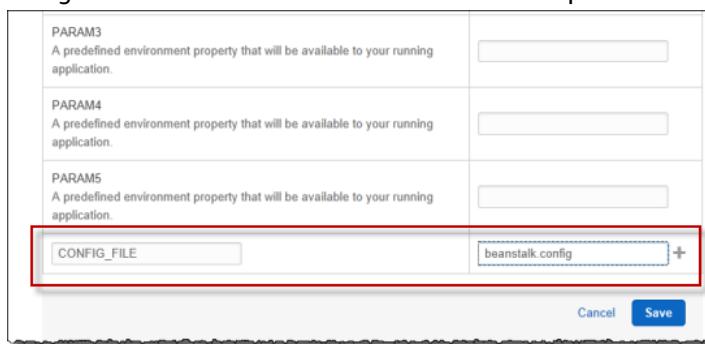
After Elastic Beanstalk creates the application, choose **Configuration**.

- a. Choose the gear box next to **Software Configuration**, as shown in the following screenshot.



- b. At the end of the **Environment Properties** section, type **CONFIG_FILE** and its value **beanstalk.config**, and then choose **Save**.

It might take a few minutes for this environment update to complete.

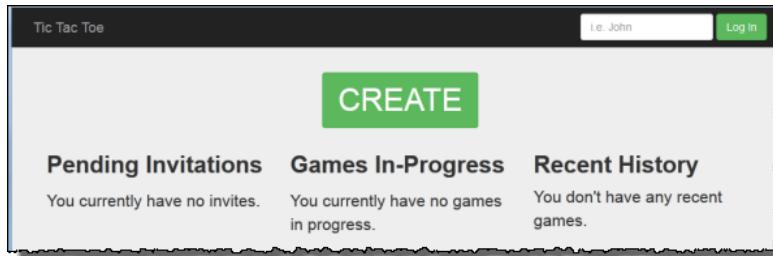


After the update completes, you can play the game.

5. In the browser, type the URL you copied in the previous step, as shown in the following example.

```
http://<app-name>.elasticbeanstalk.com
```

Doing this will open the application home page.



6. Log in as testuser1, and choose **CREATE** to start a new tic-tac-toe game.
7. Type **testuser2** in the **Choose an Opponent** box.



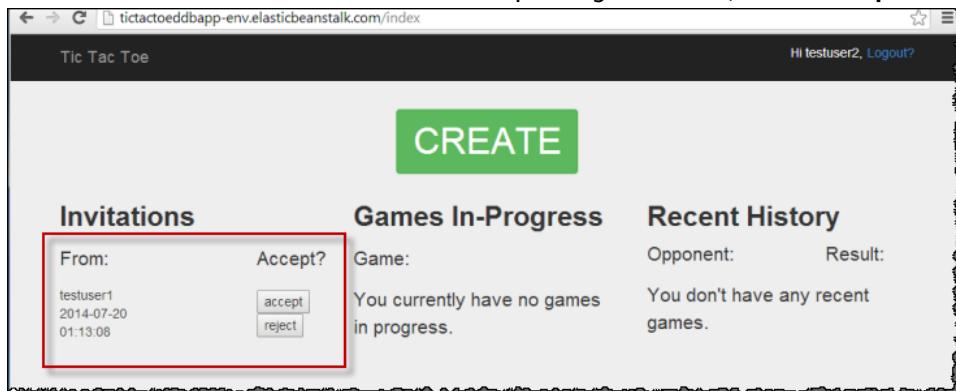
8. Open another browser window.

Make sure that you clear all cookies in your browser window so you won't be logged in as same user.

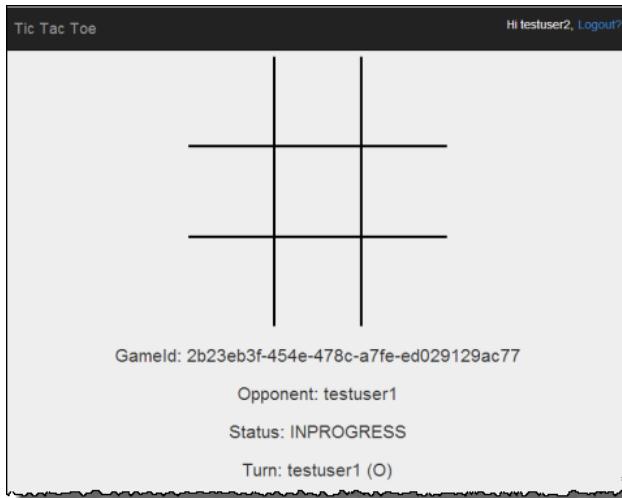
9. Type the same URL to open the application home page, as shown in the following example:

```
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 will appear.



Both testuser1 and testuser2 can play the game. For each move, the application will save 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 resources you created

1. Remove the Games table you created in DynamoDB.
2. Terminate the Elastic Beanstalk environment to free up the Amazon EC2 instances.
3. Delete the IAM role you created.
4. Remove the object you created in Amazon S3.

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.

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 \(p. 342\)](#).

```
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
FLATTERN
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.

With the introduction of expression parameters (see [Using Expressions in DynamoDB \(p. 338\)](#)), several older parameters have been deprecated. New applications should not use these legacy parameters, but should use expression parameters instead. (For more information, see [Using Expressions in DynamoDB \(p. 338\)](#).)

Note

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 APIs 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...	With These Legacy Parameters...	Use This Expression Parameter Instead
<code>BatchGetItem</code>	<code>AttributesToGet</code>	<code>ProjectionExpression</code>
<code>DeleteItem</code>	<code>Expected</code>	<code>ConditionExpression</code>
<code>GetItem</code>	<code>AttributesToGet</code>	<code>ProjectionExpression</code>
<code>PutItem</code>	<code>Expected</code>	<code>ConditionExpression</code>
<code>Query</code>	<code>AttributesToGet</code>	<code>ProjectionExpression</code>
	<code>KeyConditions</code>	<code>KeyConditionExpression</code>
	<code>QueryFilter</code>	<code>FilterExpression</code>
<code>Scan</code>	<code>AttributesToGet</code>	<code>ProjectionExpression</code>
	<code>ScanFilter</code>	<code>FilterExpression</code>
<code>UpdateItem</code>	<code>AttributeUpdates</code>	<code>UpdateExpression</code>
	<code>Expected</code>	<code>ConditionExpression</code>

The following sections provide more information about legacy conditional parameters.

Topics

- [AttributesToGet \(p. 790\)](#)
- [AttributeUpdates \(p. 791\)](#)
- [ConditionalOperator \(p. 793\)](#)
- [Expected \(p. 793\)](#)
- [KeyConditions \(p. 796\)](#)
- [QueryFilter \(p. 798\)](#)
- [ScanFilter \(p. 799\)](#)
- [Writing Conditions With Legacy Parameters \(p. 800\)](#)

AttributesToGet

`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

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"}
}'
```

But you could 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

In an `UpdateItem` operation, `AttributeUpdates` are 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

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"}
    }
}'
```

But you could 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"}
}'
```

ConditionalOperator

A logical operator 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 evaluate 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

`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 datatypes, 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 datatypes, 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 datatypes, 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 datatypes, 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 evaluate 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

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"} ]
        }
    }'
```

But you could 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

`KeyConditions` are the 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, less than, and so on.

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

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"} ]  
        }  
    }'
```

But you could 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

In a `Query` operation, `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 \(p. 793\)](#) 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 \(p. 185\)](#).

- `ComparisonOperator` - A comparator for evaluating attributes. For example, equals, greater than, less than, etc.

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

Suppose you wanted to query the `Music` table, but apply a condition to the items before they were returned to you. 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": "1.00"} ]
    }
}'
```

But you could 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

In a `Scan` operation, `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 \(p. 793\)](#) 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 \(p. 185\)](#).

- `ComparisonOperator` - A comparator for evaluating attributes. For example, equals, greater than, less than, etc.

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

Suppose you wanted to scan the *Music* table, but apply a condition to the items before they were returned to you. 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"
    }
}'
```

But you could 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

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 \(p. 338\)](#).

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 \(p. 793\)](#).

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 which 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 \(p. 793\)](#) 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.

Current Low-Level API Version (2012-08-10)

The current version of the low-level DynamoDB API is 2012-08-10. For complete documentation, see the [Amazon DynamoDB API Reference](#)

The following operations are supported:

- [BatchGetItem](#)
- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeLimits](#)
- [DescribeTable](#)
- [DescribeTimeToLive](#)
- [.GetItem](#)
- [ListTables](#)
- [ListTagsOfResource](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [TagResource](#)
- [UpdateItem](#)

- [UpdateTable](#)
- [UpdateTimeToLive](#)
- [UntagResource](#)

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 [Current Low-Level API Version \(2012-08-10\) \(p. 806\)](#).

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 \(p. 807\)](#)
- [BatchWriteItem \(p. 812\)](#)
- [CreateTable \(p. 817\)](#)
- [DeleteItem \(p. 822\)](#)
- [DeleteTable \(p. 826\)](#)
- [DescribeTables \(p. 829\)](#)
- [GetItem \(p. 832\)](#)
- [ListTables \(p. 835\)](#)
- [PutItem \(p. 837\)](#)
- [Query \(p. 842\)](#)
- [Scan \(p. 851\)](#)
- [UpdateItem \(p. 861\)](#)
- [UpdateTable \(p. 867\)](#)

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 [Item Sizes and Capacity Unit Consumption \(p. 296\)](#).

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
<code>RequestItems</code>	A container of the table name and corresponding items to get by primary key. While requesting items, each table name can be invoked only once per operation. Type: String Default: None	Yes
<code>Table</code>	The name of the table containing the items to get. The entry is simply a string	Yes

Name	Description	Required
	<p>specifying an existing table with no label.</p> <p>Type: String</p> <p>Default: None</p>	
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. 5).</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
Table:ConsistentRead	<p>If set to <code>true</code>, 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: 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"]}}], "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 attributes 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 Throughput Settings for Reads and Writes (p. 294) . Type: Number
UnprocessedKeys	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. Type: Array
UnprocessedKeys: Table: Keys	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. 5) .

Name	Description
	Type: Array of attribute name-value pairs.
UnprocessedKeys: Table: AttributesToGet	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.
	Type: Array of attribute names.
UnprocessedKeys: Table: ConsistentRead	If set to <code>true</code> , 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 in DynamoDB \(p. 327\)](#).

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":
    {"comp2":
        {"Keys":
            [{"HashKeyElement": {"S": "Julie"}}, {"HashKeyElement": {"S": "Mingus"}}],
        "AttributesToGet": ["user", "friends"]},
    "comp1":
        {"Keys":
            [{"HashKeyElement": {"S": "Casey"}, "RangeKeyElement": {"N": "1319509152"}},
             {"HashKeyElement": {"S": "Dave"}, "RangeKeyElement": {"N": "1319509155"}},
             {"HashKeyElement": {"S": "Riley"}, "RangeKeyElement": {"N": "1319509158"}],
        "AttributesToGet": ["user", "status"]}
    }
}
```

Sample Response

The following sample is the response.

```
HTTP/1.1 200 OK
x-amzn-RequestId: GTPQVRM4VJS792J1UFJTKUBVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 373
Date: Fri, 02 Sep 2011 23:07:39 GMT

{"Responses": {
    "comp2": {
        "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 in DynamoDB \(p. 290\)](#).

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 in DynamoDB \(p. 327\)](#).

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":{  
                        "ReplyDateTime":{  
                            "S":"2012-04-03T11:04:47.034Z"  
                        },  
                        "Id":{  
                            "S":"DynamoDB#DynamoDB Thread 5"  
                        }  
                    }  
                }  
            },  
            {  
                "DeleteRequest":{  
                    "Key":{  
                        "Id":{  
                            "S":"DynamoDB#DynamoDB Thread 5"  
                        }  
                    }  
                }  
            }  
        ]  
    }  
}
```

```

        "Key": {
            "HashKeyElement": {
                "S": "DynamoDB#DynamoDB Thread 4"
            },
            "RangeKeyElement": {
                "S": "oops - accidental row"
            }
        }
    ],
    "Thread": [
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "DynamoDB"
                    },
                    "Subject": {
                        "S": "DynamoDB Thread 5"
                    }
                }
            }
        }
    ]
}

```

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: G8M9ANLOE5QA26AEUHJKJE0ASBVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 829\)](#) 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>	Yes

Name	Description	Required
	Type: String	
KeySchema	<p>The primary key (simple or composite) structure for the table. A name-value pair for the <code>HashKeyElement</code> is required, and a name-value pair for the <code>RangeKeyElement</code> is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5).</p> <p>Primary key element names can be between 1 and 255 characters long with no character restrictions.</p> <p>Possible values for the <code>AttributeType</code> are "S" (string), "N" (numeric), or "B" (binary).</p> <p>Type: Map of <code>HashKeyElement</code>, or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key.</p>	Yes
ProvisionedThroughput	<p>New throughput for the specified table, consisting of values for <code>ReadCapacityUnits</code> and <code>WriteCapacityUnits</code>. For details, see Throughput Settings for Reads and Writes (p. 294).</p> <p>Note For current maximum/minimum values, see Limits in DynamoDB (p. 731).</p> <p>Type: Array</p>	Yes

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: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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 .

Name	Description
	Type: Number
KeySchema	<p>The primary key (simple or composite) structure for the table. A name-value pair for the <code>HashKeyElement</code> is required, and a name-value pair for the <code>RangeKeyElement</code> is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5).</p> <p>Type: Map of <code>HashKeyElement</code>, or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key.</p>
ProvisionedThroughput	<p>Throughput for the specified table, consisting of values for <code>ReadCapacityUnits</code> and <code>WriteCapacityUnits</code>. See Throughput Settings for Reads and Writes (p. 294).</p> <p>Type: Array</p>
ProvisionedThroughput :ReadCapacityUnits	<p>The minimum number of <code>ReadCapacityUnits</code> consumed per second before DynamoDB balances the load with other operations</p> <p>Type: Number</p>
ProvisionedThroughput :WriteCapacityUnits	<p>The minimum number of <code>ReadCapacityUnits</code> consumed per second before <code>WriteCapacityUnits</code> 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 (<code>CREATING</code>). Once the table is in the <code>ACTIVE</code> state, you can put data in it.</p> <p>Use the DescribeTables (p. 829) API to check the status of the table.</p> <p>Type: String</p>

Special Errors

Error	Description
<code>ResourceInUseException</code>	Attempt to recreate an already existing table.
<code>LimitExceededException</code>	The number of simultaneous table requests (cumulative number of tables in the <code>CREATING</code> ,

Error	Description
	<p>DELETING or UPDATING state) exceeds the maximum allowed.</p> <p>Note For current maximum/minimum values, see Limits in DynamoDB (p. 731).</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 in DynamoDB \(p. 290\)](#).

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: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 829\)](#)
- [DeleteTable \(p. 826\)](#)

DeleteItem

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 information about primary keys, see Primary Key (p. 5) . Type: Map of HashKeyElement to its value and RangeKeyElement to its value.	Yes
Expected	Designates an attribute for a conditional delete. The <code>Expected</code> parameter allows you to provide	No

Name	Description	Required
	<p>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>	
Expected: AttributeName	<p>The name of the attribute for the conditional put.</p> <p>Type: String</p>	No

Name	Description	Required
Expected:AttributeName: ExpectedAttributeValue	<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> <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 deleting the item:</p> <pre>"Expected" : {"Color":{"Exists":true}, {"Value":{"S":"Yellow"}}}</pre> <p>By default, if you use the <code>Expected</code> 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 <code>{"Exists":true}</code>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"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>	No

Name	Description	Required
ReturnValues	<p>Use this parameter if you want to get the attribute name-value pairs before they were deleted. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>. If <code>ALL_OLD</code> is specified, the content of the old item is returned. If this parameter is not provided or is <code>NONE</code>, nothing is returned.</p> <p>Type: String</p>	No

Responses

Syntax

```

HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 21:31:03 GMT

{
    "Attributes": [
        {"AttributeName3": {"S": ["AttributeValue3", "AttributeValue4", "AttributeValue5"]}},
        {"AttributeName2": {"S": "AttributeValue2"}},
        {"AttributeName1": {"N": "AttributeValue1"}}
    ],
    "ConsumedCapacityUnits": 1
}

```

Name	Description
Attributes	<p>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.</p> <p>Type: Array 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. Delete requests on non-existent items consume 1 write capacity unit. For more information see Throughput Settings for Reads and Writes (p. 294).</p> <p>Type: Number</p>

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: U9809LI6BBFJA5N2ROTB0P017JVV4KQNSO5AEMVJF66Q9ASUAAJG  
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 \(p. 837\)](#)

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 \(p. 829\)](#) 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: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
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. 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 (p. 5) . 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 Throughput Settings for Reads and Writes (p. 294) .
ProvisionedThroughput: ReadCapacityUnits	The minimum number of ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number
ProvisionedThroughput: WriteCapacityUnits	The minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number
TableName	The name of the deleted table. Type: String
TableStatus	The current state of the table (DELETING). Once the table is deleted, subsequent requests for the table return resource not found. Use the DescribeTables (p. 829) operation to check the status of the table. Type: String

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: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG  
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 \(p. 817\)](#)
- [DescribeTables \(p. 829\)](#)

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

Name	Description
	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 <code>HashKeyElement</code> is required, and a name-value pair for the <code>RangeKeyElement</code> 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. 5) .
ProvisionedThroughput	Throughput for the specified table, consisting of values for <code>LastIncreaseDateTime</code> (if applicable), <code>LastDecreaseDateTime</code> (if applicable), <code>ReadCapacityUnits</code> and <code>WriteCapacityUnits</code> . If the throughput for the table has never been increased or decreased, DynamoDB does not return values for those elements. See Throughput Settings for Reads and Writes (p. 294) .
	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 (<code>CREATING</code> , <code>ACTIVE</code> , <code>DELETING</code> or <code>UPDATING</code>). Once the table is in the <code>ACTIVE</code> 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 \(p. 817\)](#)
- [DeleteTable \(p. 826\)](#)
- [ListTables \(p. 835\)](#)

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 \(p. 15\)](#).

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 (p. 5) . Type: Map of HashKeyElement to its value and RangeKeyElement to its value.	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
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: 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 Throughput Settings for Reads and Writes (p. 294) . Type: Number

Special Errors

No errors specific to this operation.

Examples

For examples using the AWS SDK, see [Working with Items in DynamoDB \(p. 327\)](#).

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: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
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: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 829\)](#)
- [CreateTable \(p. 817\)](#)
- [DeleteTable \(p. 826\)](#)

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 in DynamoDB \(p. 327\)](#).

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

Name	Description	Required
Item	<p>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. For more information about primary keys, see Primary Key (p. 5).</p> <p>Type: Map of attribute names to attribute values.</p>	Yes
Expected	<p>Designates an attribute for a conditional put. The <code>Expected</code> 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:AttributeName: ExpectedAttributeValue	<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> <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 replacing the item:</p> <pre>"Expected" : {"Color":{"Exists":true, "Value":{"S":"Yellow"}}}</pre> <p>By default, if you use the <code>Expected</code> 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 <code>{"Exists":true}</code>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"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>	No

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>PutItem</code> request. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>. If <code>ALL_OLD</code> is specified, and <code>PutItem</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.</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 Throughput Settings for Reads and Writes (p. 294).</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 in DynamoDB \(p. 327\)](#).

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 \(p. 861\)](#)
- [DeleteItem \(p. 822\)](#)
- [GetItem \(p. 832\)](#)
- [BatchGetItem \(p. 807\)](#)

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	No

Name	Description	Required
	<p>be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p>	
Limit	<p>The maximum number of items to return (not necessarily the number of matching items). If DynamoDB 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>	No
ConsistentRead	<p>If set to <code>true</code>, then a consistent read is issued, otherwise eventually consistent is used.</p> <p>Type: Boolean</p>	No
Count	<p>If set to <code>true</code>, 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 <code>true</code> 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. 414).</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	<p>A container for the attribute values and comparison operators to use for the query. A query request does not require a <code>RangeKeyCondition</code>. If you provide only the <code>HashKeyValue</code>, DynamoDB returns all items with the specified hash key element value.</p> <p>Type: Map</p>	No
RangeKeyCondition: AttributeValueList	<p>The attribute values to evaluate for the query parameters. The <code>AttributeValueList</code> contains one attribute value, unless a <code>BETWEEN</code> comparison is specified. For the <code>BETWEEN</code> comparison, the <code>AttributeValueList</code> contains two attribute values.</p> <p>Type: A map of <code>AttributeValue</code> to a <code>ComparisonOperator</code>.</p>	No
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> <p>Note 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. For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.</p> <p>Type: String or Binary</p>	No

Name	Description	Required
	<p><code>EQ</code> : Equal.</p> <p>For <code>EQ</code>, <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> of type String, Number, or Binary (not a set). If an item contains an <code>AttributeValue</code> of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not equal <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p><code>LE</code> : Less than or equal.</p> <p>For <code>LE</code>, <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> of type String, Number, or Binary (not a set). If an item contains an <code>AttributeValue</code> of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p><code>LT</code> : Less than.</p> <p>For <code>LT</code>, <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> of type String, Number, or Binary (not a set). If an item contains an <code>AttributeValue</code> of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	

Name	Description	Required
	<p>GE : Greater than or equal.</p> <p>For GE, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p>GT : Greater than.</p> <p>For GT, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p>BEGINS_WITH : checks for a prefix.</p> <p>For BEGINS_WITH, AttributeValueList can contain only one AttributeValue 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><code>BETWEEN</code> : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For <code>BETWEEN</code>, <code>AttributeValueList</code> must contain two <code>AttributeValue</code> 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 <code>AttributeValue</code> of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not compare to <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
<code>ScanIndexForward</code>	<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 <code>true</code> (ascending).</p>	No

Name	Description	Required
ExclusiveStartKey	<p>Primary key of the item from which to continue an earlier query. An earlier query might provide this value as the <code>LastEvaluatedKey</code> if that query operation was interrupted before completing the query; either because of the result set size or the <code>Limit</code> parameter. The <code>LastEvaluatedKey</code> can be passed back in a new query request to continue the operation from that point.</p> <p>Type: <code>HashKeyElement</code>, or <code>HashKeyElement</code> and <code>RangeKeyElement</code> 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	<p>Item attributes meeting the query parameters.</p> <p>Type: Map of attribute names to and their data types and values.</p>
Count	Number of items in the response. For more information, see Counting the Items in the Results (p. 414) .

Name	Description
	Type: Number
LastEvaluatedKey	<p>Primary key of the item where the query operation stopped, inclusive of the previous result set. Use this value to start a new operation excluding this value in the new request.</p> <p>The <code>LastEvaluatedKey</code> is <code>null</code> 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 Throughput Settings for Reads and Writes (p. 294).</p> <p>Type: Number</p>

Special Errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see [Working with Queries \(p. 410\)](#).

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 \(p. 851\)](#)

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	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

Name	Description	Required
Limit	<p>The maximum number of items to evaluate (not necessarily the number of matching items). If DynamoDB 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 subsequent operation to continue the scan.</p> <p>Type: Number</p>	No
Count	<p>If set to <code>true</code>, 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 <code>true</code> 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. 429).</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:AttributeValueList	<p>The values and conditions to evaluate the scan results for the filter.</p> <p>Type: A map of <code>AttributeValue</code> to a condition.</p>	No
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> <p>Note 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. For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.</p> <p>Type: String or Binary</p>	No
	<p><code>EQ</code> : Equal.</p> <p>For <code>EQ</code>, <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> of type String, Number, or Binary (not a set). If an item contains an <code>AttributeValue</code> of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not equal <code>{"NS": ["6", "2", "1"]}</code>.</p>	

Name	Description	Required
	<p>NE : Not Equal.</p> <p>For NE, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not equal <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p>LE : Less than or equal.</p> <p>For LE, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p>LT : Less than.</p> <p>For LT, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	

Name	Description	Required
	<p>GE : Greater than or equal.</p> <p>For GE, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	<p>GT : Greater than.</p> <p>For GT, AttributeValueList can contain only one AttributeValue 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. For example, <code>{"S": "6"}</code> does not equal <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
	NOT_NULL : Attribute exists.	
	NULL : Attribute does not exist.	
	<p>CONTAINS : checks for a subsequence, or value in a set.</p> <p>For CONTAINS, AttributeValueList can contain only one AttributeValue 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).</p>	

Name	Description	Required
	<p><code>NOT_CONTAINS</code> : checks for absence of a subsequence, or absence of a value in a set.</p> <p>For <code>NOT_CONTAINS</code>, <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> 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>	
	<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, AttributeValueList 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 AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not compare to <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p>	
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": [
    {"AttributeName1":{"S":"AttributeValue1"}, "AttributeName2":{"S":"AttributeValue2"}, "AttributeName3":{"S":"AttributeValue3"}}
]
}

```

```

    },
    "AttributeName1": {"S": "AttributeValue4"},
    "AttributeName2": {"S": "AttributeValue5"},
    "AttributeName3": {"S": "AttributeValue6"},
    "AttributeName5": {"B": "dmFsdWU="}
  ],
  "LastEvaluatedKey":
    {
      "HashKeyElement": {"S": "AttributeName1"},
      "RangeKeyElement": {"N": "AttributeName2"}
    },
  "ConsumedCapacityUnits": 1,
  "ScannedCount": 2
}

```

Name	Description
Items	Container for the attributes meeting the operation 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 (p. 429) . Type: Number
ScannedCount	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. 429) . Type: Number
LastEvaluatedKey	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. The LastEvaluatedKey is null when the entire scan result set is complete (i.e. the operation processed the “last page”).
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 Throughput Settings for Reads and Writes (p. 294) . Type: Number

Special Errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see [Working with Scans \(p. 427\)](#).

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: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
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: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 842\)](#)
- [BatchGetItem \(p. 807\)](#)

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 in DynamoDB \(p. 327\)](#).

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": "AttributeValue3_New", "Action": "PUT"}},
     "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue3_Current"}}, "ReturnValues": "ReturnValuesConstant"}
}
```

Name	Description	Required
TableName	The name of the table containing the item to update. Type: String	Yes
Key	The primary key that defines the item. For more information about primary keys, see Primary Key (p. 5) . Type: Map of HashKeyElement to its value and RangeKeyElement to its value.	Yes
AttributeUpdates	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.	

Name	Description	Required
	Type: Map of attribute name, value, and an action for the attribute update.	
AttributeUpdates:Action	<p>Specifies how to perform the update. Possible values: <code>PUT</code> (default), <code>ADD</code> or <code>DELETE</code>. The semantics are explained in the <code>UpdateItem</code> description.</p> <p>Type: String</p> <p>Default: <code>PUT</code></p>	No
Expected	<p>Designates an attribute for a conditional update. The <code>Expected</code> 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:AttributeName: ExpectedAttributeValue	<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 <code>Expected</code> 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 <code>{"Exists":true}</code>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"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>	No

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	<p>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.</p> <p>Type: Map of attribute name-value pairs.</p>
ConsumedCapacityUnits	The number of write capacity units consumed by the operation. This value shows the number

Name	Description
	<p>applied toward your provisioned throughput. For more information see Throughput Settings for Reads and Writes (p. 294).</p> <p>Type: Number</p>

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 in DynamoDB \(p. 327\)](#).

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: 5IMHO7F01Q9P7Q6QMKMMI3R3QRVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 837\)](#)
- [DeleteItem \(p. 822\)](#)

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 [Throughput Settings for Reads and Writes \(p. 294\)](#).

The provisioned throughput values can be upgraded or downgraded based on the maximums and minimums listed in [Limits in DynamoDB \(p. 731\)](#).

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	The name of the table to update. Type: String	Yes
ProvisionedThroughput	New throughput for the specified table, consisting of values for <code>ReadCapacityUnits</code> and <code>WriteCapacityUnits</code> . See Throughput Settings for Reads and Writes (p. 294) . Type: Array	Yes

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: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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. Type: Number

Name	Description
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the <code>HashKeyElement</code> is required, and a name-value pair for the <code>RangeKeyElement</code> 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. 5) . Type: Map of <code>HashKeyElement</code> , or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key.
ProvisionedThroughput	Current throughput settings for the specified table, including values for <code>LastIncreaseDateTime</code> (if applicable), <code>LastDecreaseDateTime</code> (if applicable), Type: Array
TableName	The name of the updated table. Type: String
TableStatus	The current state of the table (<code>CREATING</code> , <code>ACTIVE</code> , <code>DELETING</code> or <code>UPDATING</code>), which should be <code>UPDATING</code> . Use the DescribeTables (p. 829) operation to check the status of the table. Type: String

Special Errors

Error	Description
<code>ResourceNotFoundException</code>	The specified table was not found.
<code>ResourceInUseException</code>	The table is not in the <code>ACTIVE</code> 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 \(p. 817\)](#)
- [DescribeTables \(p. 829\)](#)
- [DeleteTable \(p. 826\)](#)

Document History for DynamoDB

The following table describes important changes to the documentation.

- **API version:** 2012-08-10

Change	Description	Date Changed
Node.js support for DAX	Node.js developers can leverage Amazon DynamoDB Accelerator (DAX), using the DAX client for Node.js. For more information, see In-Memory Acceleration with DAX (p. 549) .	October 5, 2017
VPC Endpoints for DynamoDB	DynamoDB endpoints allow Amazon EC2 instances in your Amazon VPC to access DynamoDB, without exposure to the public Internet. Network traffic between your VPC and DynamoDB does not leave the Amazon network. For more information, see Amazon VPC Endpoints for DynamoDB (p. 686) .	August 16, 2017
Auto Scaling for DynamoDB	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. When the	June 14, 2017

Change	Description	Date Changed
	<p>workload decreases, DynamoDB auto scaling decreases the provisioned capacity. For more information, see Managing Throughput Capacity Automatically with DynamoDB Auto Scaling (p. 298).</p>	
Amazon DynamoDB Accelerator (DAX)	<p>Amazon 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 DAX (p. 549).</p>	April 19, 2017
DynamoDB now supports automatic item expiration with Time-To-Live (TTL)	<p>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 (p. 362).</p>	Feb 27, 2017
DynamoDB now supports Cost Allocation Tags	<p>You can now add tags to your Amazon DynamoDB tables for improved usage categorization and more granular cost reporting. For more information, see Tagging for DynamoDB (p. 314).</p>	Jan 19, 2017
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. For more information, see Limits in DynamoDB (p. 731) 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
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. 780).</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 Capturing Table Activity with DynamoDB Streams (p. 518) 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. For more information, see Cross-Region Replication (p. 540).</p> <p>The DynamoDB <code>Scan</code> operation uses eventually consistent reads, by default. You can use strongly consistent reads instead by setting the <code>ConsistentRead</code> parameter to true. For more information, see Read Consistency for Scan (p. 430) and <code>Scan</code> in the Amazon DynamoDB API Reference.</p>	July 16, 2015
AWS CloudTrail support for Amazon DynamoDB	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. For more information, see Logging DynamoDB Operations by Using AWS CloudTrail (p. 658) and the AWS CloudTrail User Guide .	May 28, 2015

Change	Description	Date Changed
Improved support for Query expressions	<p>This release adds a new <code>KeyConditionExpression</code> parameter to the <code>Query</code> API. A <code>Query</code> 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 <code>Query</code> 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 Working with Queries (p. 410).</p>	April 27, 2015
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. 346).</p>	April 27, 2015

Change	Description	Date Changed
Scan API for secondary indexes	<p>In DynamoDB, a <code>Scan</code> 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 <code>Scan</code> 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 (p. 427).</p>	February 10, 2015
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. 455).</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 (p. 12) and Using Expressions in DynamoDB (p. 338).</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 Limits in DynamoDB (p. 731).</p>	October 7, 2014
Larger item sizes	<p>The maximum item size in DynamoDB has increased from 64 KB to 400 KB. For more information, see Limits in DynamoDB (p. 731).</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 <i>ANDed</i> 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 in DynamoDB (p. 327).</p>	April 24, 2014
Query filter	<p>The DynamoDB <code>Query</code> API supports a new <code>QueryFilter</code> option. By default, a <code>Query</code> finds items that match a specific partition key value and an optional sort key condition. A <code>Query</code> filter applies conditional expressions to other, non-key attributes; if a <code>Query</code> filter is present, then items that do not match the filter conditions are discarded before the <code>Query</code> results are returned to the application. For more information, see Working with Queries (p. 410).</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. 722).</p>	March 6, 2014
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: Higher-Level Programming Interfaces for DynamoDB (p. 194).</p>	January 20, 2014

Change	Description	Date Changed
Global secondary indexes	<p>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 (p. 446) and Global Secondary Indexes (p. 448).</p>	December 12, 2013
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. 625).</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 Throughput Capacity for Reads and Writes (p. 15).</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 (p. 427).</p>	May 14, 2013
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. 484).</p>	April 18, 2013

Change	Description	Date Changed
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
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. 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 Authentication and Access Control for Amazon DynamoDB (p. 609).</p>	April 4, 2013
PHP code samples updated for AWS SDK for PHP version 2	<p>Version 2 of the AWS SDK for PHP is now available. The PHP code samples in the Amazon DynamoDB Developer Guide have been updated to use this new SDK. 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) 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
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 (p. 185). • 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 (p. 807). • 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 table. For more information, see Limits in DynamoDB (p. 731). 	November 2, 2012

Change	Description	Date Changed
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
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 attributes you can now store any binary data, for example compressed data, encrypted data, and images. For more information see Data Types (p. 12). 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 (p. 385) • Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API (p. 407) <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 Samples (p. 287).</p>	August 21, 2012

Change	Description	Date Changed
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 Limits in DynamoDB (p. 731) topic in the Amazon DynamoDB Developer Guide.	August 9, 2012
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 Sample Data (p. 280) and Using the Console (p. 48) sections have been updated for these features.	May 22, 2012
New endpoints	DynamoDB availability expands with new endpoints in the US West (N. California) region, US West (Oregon) region, and the Asia Pacific (Singapore) region. For the current list of supported endpoints, go to Regions and Endpoints .	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. 812).</p> <p>For information about working with items and using batch write feature using AWS SDKs, see Working with Items in DynamoDB (p. 327) and .NET Code Samples (p. 287).</p>	April 19, 2012
Documented more error codes	For more information, see Error Handling (p. 189) .	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
<code>ReturnedItemCount</code> 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 Monitoring DynamoDB (p. 642) .	February 24, 2012
Added examples for incrementing values	<p>DynamoDB supports incrementing and decrementing existing numeric values. Examples show adding to existing values in the "Updating an Item" sections at:</p> <p>Working with Items: Java (p. 368).</p> <p>Working with Items: .NET (p. 387).</p>	January 25, 2012
Initial product release	DynamoDB is introduced as a new service in Beta release.	January 18, 2012