
Amazon DynamoDB

Developer Guide

API Version 2012-08-10



Amazon DynamoDB: Developer Guide

Copyright © 2015 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

The following are trademarks of Amazon Web Services, Inc.: Amazon, Amazon Web Services Design, AWS, Amazon CloudFront, AWS CloudTrail, AWS CodeDeploy, Amazon Cognito, Amazon DevPay, DynamoDB, ElastiCache, Amazon EC2, Amazon Elastic Compute Cloud, Amazon Glacier, Amazon Kinesis, Kindle, Kindle Fire, AWS Marketplace Design, Mechanical Turk, Amazon Redshift, Amazon Route 53, Amazon S3, Amazon VPC, and Amazon WorkDocs. In addition, Amazon.com graphics, logos, page headers, button icons, scripts, and service names are trademarks, or trade dress of Amazon in the U.S. and/or other countries. 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.

AWS services or capabilities described in AWS Documentation may vary by region/location. Click Getting Started with Amazon AWS to see specific differences applicable to the China (Beijing) Region.

Table of Contents

What Is Amazon DynamoDB?	1
Service Highlights	2
Data Model	3
Data Model Concepts - Tables, Items, and Attributes	4
Primary Key	5
Secondary Indexes	6
Data Types	6
Supported Operations	9
Table Operations	9
Item Operations	9
Query and Scan	9
Data Read and Consistency Considerations	9
Conditional Updates and Concurrency Control	10
Provisioned Throughput	10
Read Capacity Units	11
Write Capacity Units	12
Accessing DynamoDB	12
Regions and Endpoints for DynamoDB	12
Getting Started	13
Step 1: Before You Begin	13
Step 1 of 6: Sign Up	13
Download AWS SDK	13
Step 2: Create Example Tables	14
Use case 1: Product Catalog	14
Use case 2: Forum Application	15
Creating Tables	15
Step 3: Load Sample Data	19
Load Data into Tables - Java	20
Load Data into Tables - .NET	27
Load Data into Tables - PHP	37
Verify Data Load	42
Step 4: Try a Query	43
Try a Query - Console	43
Try a Query - Java	44
Try a Query - .NET	46
Try a Query - PHP	52
Step 5: Delete Example Tables	52
Where Do I Go from Here?	53
Working with Tables	54
Specifying the Primary Key	54
Read and Write Requirements for Tables	55
Capacity Units Calculations for Various Operations	57
Item Size Calculations	57
Read Operation and Consistency	58
Listing and Describing Tables	59
Guidelines for Working with Tables	59
Design For Uniform Data Access Across Items In Your Tables	59
Understand Partition Behavior	61
Use Burst Capacity Sparingly	62
Distribute Write Activity During Data Upload	62
Understand Access Patterns for Time Series Data	63
Cache Popular Items	64
Consider Workload Uniformity When Adjusting Provisioned Throughput	64
Test Your Application At Scale	65
Working with Tables - Java Document API	66

Creating a Table	66
Updating a Table	67
Deleting a Table	68
Listing Tables	68
Example: Create, Update, Delete and List Tables - Java Document API	69
Working with Tables - .NET Low-Level API	71
Creating a Table	72
Updating a Table	73
Deleting a Table	74
Listing Tables	74
Example: Create, Update, Delete and List Tables - .NET Low-Level API	75
Working with Tables - PHP Low-Level API	78
Creating a Table	79
Updating a Table	80
Deleting a Table	81
Listing Tables	81
Example: Create, Update, Delete and List Tables - PHP Low-Level API	82
Working with Items	85
Overview	85
Reading an Item	86
Read Consistency	87
Writing an Item	87
Batch Operations	88
Atomic Counters	88
Conditional Writes	88
Reading and Writing Items Using Expressions	91
Case Study: A <i>ProductCatalog</i> Item	91
Accessing Item Attributes with Projection Expressions	92
Using Placeholders for Attribute Names and Values	94
Specifying Conditions with Condition Expressions	96
Modifying Items and Attributes with Update Expressions	100
Guidelines for Working with Items	106
Use One-to-Many Tables Instead Of Large Set Attributes	106
Use Multiple Tables to Support Varied Access Patterns	107
Compress Large Attribute Values	108
Store Large Attribute Values in Amazon S3	108
Break Up Large Attributes Across Multiple Items	109
Working with Items - Java Document API	110
Putting an Item	110
Getting an Item	113
Batch Write: Putting and Deleting Multiple Items	115
Batch Get: Getting Multiple Items	116
Updating an Item	118
Deleting an Item	120
Example: CRUD Operations - Java Document API	120
Example: Batch Operations - Java Document API	125
Example: Handling Binary Type Attributes - Java Document API	129
Working with Items - .NET Low-Level API	132
Putting an Item	133
Getting an Item	135
Updating an Item	136
Atomic Counter	138
Deleting an Item	138
Batch Write: Putting and Deleting Multiple Items	139
Batch Get: Getting Multiple Items	141
Example: CRUD Operations - .NET Low-Level API	144
Example: Batch Operations - .NET Low-Level API	151
Example: Handling Binary Type Attributes - .NET Low-Level API	162

Working with Items - PHP Low-Level API	167
Putting an Item	168
Getting an Item	170
Batch Write: Putting and Deleting Multiple Items	171
Batch Get: Getting Multiple Items	172
Updating an Item	174
Atomic Counter	176
Deleting an Item	176
Example: CRUD Operations - PHP Low-Level API	178
Example: Batch Operations-PHP SDK	180
Query and Scan	183
Narrowing the Results with Filter Expressions	184
Capacity Units Consumed by Query and Scan	185
Paginating the Results	185
Count and ScannedCount	186
Limit	186
Read Consistency for Query and Scan	186
Query and Scan Performance	186
Parallel Scan	187
Guidelines for Query and Scan	189
Avoid Sudden Bursts of Read Activity	189
Take Advantage of Parallel Scans	191
Querying Tables	192
Querying Tables—Java Document API	192
Querying Tables—.NET Low-Level API	198
Querying Tables—PHP Low-Level API	211
Scanning Tables	215
Scanning Tables—Java Document API	215
Scanning Tables—.NET Low-Level API	224
Scanning Tables—PHP Low-Level API	234
Improving Data Access with Secondary Indexes	241
Global Secondary Indexes	253
Attribute Projections	256
Querying a Global Secondary Index	258
Scanning a Global Secondary Index	258
Data Synchronization Between Tables and Global Secondary Indexes	258
Provisioned Throughput Considerations for Global Secondary Indexes	259
Storage Considerations for Global Secondary Indexes	260
Managing Global Secondary Indexes	260
Guidelines for Global Secondary Indexes	271
Global Secondary Indexes - Java Document API	273
Global Secondary Indexes - .NET Low-Level API	281
Global Secondary Indexes - PHP Low-Level API	292
Local Secondary Indexes	303
Attribute Projections	305
Creating a Local Secondary Index	307
Querying a Local Secondary Index	307
Scanning a Local Secondary Index	308
Item Writes and Local Secondary Indexes	309
Provisioned Throughput Considerations for Local Secondary Indexes	309
Storage Considerations for Local Secondary Indexes	311
Item Collections	311
Guidelines for Local Secondary Indexes	314
Local Secondary Indexes - Java Document API	316
Local Secondary Indexes - .NET Low-Level API	326
Local Secondary Indexes - PHP Low-Level API	340
Best Practices	350
Table Best Practices	350

Item Best Practices	350
Query and Scan Best Practices	351
Local Secondary Index Best Practices	351
Global Secondary Index Best Practices	351
DynamoDB Streams Preview	352
Accessing the DynamoDB Streams Preview	352
For More Information...	353
DynamoDB Console	354
Working with Items and Attributes	356
Adding an Item	356
Deleting an Item	359
Updating an Item	359
Copying an Item	360
Monitoring Tables	362
Setting Up CloudWatch Alarms	362
Exporting and Importing Data	363
Using the AWS SDKs	365
Using the AWS SDK for Java	365
Running Java Examples	367
Using the AWS SDK for .NET	368
Running .NET Examples	369
Using the AWS SDK for PHP	371
Running PHP Examples	371
Higher-Level Programming Interfaces for DynamoDB	373
Java: Object Persistence Model	373
Supported Data Types	376
Java Annotations for DynamoDB	377
The DynamoDBMapper Class	381
Optimistic Locking With Version Number	390
Mapping Arbitrary Data	392
Example: CRUD Operations	395
Example: Batch Write Operations	397
Example: Query and Scan	402
.NET: Document Model	410
Operations Not Supported by the Document Model	410
Working with Items - .NET Document Model	410
Getting an Item - Table.GetItem	414
Deleting an Item - Table.DeleteItem	415
Updating an Item - Table.UpdateItem	416
Batch Write - Putting and Deleting Multiple Items	417
Example: CRUD Operations - .NET Document Model	419
Example: Batch Operations-.NET Document Model API	425
Querying Tables - .NET Document Model	428
.NET: Object Persistence Model	441
DynamoDB Attributes	443
DynamoDBContext Class	445
Supported Data Types	450
Optimistic Locking Using Version Number	451
Mapping Arbitrary Data	453
Batch Operations	457
Example: CRUD Operations - .NET Object Persistence Model	461
Example: Batch Write Operation	464
Example: Query and Scan - .NET Object Persistence Model	470
Using the DynamoDB API	477
JSON Data Format	477
JSON Is Used as a Transport Protocol Only	478
Transferring Binary Data Type Values in JSON	479
Making HTTP Requests	479

HTTP Header Contents	479
HTTP Body Content	480
Handling HTTP Responses	480
Sample DynamoDB JSON Request and Response	481
Handling Errors	482
Error Types	482
API Error Codes	482
Catching Errors	486
Error Retries and Exponential Backoff	487
Batch Operations and Error Handling	488
Operations in DynamoDB	488
Example Application Using AWS SDK for Python (Boto)	490
Step 1: Deploy and Test Locally Using DynamoDB Local	491
1.1: Download and Install Required Packages	491
1.2: Test the Game Application	492
Step 2: Examine the Data Model and Implementation Details	494
2.1: Basic Data Model	494
2.2: Application in Action (Code Walkthrough)	496
Step 3: Deploy in Production	501
3.1: Create an IAM Role for Amazon EC2	502
3.2: Create the Games Table in Amazon DynamoDB	503
3.3: Bundle and Deploy Tic-Tac-Toe Application Code	503
3.4: Set Up the AWS Elastic Beanstalk Environment	504
Step 4: Clean Up Resources	507
Additional Tools and Resources For DynamoDB	508
DynamoDB Local	508
Downloading and Running DynamoDB Local	509
Using DynamoDB Local	510
Usage Notes	510
Differences Between DynamoDB Local and DynamoDB	511
JavaScript Shell for DynamoDB Local	511
Tutorial	512
Code Editor	514
AWS Command Line Interface for DynamoDB	516
Downloading and Configuring the AWS CLI	516
Using the AWS CLI with DynamoDB	517
Using the AWS CLI with DynamoDB Local	518
Integration with Other Services	519
Monitoring DynamoDB with CloudWatch	519
AWS Management Console	520
Command Line Interface (CLI)	520
API	520
DynamoDB Metrics	521
Dimensions for DynamoDB Metrics	526
Using IAM to Control Access to DynamoDB Resources	527
Amazon Resource Names (ARNs) for DynamoDB	528
DynamoDB Actions	528
Condition Types and Operators	529
IAM Policy Keys	529
Example Policies for API Actions and Resource Access	531
Fine-Grained Access Control for DynamoDB	536
Example Policies for Fine-Grained Access Control	538
Using Web Identity Federation	544
Exporting, Importing and Transforming Data Using AWS Data Pipeline	550
Using the AWS Management Console to Export and Import Data	550
Predefined Templates for AWS Data Pipeline and DynamoDB	564
Querying and Joining Tables Using Amazon Elastic MapReduce	564
Prerequisites for Integrating Amazon EMR	566

Step 1: Create a Key Pair	566
Step 2: Create a Cluster	567
Step 3: SSH into the Master Node	570
Step 4: Set Up a Hive Table to Run Hive Commands	572
Hive Command Examples for Exporting, Importing, and Querying Data	576
Optimizing Performance	583
Walkthrough: Using DynamoDB and Amazon Elastic MapReduce	586
Loading Data From DynamoDB Into Amazon Redshift	595
Limits	597
Document History	602
Appendix	609
Example Tables and Data	609
ProductCatalog Table - Sample Data	610
Forum Table - Sample Data	612
Thread Table - Sample Data	613
Reply Sample Data	613
Creating Example Tables and Uploading Data	614
Creating Example Tables and Uploading Data - Java	614
Creating Example Tables and Uploading Data - .NET	623
Creating Example Tables and Uploading Data - PHP	641
Reserved Words in DynamoDB	649
Legacy Conditional Parameters	659
Simple Conditions	659
Using Multiple Conditions	662
Other Conditional Operators	664
Current API Version (2012-08-10)	665
Previous API Version (2011-12-05)	666
BatchGetItem	666
BatchWriteItem	671
CreateTable	677
DeleteItem	681
DeleteTable	685
DescribeTables	688
GetItem	691
ListTables	693
PutItem	695
Query	700
Scan	706
UpdateItem	714
UpdateTable	719
AWS Glossary	724

What Is Amazon DynamoDB?

Topics

- [DynamoDB Service Highlights \(p. 2\)](#)
- [DynamoDB Data Model \(p. 3\)](#)
- [Supported Operations in DynamoDB \(p. 9\)](#)
- [Provisioned Throughput in Amazon DynamoDB \(p. 10\)](#)
- [Accessing DynamoDB \(p. 12\)](#)

Welcome to the Amazon DynamoDB Developer Guide. DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. If you are a developer, you can use DynamoDB to create a database table that can store and retrieve any amount of data, and serve any level of request traffic. DynamoDB automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent and fast performance. All data items are stored on solid state disks (SSDs) and are automatically replicated across multiple Availability Zones in a Region to provide built-in high availability and data durability.

If you are a database administrator, you can create a new DynamoDB database table, scale up or down your request capacity for the table without downtime or performance degradation, and gain visibility into resource utilization and performance metrics, all through the AWS Management Console. With DynamoDB, you can offload the administrative burdens of operating and scaling distributed databases to AWS, so you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

If you are a first-time user of DynamoDB, we recommend that you begin by reading the following sections:

- **What is DynamoDB**—The rest of this section describes the underlying data model, the operations it supports, and the class libraries that you can use to develop applications that use DynamoDB.
- **Getting Started with DynamoDB (p. 13)**—The Getting Started section walks you through the process of creating sample tables, uploading data, and performing some basic database operations.

Beyond getting started, you'll probably want to learn more about application development with DynamoDB. The following sections provide additional information.

- **Working with DynamoDB**—The following sections provide in-depth information about the key DynamoDB concepts:
 - [Working with Tables in DynamoDB \(p. 54\)](#)

- [Working with Items in DynamoDB \(p. 85\)](#)
- [Query and Scan Operations in DynamoDB \(p. 183\)](#)
- [Improving Data Access with Secondary Indexes in DynamoDB \(p. 241\)](#)
- **Using AWS SDKs**—AWS provides SDKs for you to develop applications using DynamoDB. These SDKs provide low-level API methods that correspond closely to the underlying DynamoDB operations. The .NET SDK also provides a document model to further simplify your development work. In addition, the AWS SDKs for Java and .NET also provide an object persistence model API that you can use to map your client-side classes to DynamoDB tables. This allows you to call object methods instead of making low-level API calls. For more information, including working samples, see [Using the AWS SDKs with DynamoDB \(p. 365\)](#).

In addition to .NET, Java, and PHP examples provided in this guide, the other AWS SDKs also support DynamoDB, including JavaScript, Python, Android, iOS, and Ruby. For links to the complete set of AWS SDKs, see [Start Developing with Amazon Web Services](#).

DynamoDB Service Highlights

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. With a few clicks in the AWS Management Console, customers can create a new DynamoDB database table, scale up or down their request capacity for the table without downtime or performance degradation, and gain visibility into resource utilization and performance metrics. DynamoDB enables customers to offload the administrative burdens of operating and scaling distributed databases to AWS, so they don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

DynamoDB is designed to address the core problems of database management, performance, scalability, and reliability. Developers can create a database table and grow its request traffic or storage without limit. DynamoDB automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent, fast performance. All data items are stored on Solid State Disks (SSDs) and are automatically replicated across multiple Availability Zones in a Region to provide built-in high availability and data durability.

DynamoDB enables customers to offload the administrative burden of operating and scaling a highly available distributed database cluster while only paying a low variable price for the resources they consume.

The following are some of the major DynamoDB features:

- **Scalable** — DynamoDB is designed for seamless throughput and storage scaling.
- **Provisioned Throughput** — When creating a table, simply specify how much throughput capacity you require. DynamoDB allocates dedicated resources to your table to meet your performance requirements, and automatically partitions data over a sufficient number of servers to meet your request capacity. If your application requirements change, simply update your table throughput capacity using the AWS Management Console or the DynamoDB APIs. You are still able to achieve your prior throughput levels while scaling is underway.
- **Automated Storage Scaling** — There is no limit to the amount of data you can store in a DynamoDB table, and the service automatically allocates more storage, as you store more data using the DynamoDB write APIs.
- **Fully Distributed, Shared Nothing Architecture** — DynamoDB scales horizontally and seamlessly scales a single table over hundreds of servers.
- **Fast, Predictable Performance** — Average service-side latencies for DynamoDB are typically single-digit milliseconds. The service runs on solid state disks, and is built to maintain consistent, fast latencies at any scale.

- **Easy Administration**— DynamoDB is a fully managed service – you simply create a database table and let the service handle the rest. You don't need to worry about hardware or software provisioning, setup and configuration, software patching, operating a reliable, distributed database cluster, or partitioning data over multiple instances as you scale.
- **Built-in Fault Tolerance**— DynamoDB has built-in fault tolerance, automatically and synchronously replicating your data across multiple Availability Zones in a Region for high availability and to help protect your data against individual machine, or even facility failures.
- **Flexible** — DynamoDB does not have a fixed schema. Instead, each data item may have a different number of attributes. Multiple data types (strings, numbers, binary, and sets) add richness to the data model.
- **Efficient Indexing** — Every item in an DynamoDB table is identified by a primary key, allowing you to access data items quickly and efficiently. You can also define secondary indexes on non-key attributes, and query your data using an alternate key.
- **Strong Consistency, Atomic Counters**— Unlike many non-relational databases, DynamoDB makes development easier by allowing you to use strong consistency on reads to ensure you are always reading the latest values. DynamoDB supports multiple native data types (numbers, strings, binaries, and multi-valued attributes). The service also natively supports atomic counters, allowing you to atomically increment or decrement numerical attributes with a single API call.
- **Cost Effective**— DynamoDB is designed to be extremely cost-efficient for workloads of any scale. You can get started with a free tier that allows more than 40 million database operations per month, and pay low hourly rates only for the resources you consume above that limit. With easy administration and efficient request pricing, DynamoDB can offer significantly lower total cost of ownership (TCO) for your workload compared to operating a relational or non-relational database on your own.
- **Secure**— DynamoDB is secure and uses proven cryptographic methods to authenticate users and prevent unauthorized data access. It also integrates with AWS Identity and Access Management for fine-grained access control for users within your organization.
- **Integrated Monitoring**— DynamoDB displays key operational metrics for your table in the AWS Management Console. The service also integrates with CloudWatch so you can see your request throughput and latency for each DynamoDB table, and easily track your resource consumption.
- **Amazon Redshift Integration**— You can load data from DynamoDB tables into Amazon Redshift, a fully managed data warehouse service. You can connect to Amazon Redshift with a SQL client or business intelligence tool using standard PostgreSQL JDBC or ODBC drivers, and perform complex SQL queries and business intelligence tasks on your data.
- **Amazon Elastic MapReduce Integration**— DynamoDB also integrates with Amazon Elastic MapReduce (Amazon EMR). Amazon EMR allows businesses to perform complex analytics of their large datasets using a hosted pay-as-you-go Hadoop framework on AWS. With the launch of DynamoDB, it is easy for customers to use Amazon EMR to analyze datasets stored in DynamoDB and archive the results in Amazon Simple Storage Service (Amazon S3), while keeping the original dataset in DynamoDB intact. Businesses can also use Amazon EMR to access data in multiple stores (i.e. DynamoDB and Amazon RDS), perform complex analysis over this combined dataset, and store the results of this work in Amazon S3.

DynamoDB Data Model

Topics

- [Data Model Concepts - Tables, Items, and Attributes \(p. 4\)](#)
- [Primary Key \(p. 5\)](#)
- [Secondary Indexes \(p. 6\)](#)
- [DynamoDB Data Types \(p. 6\)](#)

Data Model Concepts - Tables, Items, and Attributes

The DynamoDB data model concepts include tables, items and attributes.

In Amazon DynamoDB, a database is a collection of tables. A table is a collection of items and each item is a collection of attributes.

In a relational database, a table has a predefined schema such as the table name, primary key, list of its column names and their data types. All records stored in the table must have the same set of columns. DynamoDB is a NoSQL database: Except for the required primary key, a DynamoDB table is schema-less. Individual items in a DynamoDB table can have any number of attributes, although there is a limit of 400 KB on the item size. An item size is the sum of lengths of its attribute names and values (binary and UTF-8 lengths).

Each attribute in an item is a name-value pair. An attribute can be single-valued or multi-valued set. For example, a book item can have title and authors attributes. Each book has one title but can have many authors. The multi-valued attribute is a set; duplicate values are not allowed.

For example, consider storing a catalog of products in DynamoDB. You can create a table, *ProductCatalog*, with the *Id* attribute as its primary key.

```
ProductCatalog ( Id, ... )
```

You can store various kinds of product items in the table. The following table shows sample items.

Example items

```
{  
    Id = 101  
    ProductName = "Book 101 Title"  
    ISBN = "111-1111111111"  
    Authors = [ "Author 1", "Author 2" ]  
    Price = -2  
    Dimensions = "8.5 x 11.0 x 0.5"  
    PageCount = 500  
    InPublication = 1  
    ProductCategory = "Book"  
}
```

```
{  
    Id = 201  
    ProductName = "18-Bicycle 201"  
    Description = "201 description"  
    BicycleType = "Road"  
    Brand = "Brand-Company A"  
    Price = 100  
    Gender = "M"  
    Color = [ "Red", "Black" ]  
    ProductCategory = "Bike"  
}
```

Example items

```
{
    Id = 202
    ProductName = "21-Bicycle 202"
    Description = "202 description"
    BicycleType = "Road"
    Brand = "Brand-Company A"
    Price = 200
    Gender = "M"
    Color = [ "Green", "Black" ]
    ProductCategory = "Bike"
}
```

In the example, the *ProductCatalog* table has one book item and two bicycle items. Item 101 is a book with many attributes including the Authors multi-valued attribute. Item 201 and 202 are bikes, and these items have a Color multi-valued attribute. The *Id* is the only required attribute. Note that attribute values are shown using JSON-like syntax for illustration purposes.

DynamoDB does not allow null or empty string attribute values.

Primary Key

When you create a table, in addition to the table name, you must specify the primary key of the table. DynamoDB supports the following two types of primary keys:

- **Hash Type Primary Key**—In this case the primary key is made of one attribute, a hash attribute. DynamoDB builds an unordered hash index on this primary key attribute. In the preceding example, the hash attribute for the *ProductCatalog* table is *Id*.
- **Hash and Range Type Primary Key**—In this case, the primary key is made of two attributes. The first attribute is the hash attribute and the second one is the range attribute. DynamoDB builds an unordered hash index on the hash primary key attribute and a sorted range index on the range primary key attribute.

You must define the data type of each primary key attribute: String, Number, or Binary.

Different applications will have different requirements for tables and primary keys. For example, Amazon Web Services maintains several forums (see [Discussion Forums](#)). Each forum has many threads of discussion and each thread has many replies. You could potentially model this by creating the following three tables:

Table Name	Primary Key Type	Hash Attribute Name	Range Attribute Name
Forum (<u>Name</u> , ...)	Hash	Name	-
Thread (<u>ForumName</u> , <u>Subject</u> , ...)	Hash and Range	ForumName	Subject
Reply (<u>Id</u> , <u>ReplyDateTime</u> , ...)	Hash and Range	Id	ReplyDateTime

In this example, both the *Thread* and *Reply* tables have primary key of the hash and range type. For the *Thread* table, each forum name can have one or more subjects. In this case, *ForumName* is the hash attribute and *Subject* is the range attribute.

The *Reply* table has *Id* as the hash attribute and *ReplyDateTime* as the range attribute. The reply *Id* identifies the thread to which the reply belongs. When designing DynamoDB tables you have to take into account the fact that DynamoDB does not support cross-table joins. For example, the *Reply* table stores both the forum name and subject values in the *Id* attribute. If you have a thread reply item, you can then parse the *Id* attribute to find the forum name and subject and use the information to query the *Thread* or the *Forum* tables. This developer guide uses these tables to illustrate DynamoDB functionality. For information about these tables and sample data stored in these tables, see [Example Tables and Data \(p. 609\)](#).

Secondary Indexes

When you create a table with a hash-and-range key, you can optionally define one or more secondary indexes on that table. A secondary index lets you query the data in the table using an alternate key, in addition to queries against the primary key.

With the *Reply* table, you can query data items by *Id* (hash) or by *Id* and *ReplyDateTime* (hash and range). Now suppose you had an attribute in the table—*PostedBy*—with the user ID of the person who posted each reply. With a secondary index on *PostedBy*, you could query the data by *Id* (hash) and *PostedBy* (range). Such a query would let you retrieve all the replies posted by a particular user in a thread, with maximum efficiency and without having to access any other items.

DynamoDB supports two kinds of secondary indexes:

- Local secondary index — an index that has the same hash key as the table, but a different range key.
- Global secondary index — an index with a hash and range key that can be different from those on the table.

You can define up to 5 global secondary indexes and 5 local secondary indexes per table. For more information, see [Improving Data Access with Secondary Indexes in DynamoDB \(p. 241\)](#).

DynamoDB Data Types

Amazon DynamoDB supports the following data types:

- **Scalar types** – Number, String, Binary, Boolean, and Null.
- **Multi-valued types** – String Set, Number Set, and Binary Set.
- **Document types** – List and Map.

For example, in the *ProductCatalog* table, the *Id* is a Number type attribute and *Authors* is a String Set type attribute. Note that primary key attributes must be of type String, Number, or Binary.

The following are descriptions of each data type, along with examples. Note that the examples use JSON syntax.

Scalar Data Types

String

Strings are Unicode with UTF8 binary encoding. There is no upper limit to the string size when you assign it to an attribute except when the attribute is part of the primary key. For more information, see [Limits in](#)

DynamoDB (p. 597). Also, the length of the attribute is constrained by the 400 KB item size limit. Note that the length of the attribute must be greater than zero.

String value comparison is used when returning ordered results in the `Query` and `Scan` API actions. Comparison is 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.

Example

```
"Bicycle"
```

Number

Numbers are positive or negative exact-value decimals and integers. A number can have up to 38 digits of precision, and can be between 10^{-128} to 10^{+126} . The representation in DynamoDB is of variable length. Leading and trailing zeroes are trimmed.

All numbers are sent to DynamoDB as String types, which maximizes compatibility across languages and libraries. However DynamoDB handles them as the Number type for mathematical operations.

Note

If number precision is important, you should pass numbers to DynamoDB using strings that you convert from a number type. DynamoDB limits numbers to 38 digits. More than 38 digits will cause an error.

Example

```
"300"
```

Binary

Binary type attributes can store any binary data, for example compressed data, encrypted data, or images. DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.

There is no upper limit to the length of the binary value when you assign it to an attribute except when the attribute is part of the primary key. For more information, see [Limits in DynamoDB \(p. 597\)](#). Also, the length of the attribute is constrained by the 400 KB item size limit. Note that the length of the attribute must be greater than zero.

Client applications must encode binary values in base64 format. When DynamoDB receives the data from the client, it decodes the data into an unsigned byte array and uses that as the length of the attribute.

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

Example

```
"dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbnNvZGVk"
```

Boolean

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

Example

```
true
```

Null

Null represents an attribute with an unknown or undefined state.

Example

```
NULL
```

Multi-Valued Data Types

DynamoDB also supports types that represent number sets, string sets and binary sets. Multi-valued attributes such as an `Authors` attribute in a book item and a `Color` attribute of a product item are examples of String Set type attributes. Because each of these types is a set, the values in each must be unique. Attribute sets are not ordered; the order of the values returned in a set is not preserved. DynamoDB does not support empty sets.

Examples

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

Document Data Types

DynamoDB supports List and Map data types, which can be nested to represent complex data structures.

- A List type contains an ordered collection of values.
- A Map type contains an unordered collection of name-value pairs.

Lists and maps are ideal for storing JSON documents. The List data type is similar to a JSON array, and the Map data type is similar to a JSON object. There are no restrictions on the data types that can be stored in List or Map elements, and the elements do not have to be of the same type.

The following example shows a Map that contains a String, a Number, and a nested List (which itself contains another Map).

Example

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

]
}

Note

DynamoDB lets you access individual elements within lists and arrays, even if those elements are deeply nested. For more information, see [Document Paths \(p. 93\)](#).

Supported Operations in DynamoDB

To work with tables and items, Amazon DynamoDB offers the following set of operations:

Table Operations

DynamoDB provides operations to create, update and delete tables. After the table is created, you can use the `UpdateTable` operation to increase or decrease a table's provisioned throughput. DynamoDB also supports an operation to retrieve table information (the `DescribeTable` operation) including the current status of the table, the primary key, and when the table was created. The `ListTables` operation enables you to get a list of tables in your account in the region of the endpoint you are using to communicate with DynamoDB. For more information, see [Working with Tables in DynamoDB \(p. 54\)](#).

Item Operations

Item operations enable you to add, update and delete items from a table. The `UpdateItem` operation allows you to update existing attribute values, add new attributes, and delete existing attributes from an item. You can also perform conditional updates. For example, if you are updating a price value, you can set a condition so the update happens only if the current price is \$20.

DynamoDB provides an operation to retrieve a single item (`GetItem`) or multiple items (`BatchGetItem`). You can use the `BatchGetItem` operation to retrieve items from multiple tables. For more information, see [Working with Items in DynamoDB \(p. 85\)](#).

Query and Scan

The `Query` operation enables you to query a table using the hash attribute and an optional range filter. If the table has a secondary index, you can also `Query` the index using its key. You can query only tables whose primary key is of hash-and-range type; you can also query any secondary index on such tables. `Query` is the most efficient way to retrieve items from a table or a secondary index.

DynamoDB also supports a `Scan` operation, which you can use on a table or a secondary index. The `Scan` operation reads every item in the table or secondary index. For large tables and secondary indexes, a `Scan` can consume a large amount of resources; for this reason, we recommend that you design your applications so that you can use the `Query` operation mostly, and use `Scan` only where appropriate. For more information, see [Query and Scan Operations in DynamoDB \(p. 183\)](#).

You can use conditional expressions in both the `Query` and `Scan` operations to control which items are returned.

Data Read and Consistency Considerations

DynamoDB maintains multiple copies of each item to ensure durability. When you receive an "operation successful" response to your write request, DynamoDB ensures that the write is durable on multiple servers. However, it takes time for the update to propagate to all copies. The data is eventually consistent,

meaning that a read request immediately after a write operation might not show the latest change. However, DynamoDB offers you the option to request the most up-to-date version of the data. To support varied application requirements, DynamoDB supports both eventually consistent and strongly consistent read options.

Eventually Consistent Reads

When you read data (`GetItem`, `BatchGetItem`, `Query` or `Scan` operations), the response might not reflect the results of a recently completed write operation (`PutItem`, `UpdateItem` or `DeleteItem`). The response might include some stale data. Consistency across all copies of the data is usually reached within a second; so if you repeat your read request after a short time, the response returns the latest data. By default, the `Query` and `GetItem` operations perform eventually consistent reads, but you can optionally request strongly consistent reads. `BatchGetItem` operations are eventually consistent by default, but you can specify strongly consistent on a per-table basis. `Scan` operations are always eventually consistent. For more information about operations in DynamoDB, see [Using the DynamoDB API \(p. 477\)](#).

Strongly Consistent Reads

When you issue a strongly consistent read request, DynamoDB returns a response with the most up-to-date data that reflects updates by all prior related write operations to which DynamoDB returned a successful response. A strongly consistent read might be less available in the case of a network delay or outage. For the query or get item operations, you can request a strongly consistent read result by specifying optional parameters in your request.

Conditional Updates and Concurrency Control

In a multiuser environment, it is important to ensure data updates made by one client don't overwrite updates made by another client. This "lost update" problem is a classic database concurrency issue. Suppose two clients read the same item. Both clients get a copy of that item from DynamoDB. Client 1 then sends a request to update the item. Client 2 is not aware of any update. Later, Client 2 sends its own request to update the item, overwriting the update made by Client 1. Thus, the update made by Client 1 is lost.

DynamoDB supports a "conditional write" feature that lets you specify a condition when updating an item. DynamoDB writes the item only if the specified condition is met; otherwise it returns an error. In the "lost update" example, client 2 can add a condition to verify item values on the server-side are same as the item copy on the client-side. If the item on the server is updated, client 2 can choose to get an updated copy before applying its own updates.

DynamoDB also supports an "atomic counter" feature where you can send a request to add or subtract from an existing attribute value without interfering with another simultaneous write request. For example, a web application might want to maintain a counter per visitor to its site. In this case, the client only wants to increment a value regardless of what the previous value was. DynamoDB write operations support incrementing or decrementing existing attribute values.

For more information, see [Working with Items in DynamoDB \(p. 85\)](#).

Provisioned Throughput in Amazon DynamoDB

When you create or update a table, you specify how much provisioned throughput capacity you want to reserve for reads and writes. DynamoDB will reserve the necessary machine resources to meet your throughput needs while ensuring consistent, low-latency performance.

A unit of *read capacity* represents one strongly consistent read per second (or two eventually consistent reads per second) for items as large as 4 KB. A unit of *write capacity* represents one write per second for items as large as 1 KB.

Items larger than 4 KB will require more than one read operation. The total number of read operations necessary is the item size, rounded up to the next multiple of 4 KB, divided by 4 KB. For example, to calculate the number of read operations for an item of 10 KB, you would round up to the next multiple of 4 KB (12 KB) and then divide by 4 KB, for 3 read operations.

The following table explains how to calculate the provisioned throughput capacity that you need.

Capacity Units Required For	How to Calculate
Reads	Number of item reads per second \times 4 KB item size (If you use eventually consistent reads, you'll get twice as many reads per second.)
Writes	Number of item writes per second \times 1 KB item size

If your application's read or write requests exceed the provisioned throughput for a table, then those requests might be throttled. You can use the AWS Management Console to monitor your provisioned and actual throughput and to change your provisioned capacity in anticipation of traffic changes.

For more information about specifying the provisioned throughput requirements for a table, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

For tables with secondary indexes, DynamoDB consumes additional capacity units. For example, if you wanted to add a single 1 KB item to a table, and that item contained an indexed attribute, then you would need *two* write capacity units—one for writing to the table, and another for writing to the index. For more information, see:

- [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 309\)](#)
- [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 259\)](#)

Read Capacity Units

If your items are smaller than 4 KB in size, each read capacity unit will give you one strongly consistent read per second, or two eventually consistent reads per second. You cannot group multiple items in a single read operation, even if the items together are 4 KB or smaller. For example, if your items are 3 KB and you want to read 80 items per second from your table, then you need to provision $80 \text{ (reads per second)} \times 1 \text{ (3 KB / 4 KB = 0.75, rounded up to the next whole number)} = 80$ read capacity units for strong consistency. For eventual consistency, you need to provision only 40 read capacity units.

If your items are larger than 4 KB, you will need to round up the item size to the next 4 KB boundary. For example, if your items are 6 KB and you want to do 100 strongly consistent reads per second, you need to provision $100 \text{ (reads per second)} \times 2 \text{ (6 KB / 4 KB = 1.5, rounded up to the next whole number)} = 200$ read capacity units.

You can use the `Query` and `Scan` operations in DynamoDB to retrieve multiple consecutive items from a table or an index in a single request. With these operations, DynamoDB uses the cumulative size of the processed items to calculate provisioned throughput. For example, if a `Query` operation retrieves 100 items that are 1 KB each, the read capacity calculation is *not* $(100 \times 4 \text{ KB}) = 100$ read capacity units, as if those items were retrieved individually using `GetItem` or `BatchGetItem`. Instead, the total would be only 25 read capacity units ($(100 * 1024 \text{ bytes}) = 100 \text{ KB}$, which is then divided by 4 KB). For more information see [Item Size Calculations \(p. 57\)](#).

Write Capacity Units

If your items are smaller than 1 KB in size, then each write capacity unit will give you 1 write per second. You cannot group multiple items in a single write operation, even if the items together are 1 KB or smaller. For example, if your items are 512 bytes and you want to write 100 items per second to your table, then you would need to provision 100 write capacity units.

If your items are larger than 1 KB in size, you will need to round the item size up to the next 1 KB boundary. For example, if your items are 1.5 KB and you want to do 10 writes per second, then you would need to provision 10 (writes per second) \times 2 (1.5 KB rounded up to the next whole number) = 20 write capacity units.

Accessing DynamoDB

Amazon DynamoDB is a web service that uses HTTP and HTTPS as a transport and JavaScript Object Notation (JSON) as a message serialization format. Your application code can make requests directly to the DynamoDB web service API. Instead of making the requests to the DynamoDB API directly from your application, we recommend that you use the AWS Software Development Kits (SDKs). The easy-to-use libraries in the AWS SDKs make it unnecessary to call the DynamoDB API directly from your application. The libraries take care of request authentication, serialization, and connection management. For more information about using the AWS SDKs, see [Using the AWS SDKs with DynamoDB \(p. 365\)](#).

The AWS SDKs provide low-level APIs that closely match the underlying DynamoDB API. To further simplify application development, the SDKs also provide the following additional APIs:

- The Java and .NET SDKs provide APIs with higher levels of abstraction. These higher-level interfaces 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. This allows you to write object-centric code, rather than database-centric code.
- The .NET SDK provides a document model that wraps some of the low-level API functionality to further simplify your coding.

For more information, see [Using the AWS SDK for Java \(p. 365\)](#) and [Using the AWS SDK for .NET \(p. 368\)](#).

If you decide not to use the AWS SDKs, then your application will need to construct individual service requests. Each request must contain a valid JSON payload and correct HTTP headers, including a valid AWS signature. For more information on constructing your own service requests, see [Using the DynamoDB API \(p. 477\)](#).

DynamoDB also provides a management console that enables you to work with tables and items. You can create, update, and delete tables without writing any code. You can view all the existing items in a table or use a query to filter the items in the table. You can add new items or delete items. You can also use the management console to monitor the performance of your tables. Using CloudWatch metrics in the console, you can monitor table throughput and other performance metrics. For more information, go to [DynamoDB console](#).

Regions and Endpoints for DynamoDB

By default, the AWS SDKs and console for DynamoDB reference the US-West (Oregon) Region. As DynamoDB expands availability to new regions, new endpoints for these regions are also available to use in your own HTTP requests, the AWS SDKs, and the console. For a current list of supported regions and endpoints, see [Regions and Endpoints](#).

Getting Started with DynamoDB

Topics

- Step 1: Before You Begin (p. 13)
- Step 2: Create Example Tables (p. 14)
- Step 3: Load Data into Tables (p. 19)
- Step 4: Try a Query (p. 43)
- Step 5: Delete Example Tables (p. 52)
- Where Do I Go from Here? (p. 53)

Step 1: Before You Begin

Before you can start with this exercise, you must sign up for the service and download one of the AWS SDKs. The following sections provide step-by-step instructions.

Sign up for the Service

To use DynamoDB, you need an AWS account. If you don't already have one, you'll be 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 DynamoDB

1. Open <http://www.amazonaws.cn/>, and then click **Sign Up**.
2. Follow the on-screen instructions.

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

Download AWS SDK

To try the getting started examples, you must determine the programming language that you want to use and download the appropriate AWS Software Development Kit (SDK) for your development platform.

Note

This developer guide provides code examples in Java, C#, and PHP.

If you want to use a different programming language with DynamoDB, go to <http://www.amazonaws.cn/code> and download the appropriate SDK. AWS provides SDK support for Python, Ruby, JavaScript, and more.

Downloading the AWS SDK for Java

To test the Java examples in this developer guide, you need the AWS SDK for Java.

You have the following download options:

- If you are using Eclipse, you can download and install the AWS Toolkit for Eclipse using the update site <http://www.amazonaws.cn/eclipse/>. For more information, go to [AWS Toolkit for Eclipse](#).
- If you are using any other IDE to create your application, download the [AWS SDK for Java](#).

Downloading the AWS SDK for .NET

To test the C# examples in this developer guide, you need the AWS SDK for .NET.

You have the following download options:

- If you are using Visual Studio, you can install both the AWS SDK for .NET and the Toolkit for Visual Studio. The toolkit provides AWS Explorer for Visual Studio and project templates that you can use for development. Go to <http://www.amazonaws.cn/sdkfornet> and click **Download AWS .NET SDK**. By default, the installation script installs both the AWS SDK and the Toolkit for Visual Studio. To learn more about the toolkit, go to [AWS Toolkit for Visual Studio User Guide](#).
- If you are using any other IDE to create your application, you can use the same link provided in the preceding step and install only the AWS SDK for .NET.

Downloading the AWS SDK for PHP

To test the PHP examples in this developer guide, you need the AWS SDK for PHP. Go to <http://www.amazonaws.cn/sdkforphp> and follow the instructions on that page to download the AWS SDK for PHP.

Step 2: Create Example Tables

The getting started example covers the two following simple use cases.

Use case 1: Product Catalog

Suppose you want to store product information in DynamoDB. Each product you store has its own set of properties, and accordingly, you need to store different information about each of these products. DynamoDB is a NoSQL database: Except for a required common primary key, individual items in a table can have any number of attributes. This enables you to save all the product data in the same table. So you will create a ProductCatalog table that uses Id as the primary key and stores information for products such as books and bicycles in the table. Id is a numeric attribute and hash type primary key. After creating the table, in the next step you will write code to retrieve items from this table. Note that while you can retrieve an item, you cannot query the table. To query the table, the primary key must be of the hash and range type.

Table Name	Primary Key Type	Hash Attribute Name and Type	Range Attribute Name and Type	Provisioned Throughput
ProductCatalog (<u>Id</u> , ...)	Hash	Attribute Name: Id Type: Number	-	Read capacity units: 10 Write capacity units: 5

Use case 2: Forum Application

Amazon Web Services maintains several forums (see [Discussion Forums](#)) for customers to engage with the developer community, ask questions, or reply to other customer queries. AWS maintains one or more forums for each of its services. Customers go to a forum and start a thread by posting a message. Over time, each thread receives one or more replies.

In this exercise, we model this application by creating the three following tables. Note that the Thread and Reply tables have hash and range type primary keys and therefore you can query these tables.

Table Name	Primary Key Type	Hash Attribute Name and Type	Range Attribute Name and Type	Provisioned Throughput
Forum (<u>Name</u> , ...)	Hash	Attribute Name: Name Type: String	-	Read capacity units: 10 Write capacity units: 5
Thread (<u>ForumName</u> , <u>Subject</u> , ...)	Hash and Range	Attribute Name: ForumName Type: String	Attribute Name: Subject Type: String	Read capacity units: 10 Write capacity units: 5
Reply (<u>Id</u> , <u>ReplyDateTime</u> , ...)	Hash and Range	Attribute Name: Id Type: String	Attribute Name: ReplyDateTime Type: String	Read capacity units: 10 Write capacity units: 5

The Reply table has the following local secondary index:

Index Name	Attribute to Index	Projected Attributes
PostedBy-index	PostedBy	Table and Index Keys

In the next step, you will write a simple query to retrieve data from these tables.

Creating Tables

For this exercise, you will use the DynamoDB console to create the ProductCatalog, Forum, Thread and Reply tables.

Note

In these getting started steps, you use these tables to explore some of the basic DynamoDB operations. However, these tables are also used in other examples throughout this reference.

If you delete these tables and later want to recreate them, you can repeat this getting started step, or programmatically recreate them and upload sample data. For more information about creating the tables and loading the data programmatically, see [Creating Example Tables and Uploading Data \(p. 614\)](#).

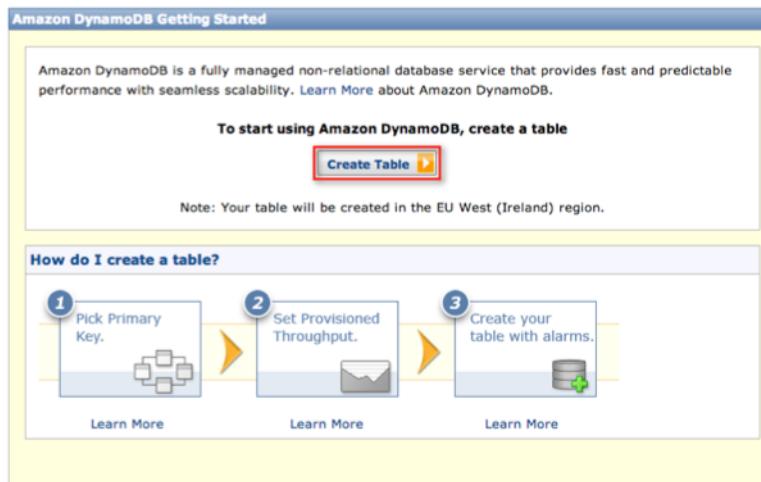
To Create the Sample Tables

Use the following procedure to create a table. You will need to perform this procedure once for each of the tables described in [Use case 1: Product Catalog \(p. 14\)](#) and [Use case 2: Forum Application \(p. 15\)](#):

- ProductCatalog
- Forum
- Thread
- Reply

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.

For first time users, the following wizard opens.



If you already have tables in DynamoDB, you'll see the console with your list of tables.

2. Click **Create Table**.

The **Create Table** wizard opens.

3. Set the table name and its primary key

- a. Specify the table name in the **Table Name** field.

See the preceding table for the list of tables that you are creating.

- b. Select the primary key type.

See the preceding table for the primary key type of the table that you are creating.

- c. If the table's primary key is of Hash and Range type, specify the hash attribute name and type for both the hash and range attributes.

The screenshot shows the 'Create Table' wizard with the 'PRIMARY KEY' tab selected. The 'Table Name' field is empty. Under 'Primary Key', 'DynamoDB is a schema-less database. You only need to tell us your primary key attribute(s)'. The 'Primary Key Type' is set to 'Hash and Range' (radio button selected). Below it, 'Hash Attribute Name' is set to 'String' (radio button selected) and 'Range Attribute Name' is also set to 'String' (radio button selected). A note at the bottom says: 'Choose a hash attribute that ensures that your workload is evenly distributed across hash keys. For example, "Customer ID" is a good hash key, while "Game ID" would be a bad choice if most of your traffic relates to a few popular games.' A link 'Learn more about choosing your primary key' is present.

- d. If the table's primary key is of Hash type, specify the hash attribute name and select the attribute type.

The screenshot shows the 'Create Table' wizard with the 'PRIMARY KEY' tab selected. The 'Table Name' field contains 'enter table name...'. Under 'Primary Key', 'DynamoDB is a schema-less database. You only need to tell us your primary key attribute(s)'. The 'Primary Key Type' is set to 'Hash' (radio button selected). Below it, 'Hash Attribute Name' is set to 'String' (radio button selected). A note at the bottom says: 'Choose a hash attribute that ensures that your workload is evenly distributed across hash keys. For example, "Customer ID" is a good hash key, while "Game ID" would be a bad choice if most of your traffic relates to a few popular games.' A link 'Learn more about choosing your primary key' is present.

- e. Click **Continue**.

4. If you are creating the *Reply* table, you will need to define a local secondary index:

A local secondary index allows you to perform queries against an attribute that is not part of the primary key. For this exercise, you will create a local secondary index on the *PostedBy* attribute of the *Reply* table.

- a. In the **Index Type** field, select *Local Secondary Index*.
- b. In the **Index Range Key** field, enter *PostedBy*.
- c. In the **Index Name** field, accept the default name of *PostedBy-index*.
- d. In the **Projected Attributes** field, select **Table and Index Keys**.
- e. Click **Add Index To Table**.

f. Click **Continue**.

5. Specify the provisioned throughput

- a. In the **Create Table - Provisioned Throughput** step, leave the **Help me estimate Provisioned Throughput** checkbox unchecked.

It is important to configure the appropriate provisioned throughput based on your expected item size and your expected read and write request rates. There is cost associated with the configured provisioned throughput. For more information, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#). However, for the getting started exercise, you will set finite values.

- b. In the **Read Capacity Units** field, enter 10. In the **Write Capacity Units** field, enter 5 and click **Continue**.

These throughput values allow you up to ten 4 KB read operations and up to five 1 KB write operations per second. For more information, see [DynamoDB Data Model \(p. 3\)](#).

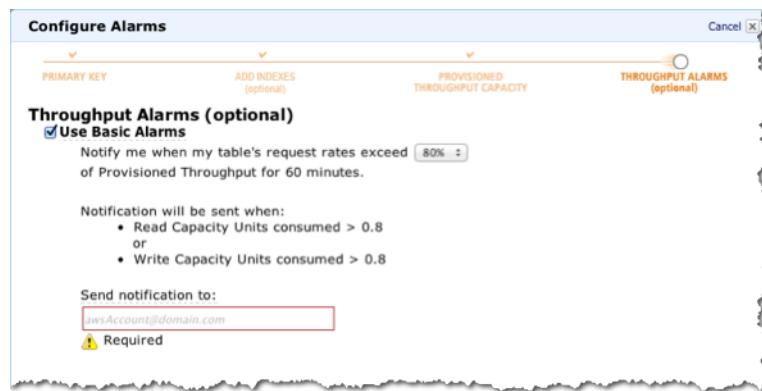
6. Configure CloudWatch Alarms

In the **Create Table - Throughput Alarms (optional)** wizard, select the **Use Basic Alarms** check box.

This automatically configures CloudWatch alarms to notify you when your consumption reaches 80% of the table's provisioned throughput. By default, the alarm is set to send an email to the AWS Account email address that you are using to create the table. You can edit the **Send notification to:** text box and specify additional email addresses that are separated by commas.

When you delete the table using the console, you can optionally delete the associated CloudWatch alarms.

For more information about CloudWatch alarms, see the [Amazon CloudWatch Developer Guide](#).

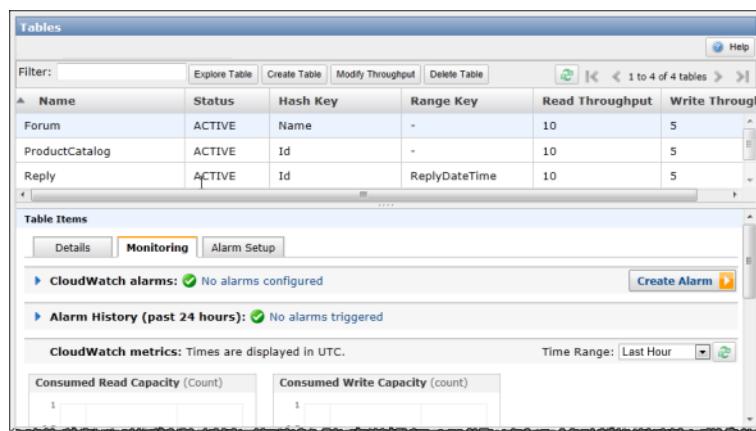


7. Click **Create Table**.

Note

Repeat this procedure to create the remaining tables described in [Use case 1: Product Catalog \(p. 14\)](#) and [Use case 2: Forum Application \(p. 15\)](#).

The console shows the list of tables. You must wait for the status of all the tables to become ACTIVE. The console also shows the **Details**, **Monitoring**, and **Alarm Setup** tabs that provide additional information about the selected table.



Step 3: Load Data into Tables

Topics

- [Load Data into Tables Using the AWS SDK for Java \(p. 20\)](#)
- [Load Data into Tables Using the AWS SDK for .NET \(p. 27\)](#)

- [Load Data into Tables Using the AWS SDK for PHP \(p. 37\)](#)
- [Verify Data Load \(p. 42\)](#)

In this step, you will upload sample data to the tables that you created. You can choose the application development platform that you want to use to explore DynamoDB.

After you do this, you can use the DynamoDB console to verify your data upload.

Load Data into Tables Using the AWS SDK for Java

In the preceding step, you created sample tables using the console. Now, you can upload the sample data to these tables. The following Java code example uses the AWS SDK for Java to upload the sample data. For step-by-step instructions on configuring your AWS access keys, setting the default endpoint and running the sample, see [Running Java Examples for DynamoDB \(p. 367\)](#).

Note

After you run this program, see [Verify Data Load \(p. 42\)](#) to view the tables and data in the DynamoDB console.

Example - Upload Sample Items Using the AWS SDK for Java

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.HashSet;
import java.util.TimeZone;

import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;

public class GettingStartedLoadData {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));
    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 {

            loadSampleProducts(productCatalogTableName);
            loadSampleForums(forumTableName);
            loadSampleThreads(threadTableName);
            loadSampleReplies(replyTableName);

        } catch (AmazonServiceException ase) {
            System.err.println("Data load script failed.");
        }
    }

    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)
.withString("Gender", "M")
// Men's
.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)
.withString("Gender", "M")
.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)
.withString("Gender", "W")
// Women's
.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)
.withString("Gender", "W")
.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)
.withString("Gender", "B")
// Boy's
.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", "primarykey", "rangekey")));
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());
}

}
```

Load Data into Tables Using the AWS SDK for .NET

In the preceding step, you created sample tables using the console. Now, you can upload sample data to these tables. The following C# code example uses the AWS SDK for .NET document model API to upload sample data. For step-by-step instructions on configuring your AWS access keys, setting the default endpoint and running the sample, see [Running .NET Examples for DynamoDB \(p. 369\)](#).

Note

After you run this program, see [Verify Data Load \(p. 42\)](#) to view the tables and data in the DynamoDB console.

Example - Upload Sample Items Using the AWS SDK for .NET Document Model API

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class GettingStartedLoadData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                // Load data (using the .NET document API)
                LoadSampleProducts();
                LoadSampleForums();
                LoadSampleThreads();
                LoadSampleReplies();

                Console.WriteLine("Data loaded... 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 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" ] = "333-3333333333";

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[ "Gender" ] = "M";

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["Gender"] = "M"; // Mens.
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["Gender"] = "W";
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[ "Gender" ] = "W"; // Women.
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[ "Gender" ] = "B"; // Boys.
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);

    }

}

}
```

Load Data into Tables Using the AWS SDK for PHP

Note

This topic assumes that you are already following the instructions for [Getting Started with DynamoDB \(p. 13\)](#) and have the AWS SDK for PHP properly installed. For information about setting up the SDK, configuring your AWS access keys and setting the default endpoint, go to [Running PHP Examples \(p. 371\)](#).

After you create a table and the table is in the ACTIVE state, you can begin performing data operations on the table.

Example - Upload Sample Items Using the AWS SDK for PHP

The following PHP code example adds items to your existing tables using the PHP command `put_item`. Notice the following code example puts 8 items in the `ProductCatalog` table. The table has a write capacity units value of 5. You might see `ProvisionedThroughputExceeded` errors in the response from DynamoDB. However, the AWS SDKs retry requests for this error, and eventually all of the data is written to the table.

Note

After you run this program, see [Verify Data Load \(p. 42\)](#) to view the tables and data in the DynamoDB console.

```
<?php

use Aws\AwsClient;
use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

# Setup some local variables for dates

date_default_timezone_set('UTC');

$oneDayAgo = date('Y-m-d H:i:s', strtotime('-1 days'));
$sevenDaysAgo = date('Y-m-d H:i:s', strtotime('-7 days'));
$fourteenDaysAgo = date('Y-m-d H:i:s', strtotime('-14 days'));
$twentyOneDaysAgo = date('Y-m-d H:i:s', strtotime('-21 days'));

$tableName = 'ProductCatalog';
echo "Adding data to the $tableName table..." . PHP_EOL;

$response = $client->batchWriteItem(array(
    'RequestItems' => array(
        $tableName => array(
            array(
                'PutRequest' => array(
                    'Item' => array(
                        'Id' => array('N' => '1101'),
                        'Title' => array('S' => 'Book 101 Title'),
                        'ISBN' => array('S' => '111-1111111111'),
                        'Authors' => array('SS' => array('Author1')),
                        'Price' => array('N' => '2'),
                        'Dimensions' => array('S' => '8.5 x 11.0 x 0.5'),
                        'PageCount' => array('N' => '500'),
                        'InPublication' => array('N' => '1'),
                        'ProductCategory' => array('S' => 'Book')
                    )
                ),
            ),
            array(
                'PutRequest' => array(
                    'Item' => array(
                        'Id' => array('N' => '102'),
                        'Title' => array('S' => 'Book 102 Title'),
                        'ISBN' => array('S' => '222-2222222222'),
                        'Authors' => array('SS' => array('Author1'),

```

```

'Author2')),
    'Price'          => array('N' => '20'),
    'Dimensions'    => array('S' => '8.5 x 11.0 x 0.8'),
    'PageCount'     => array('N' => '600'),
    'InPublication' => array('N' => '1'),
    'ProductCategory' => array('S' => 'Book')

)
),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'          => array('N' => '103'),
            'Title'       => array('S' => 'Book 103 Title'),
            'ISBN'        => array('S' => '333-333333333'),
            'Authors'     => array('SS' => array('Author1',
'Author2'))),
            'Price'        => array('N' => '2000'),
            'Dimensions'  => array('S' => '8.5 x 11.0 x 1.5'),
            'PageCount'   => array('N' => '600'),
            'InPublication' => array('N' => '0'),
            'ProductCategory' => array('S' => 'Book')

)
),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'          => array('N' => '201'),
            'Title'       => array('S' => '18-Bike-201'),
            'Description' => array('S' => '201 Description'),

            'BicycleType' => array('S' => 'Road'),
            'Brand'        => array('S' => 'Mountain A'),
            'Price'        => array('N' => '100'),
            'Gender'       => array('S' => 'M'),
            'Color'        => array('SS' => array('Red',
'Black'))),
            'ProductCategory' => array('S' => 'Bicycle')

)
),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'          => array('N' => '202'),
            'Title'       => array('S' => '21-Bike-202'),
            'Description' => array('S' => '202 Description'),

            'BicycleType' => array('S' => 'Road'),
            'Brand'        => array('S' => 'Brand-Company A'),
            'Price'        => array('N' => '200'),

```

```

        'Gender'          => array('S' => 'M'),
        'Color'           => array('SS' => array('Green',
'Black'))),
        'ProductCategory' => array('S' => 'Bicycle')
    )
),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'          => array('N' => '203'),
            'Title'        => array('S' => '19-Bike-203'),
            'Description'  => array('S' => '203 Description'),
            'BicycleType'  => array('S' => 'Road'),
            'Brand'         => array('S' => 'Brand-Company B'),
            'Price'          => array('N' => '300'),
            'Gender'        => array('S' => 'W'),
            'Color'          => array('SS' => array('Red', 'Green',
'Black'))),
            'ProductCategory' => array('S' => 'Bicycle')

        )
    ),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'          => array('N' => '204'),
            'Title'        => array('S' => '18-Bike-204'),
            'Description'  => array('S' => '204 Description'),
            'BicycleType'  => array('S' => 'Mountain'),
            'Brand'         => array('S' => 'Brand-Company B'),
            'Price'          => array('N' => '400'),
            'Gender'        => array('S' => 'W'),
            'Color'          => array('SS' => array('Red'))),
            'ProductCategory' => array('S' => 'Bicycle')

        )
    ),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'          => array('N' => '205'),
            'Title'        => array('S' => '20-Bike-205'),
            'Description'  => array('S' => '205 Description'),
            'BicycleType'  => array('S' => 'Hybrid'),
            'Brand'         => array('S' => 'Brand-Company C'),
            'Price'          => array('N' => '500'),
            'Gender'        => array('S' => 'B'),
            'Color'          => array('SS' => array('Red',
'Black'))),
            'ProductCategory' => array('S' => 'Bicycle')

        )
    )
)
)
)
```

```

        )
    ),
),
));
echo "done." . PHP_EOL;

$tableName = 'Forum';
echo "Adding data to the $tableName table..." . PHP_EOL;

$response = $client->batchWriteItem(array(
    'RequestItems' => array(
        $tableName => array(
            array(
                'PutRequest' => array(
                    'Item' => array(
                        'Name' => array('S' => 'Amazon DynamoDB'),
                        'Category' => array('S' => 'Amazon Web Services'),
                        'Threads' => array('N' => '0'),
                        'Messages' => array('N' => '0'),
                        'Views' => array('N' => '1000')
                    )
                )
            ),
            array(
                'PutRequest' => array(
                    'Item' => array(
                        'Name' => array('S' => 'Amazon S3'),
                        'Category' => array('S' => 'Amazon Web Services'),
                        'Threads' => array('N' => '0')
                    )
                )
            )
        )
    )
));
echo "done." . PHP_EOL;

$tableName = 'Reply';
echo "Adding data to the $tableName table..." . PHP_EOL;

$response = $client->batchWriteItem(array(
    'RequestItems' => array(
        $tableName => array(
            array(
                'PutRequest' => array(
                    'Item' => array(
                        'Id' => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 1'),
                        'ReplyDateTime' => array('S' => $fourteenDaysAgo),
                        'Message' => array('S' => 'DynamoDB Thread 1 Reply
2 text'),
                        'PostedBy' => array('S' => 'User B')
                    )
                )
            )
        )
    )
));
echo "done." . PHP_EOL;

```

```

        )
    ),
    array(
        'PutRequest' => array(
            'Item' => array(
                'Id' => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 2'),
                'ReplyDateTime' => array('S' => $twentyOneDaysAgo),
                'Message' => array('S' => 'DynamoDB Thread 2 Reply
3 text'),
                'PostedBy' => array('S' => 'User B')
            )
        )
    ),
    array(
        'PutRequest' => array(
            'Item' => array(
                'Id' => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 2'),
                'ReplyDateTime' => array('S' => $sevenDaysAgo),
                'Message' => array('S' => 'DynamoDB Thread 2 Reply
2 text'),
                'PostedBy' => array('S' => 'User A')
            )
        )
    ),
    array(
        'PutRequest' => array(
            'Item' => array(
                'Id' => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 2'),
                'ReplyDateTime' => array('S' => $oneDayAgo),
                'Message' => array('S' => 'DynamoDB Thread 2 Reply
1 text'),
                'PostedBy' => array('S' => 'User A')
            )
        )
    ),
),
));
echo "done." . PHP_EOL;
?>
```

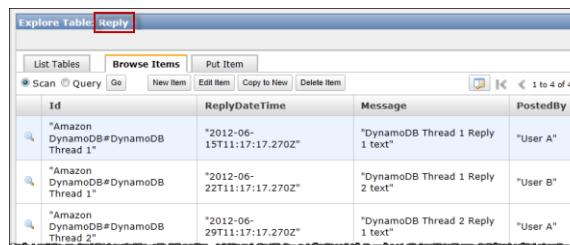
Verify Data Load

Using the AWS Management Console

You can use the AWS Management Console to view the data that you loaded into the tables.

To view table data

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. In the **Tables** pane, select the **Reply** table.
3. Click **Explore Table** to view the items you uploaded. The **Browse Items** tab lists the items in the table.

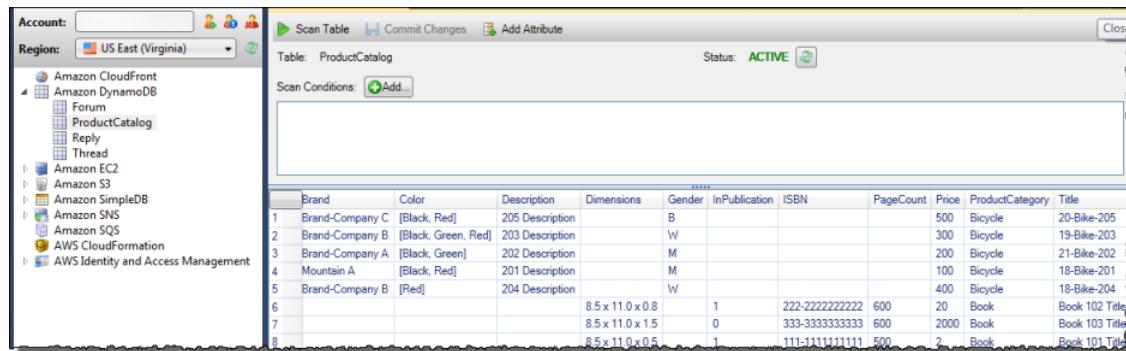


The screenshot shows the 'Explore Table' interface for the 'Reply' table. The 'Browse Items' tab is selected. The table has four columns: Id, ReplyDateTime, Message, and PostedBy. The data is as follows:

Id	ReplyDateTime	Message	PostedBy
"Amazon DynamoDB#DynamoDB Thread 1"	"2012-06-15T11:17:17.270Z"	"DynamoDB Thread 1 Reply 1 text"	"User A"
"Amazon DynamoDB#DynamoDB Thread 1"	"2012-06-22T11:17:17.270Z"	"DynamoDB Thread 1 Reply 2 text"	"User B"
"Amazon DynamoDB#DynamoDB Thread 2"	"2012-06-29T11:17:17.270Z"	"DynamoDB Thread 2 Reply 1 text"	"User A"

Using the AWS Explorer

In addition to the AWS Management Console, you can use the AWS Explorer to see all of your tables and data. The AWS Explorer is available in the AWS Toolkits for Java and .NET.



The screenshot shows the AWS Explorer interface. On the left, there is a tree view of AWS services under an account in the US East (Virginia) region. Under the Amazon DynamoDB node, the 'ProductCatalog' table is selected. The main pane shows the 'Scan Table' interface for the 'ProductCatalog' table, which is currently active. The table has several columns: Brand, Color, Description, Dimensions, Gender, InPublication, ISBN, PageCount, Price, ProductCategory, and Title. Data rows are listed below:

	Brand	Color	Description	Dimensions	Gender	InPublication	ISBN	PageCount	Price	ProductCategory	Title
1	Brand-Company C	[Black, Red]	205 Description		B			500	Bicycle	20-Bike-205	
2	Brand-Company B	[Black, Green, Red]	203 Description		W			300	Bicycle	19-Bike-203	
3	Brand-Company A	[Black, Green]	202 Description		M			200	Bicycle	21-Bike-202	
4	Mountain A	[Black, Red]	201 Description		M			100	Bicycle	18-Bike-201	
5	Brand-Company B	[Red]	204 Description		W			400	Bicycle	18-Bike-204	
6				8.5 x 11.0 x 0.8		1	222-2222222222	600	20	Book	Book 102 Title
7				8.5 x 11.0 x 1.5		0	333-3333333333	600	2000	Book	Book 103 Title
8				8.5 x 11.0 x 0.5		1	111-1111111111	500	2	Book	Book 101 Title

Step 4: Try a Query

Topics

- Try a Query Using the DynamoDB Console (p. 43)
- Try a Query Using the AWS SDK for Java (p. 44)
- Try a Query Using the AWS SDK for .NET (p. 46)
- Try a Query Using the AWS SDK for PHP (p. 52)

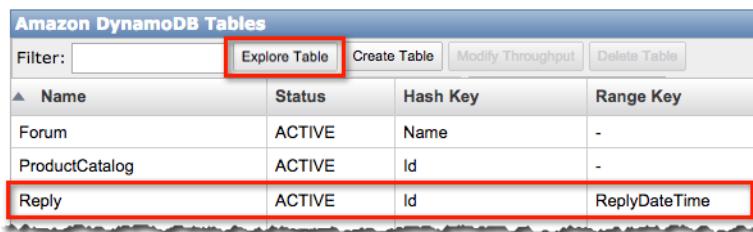
In this step, you will try a simple query against the tables that you created in the preceding step. You can either use the DynamoDB console to query the tables or query the table programmatically.

Try a Query Using the DynamoDB Console

In this section, you will use the DynamoDB console to try an example query against the Reply table. The query finds forum replies posted in the last 15 days for the "DynamoDB Thread 1" thread in the "DynamoDB" forum.

To query a table

1. Open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/home>. If you have not already signed in, you will see the **Sign In** dialog before you see the console.
2. In the **Tables** pane, select the Reply table and click **Explore Table**.



Name	Status	Hash Key	Range Key
Forum	ACTIVE	Name	-
ProductCatalog	ACTIVE	Id	-
Reply	ACTIVE	Id	ReplyDateTime

3. In the **Browse Items** tab, click **Query**.

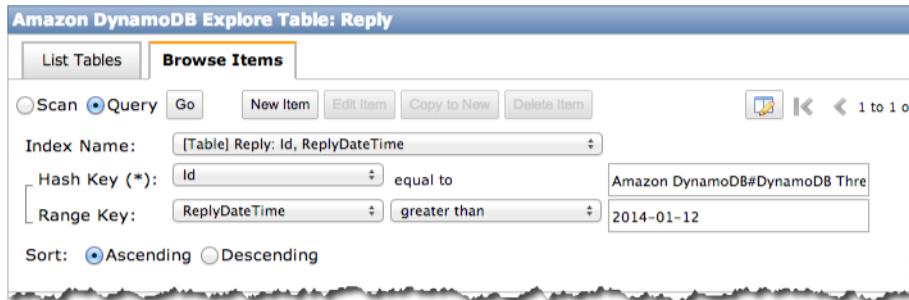
The console shows data entry fields for you to specify the hash and range primary key values, **Hash Key** and **Range Key**. It also shows drop-down list boxes for you to select comparison operators.

Note

A **Query** operation is only valid for tables that have a hash and range type primary key. If you explore a table with a hash type primary key, the console will display **Get** instead of **Query**.

4. Specify the **Hash Key** and **Range Key** values and select comparison operators as shown in the following screen shot. For the **Hash Key**, enter Amazon DynamoDB#DynamoDB Thread 1. For the **Range Key (ReplyDateTime)**, set the condition to *greater than* and enter a date 15 days earlier than today's date. Use the format YYYY-MM-DD for the date.

Note that the **Range Key (ReplyDateTime)** value shown is only for illustration. The date value you will use depends on when you uploaded the sample data.



5. Click **Go**.

The **Browse Items** tab shows the query result.

Try a Query Using the AWS SDK for Java

The following Java code example uses the AWS SDK for Java to perform the following tasks:

- Get an item from the ProductCatalog table.
- Query the Reply table to find all replies posted in the last 15 days for a forum thread. In the code, you first describe your request by creating a `QuerySpec` object. The `QuerySpec` describes the primary key hash attribute value, a condition on the range attribute (`ReplyDateTime`) to retrieve replies posted after a specific date, and the item attributes that you want to retrieve.

For step-by-step instructions on configuring your AWS access keys, setting the default endpoint and running the sample, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Iterator;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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.RangeKeyCondition;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;

public class GettingStartedTryQuery {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));
    static SimpleDateFormat dateFormatter = new SimpleDateFormat(
        "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
}

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

    try {

        String forumName = "Amazon DynamoDB";
        String threadSubject = "DynamoDB Thread 1";

        // Get an item.
        getBook(101, "ProductCatalog");

        // Query replies posted in the past 15 days for a forum thread.
        findRepliesInLast15DaysWithConfig("Reply", forumName, threadSubject);

    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
}

private static void getBook(int id, String tableName) {

    Table table = dynamoDB.getTable(tableName);

    Item item = table.getItem("Id", // attribute name
        id, // attribute value
        "Id, ISBN, Title, Authors", // projection expression
        null); // name map - don't need this

    System.out.println("GetItem: printing results...");
    System.out.println(item.toJSONPretty());
}
}
```

```
private static void findRepliesInLast15DaysWithConfig(
    String tableName, String forumName, String threadSubject) {

    String replyId = forumName + "#" + threadSubject;
    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);

    Table table = dynamoDB.getTable(tableName);

    QuerySpec querySpec = new QuerySpec()
        .withHashKey("Id", replyId)
        .withRangeKeyCondition(
            new RangeKeyCondition("ReplyDateTime")
                .gt(twoWeeksAgoStr))
        .withProjectionExpression("Message, ReplyDateTime, PostedBy");

    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());
    }
}
```

Try a Query Using the AWS SDK for .NET

The following C# code example uses the AWS SDK for .NET low-level API to perform the following tasks:

- Get an item from the ProductCatalog table.
- Query the Reply table to find all replies posted in the last 15 days for a forum thread. In the code, you first describe your request by creating a `QueryRequest` object. The request specifies the table name, the primary key hash attribute value, a condition on the range attribute (`ReplyDateTime`) to retrieve replies posted after a specific date, and other optional parameters. The example uses pagination to retrieve one page of query results at a time. It sets the page size as part of the request.

For step-by-step instructions on configuring your AWS access keys, setting the default endpoint and running the sample, see [Running .NET Examples for DynamoDB \(p. 369\)](#).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDBv2;
```

```
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.Util;

namespace com.amazonaws.codesamples
{

    class GettingStartedTryQuery
    {

        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                // Get  - Get a book item.

                GetBook(101, "ProductCatalog");

                // Query - Get replies posted in the last 15 days for a forum
                thread.

                string forumName = "Amazon DynamoDB";
                string threadSubject = "DynamoDB Thread 1";

                FindRepliesInLast15DaysWithConfig(forumName, threadSubject);

                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(int id, string tableName)
    {
        var request = new GetItemRequest
        {
            TableName = tableName,
            Key = new Dictionary<string, AttributeValue>()
        {
            { "Id", new AttributeValue { N = id.ToString() } }
        },
            ReturnConsumedCapacity = "TOTAL"
        };

        var response = client.GetItem(request);

        Console.WriteLine("No. of reads used (by get book item) {0}\n",
            response.ConsumedCapacity.CapacityUnits);

        PrintItem(response.Item);

        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
    }

    private static void FindRepliesInLast15DaysWithConfig(string forumName,
        string threadSubject)
```

```
{  
  
    string replyId = forumName + "#" + threadSubject;  
  
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);  
  
    string twoWeeksAgoString =  
        twoWeeksAgoDate.ToString(AWSSDKUtils.ISO8601DateFormat);  
  
    Dictionary<string, AttributeValue> lastKeyEvaluated = null;  
  
    do  
  
    {  
  
        var request = new QueryRequest  
  
        {  
  
            TableName = "Reply",  
  
            KeyConditions = new Dictionary<string, Condition>()  
  
        }  
  
        {  
  
            "Id",  
            new Condition()  
  
            {  
  
                ComparisonOperator = "EQ",  
  
                AttributeValueList = new List<AttributeValue>()  
  
                {  
  
                    new AttributeValue { S = replyId }  
  
                }  
  
            }  
  
        },  
  
        {  
  
            "ReplyDateTime",  
  
            new Condition()  
  
            {  
  
                ComparisonOperator = "LT",  
  
               AttributeValueList = new List<AttributeValue>()  
  
                {  
  
                    new AttributeValue { S = twoWeeksAgoString }  
  
                }  
  
            }  
  
        }  
  
    }  
  
}
```

```
        new Condition()
    {
        ComparisonOperator = "GT",
        AttributeValueList = new List<AttributeValue>()
    {
        new AttributeValue { S = twoWeeksAgoString }
    }
}
},
// Optional parameter.

ProjectionExpression = "Id, ReplyDateTime, PostedBy",

// Optional parameter.

ConsistentRead = true,
Limit = 2, // The Reply table has only a few sample items.
So the page size is smaller.

ExclusiveStartKey = lastKeyEvaluated,
ReturnConsumedCapacity = "TOTAL"
};

var response = client.Query(request);

Console.WriteLine("No. of reads used (by query in FindReplies
ForAThreadSpecifyLimit) {0}\n",
response.ConsumedCapacity.CapacityUnits);

foreach (var 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 PrintItem(Dictionary<string, AttributeValue> attributeList)
{
    foreach (var 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("*****");
}
```

Try a Query Using the AWS SDK for PHP

Note

This topic assumes you are already following the instructions for [Getting Started with DynamoDB \(p. 13\)](#) and have the AWS SDK for PHP properly installed. For information about setting up the SDK, configuring your AWS access keys and setting the default endpoint, go to [Running PHP Examples \(p. 371\)](#).

Example - Query for Items in your DynamoDB Tables with PHP

The following PHP code example uses the AWS SDK for PHP to query the Reply table for all replies posted less than 14 days ago for a forum thread. The request specifies the table name, the primary key hash attribute value, and a condition on the range attribute (ReplyDateTime) to retrieve replies posted after a specific date.

```
<?php

use Aws\AwsClient;
use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

date_default_timezone_set('UTC');

$fourteenDaysAgo = date('Y-m-d H:i:s', strtotime('-14 days'));

$response = $client->query(array(
    'TableName' => 'Reply',
    'KeyConditions' => array(
        'Id' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array(
                array('S' => 'Amazon DynamoDB#DynamoDB Thread 2')
            )
        ),
        'ReplyDateTime' => array(
            'ComparisonOperator' => 'GE',
            'AttributeValueList' => array(
                array('S' => $fourteenDaysAgo)
            )
        )
    )
));
print_r($response['Items']);
?>
```

Step 5: Delete Example Tables

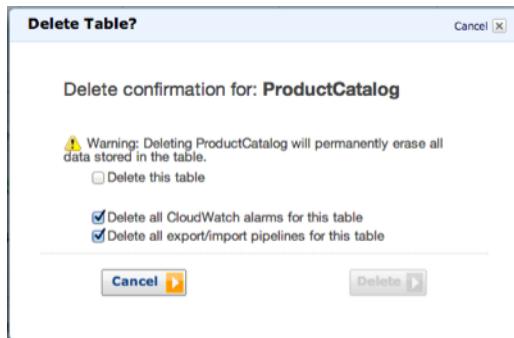
These tables are also used in various sections of this developer guide to illustrate table and item operations using various AWS SDKs. Don't delete these tables if you are reading the rest of the developer guide.

However, if you don't plan to use these tables, you should delete them to avoid getting charged for resources you don't use.

You can also delete tables programmatically. For more information, see [Working with Tables in DynamoDB \(p. 54\)](#).

To Delete the Sample Tables

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. Select the table that you want to delete.
3. Click the **Delete Table** button. You will be asked to confirm your selection.



4. Select the **Delete this table** check box and click **Delete**.

This deletes the table from DynamoDB, along with the CloudWatch alarms and export/import pipelines associated with this table.

Where Do I Go from Here?

Now that you have tried the getting started exercise, you can explore the following sections to learn more about DynamoDB:

- [Working with Tables in DynamoDB \(p. 54\)](#)
- [Working with Items in DynamoDB \(p. 85\)](#)
- [Query and Scan Operations in DynamoDB \(p. 183\)](#)
- [Improving Data Access with Secondary Indexes in DynamoDB \(p. 241\)](#)
- [Using the AWS SDKs with DynamoDB \(p. 365\)](#)

Working with Tables in DynamoDB

Topics

- [Specifying the Primary Key \(p. 54\)](#)
- [Specifying Read and Write Requirements for Tables \(p. 55\)](#)
- [Capacity Units Calculations for Various Operations \(p. 57\)](#)
- [Listing and Describing Tables \(p. 59\)](#)
- [Guidelines for Working with Tables \(p. 59\)](#)
- [Working with Tables Using the AWS SDK for Java Document API \(p. 66\)](#)
- [Working with Tables Using the AWS SDK for .NET Low-Level API \(p. 71\)](#)
- [Working with Tables Using the AWS SDK for PHP Low-Level API \(p. 78\)](#)

When you create a table in Amazon DynamoDB, you must provide a table name, its primary key and your required read and write throughput values. The table name can include characters a-z, A-Z, 0-9, '_' (underscore), '-' (dash), and '.' (dot). Names can be between 3 and 255 characters long. In a relational database, a table has a predefined schema that describes properties such as the table name, primary key, column names, and data types. All records stored in the table must have the same set of columns. DynamoDB is a NoSQL database: Except for the required primary key, a DynamoDB table is schema-less. Individual items in a DynamoDB table can have any number of attributes, although there is a limit of 400 KB on the item size.

Specifying the Primary Key

When you create a table, in addition to the table name, you must specify the primary key of the table. DynamoDB supports the following two types of primary keys:

- **Hash Primary Key** – The primary key is made of one attribute, a hash attribute. For example, a *ProductCatalog* table can have ProductID as its primary key. DynamoDB builds an unordered hash index on this primary key attribute.
- **Hash and Range Primary Key** – The primary key is made of two attributes. The first attribute is the hash attribute and the second attribute is the range attribute. For example, the forum *Thread* table can have ForumName and Subject as its primary key, where ForumName is the hash attribute and Subject is the range attribute. DynamoDB builds an unordered hash index on the hash attribute and a sorted range index on the range attribute.

Specifying Read and Write Requirements for Tables

DynamoDB is built to support workloads of any scale with predictable, low-latency response times.

To ensure high availability and low latency responses, DynamoDB requires that you specify your required read and write throughput values when you create a table. DynamoDB uses this information to reserve sufficient hardware resources and appropriately partitions your data over multiple servers to meet your throughput requirements. As your application data and access requirements change, you can easily increase or decrease your provisioned throughput using the DynamoDB console or the API.

DynamoDB allocates and reserves resources to handle your throughput requirements with sustained low latency and you pay for the hourly reservation of these resources. However, you pay as you grow and you can easily scale up or down your throughput requirements. For example, you might want to populate a new table with a large amount of data from an existing data store. In this case, you could create the table with a large write throughput setting, and after the initial data upload, you could reduce the write throughput and increase the read throughput to meet your application's requirements.

During the table creation, you specify your throughput requirements in terms of the following capacity units. You can also specify these units in an [UpdateTable](#) request to increase or decrease the provisioned throughput of an existing table:

- **Read capacity units** – The number of strongly consistent reads per second of items up to 4 KB in size per second. For example, when you request 10 read capacity units, you are requesting a throughput of 10 strongly consistent reads per second of 4 KB for that table. For eventually consistent reads, one read capacity unit is two reads per second for items up to 4 KB. For more information about read consistency, see [Data Read and Consistency Considerations \(p. 9\)](#).
- **Write capacity units** – The number of 1 KB writes per second. For example, when you request 10 write capacity units, you are requesting a throughput of 10 writes per second of 1 KB size per second for that table.

DynamoDB uses these capacity units to provision sufficient resources to provide the requested throughput.

When deciding the capacity units for your table, you must take the following into consideration:

- **Item size** – DynamoDB allocates resources for your table according to the number of read or write capacity units that you specify. These capacity units are based on a data item size of 4 KB per read or 1 KB per write. For example, if the items in your table are 4 KB or smaller, each item read operation will consume one read capacity unit. If your items are larger than 4 KB, each read operation consumes additional capacity units, in which case you can perform fewer database read operations per second than the number of read capacity units you have provisioned. For example, if you request 10 read capacity units throughput for a table, but your items are 8 KB in size, then you will get a maximum of 5 strongly consistent reads per second on that table.
- **Expected read and write request rates** – You must also determine the expected number of read and write operations your application will perform against the table, per second. This, along with the estimated item size helps you to determine the read and write capacity unit values.
- **Consistency** – Read capacity units are based on strongly consistent read operations, which require more effort and consume twice as many database resources as eventually consistent reads. For example, a table that has 10 read capacity units of provisioned throughput would provide either 10 strongly consistent reads per second of 4 KB items, or 20 eventually consistent reads per second of the same items. Whether your application requires strongly or eventually consistent reads is a factor in determining how many read capacity units you need to provision for your table. By default, DynamoDB read operations are eventually consistent. Some of these operations allow you to specify strongly consistent reads.

- **Local secondary indexes** – If you want to create one or more local secondary indexes on a table, you must do so at table creation time. DynamoDB automatically creates and maintains these indexes. Queries against indexes consume provisioned read throughput. If you write to a table, DynamoDB will automatically write data to the indexes when needed, to keep them synchronized with the table. The capacity units consumed by index operations are charged against the table's provisioned throughput. In other words, you only specify provisioned throughput settings for the table, not for each individual index on that table. For more information, see [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 309\)](#).

These factors help you to determine your application's throughput requirements that you provide when you create a table. You can monitor the performance using CloudWatch metrics, and even configure alarms to notify you in the event you reach certain threshold of consumed capacity units. The DynamoDB console provides several default metrics that you can review to monitor your table performance and adjust the throughput requirements as needed. For more information, go to [DynamoDB Console](#).

DynamoDB automatically distributes your data across table partitions, which are stored on multiple servers. For optimal throughput, you should distribute read requests as evenly as possible across these partitions. For example, you might provision a table with 1 million read capacity units per second. If you issue 1 million 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 of the items in the table, DynamoDB can access the table partitions in parallel, and allow you to reach your provisioned throughput goal for the table.

For reads, the following table compares some provisioned throughput values for different average item sizes, request rates, and consistency combinations.

Expected Item Size	Consistency	Desired Reads Per Second	Provisioned Throughput Required
4 KB	Strongly consistent	50	50
8 KB	Strongly consistent	50	100
4 KB	Eventually consistent	50	25
8 KB	Eventually consistent	50	50

Item sizes for reads are rounded up to the next 4 KB multiple. For example, an item of 3,500 bytes consumes the same throughput as a 4 KB item.

For writes, the following table compares some provisioned throughput values for different average item sizes and write request rates.

Expected Item Size	Desired Writes Per Second	Provisioned Throughput Required
1 KB	50	50
2 KB	50	100

Item sizes for writes are rounded up to the next 1 KB multiple. For example, an item of 500 bytes consumes the same throughput as a 1 KB item.

DynamoDB commits resources to your requested read and write capacity units, and, consequently, you are expected to stay within your requested rates. Provisioned throughput also depends on the size of the requested data. If your read or write request rate, combined with the cumulative size of the requested

If your provisioned throughput is exceeded, DynamoDB returns an error that indicates that the provisioned throughput level has been exceeded.

Set your provisioned throughput using the *ProvisionedThroughput* parameter. For information about setting the *ProvisionedThroughput* parameter, see [CreateTable](#) in the Amazon DynamoDB API Reference.

For information about using provisioned throughput, see [Guidelines for Working with Tables \(p. 59\)](#).

Note

If you expect upcoming spikes in your workload (such as a new product launch) that will cause your throughput to exceed the current provisioned throughput for your table, we advise that you use the [UpdateTable](#) operation to increase the *ProvisionedThroughput* value. For the current maximum Provisioned Throughput values per table or account, see [Limits in DynamoDB \(p. 597\)](#). 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`. During this time, DynamoDB allocates the necessary resources to support the new provisioned throughput levels. When this process is completed, the table status changes from `UPDATING` to `AVAILABLE`.

Capacity Units Calculations for Various Operations

The capacity units consumed by an operation depends on the following:

- Item size
- Read consistency (in case of a read operation)

For a table without local secondary indexes, the basic rule is that if your request reads a item of 4 KB or writes an item of 1 KB in size, you consume 1 capacity unit. This section describes how DynamoDB computes the item size for the purpose of determining capacity units consumed by an operation. In the case of a read operation, this section describes the impact of strong consistency vs. eventual consistency read on the capacity unit consumed by the read operation.

Item Size Calculations

For each request that you send, DynamoDB computes the capacity units consumed by that operation. Item size is one of the factors that DynamoDB uses in computing the capacity units consumed. This section describes how DynamoDB determines the size of items involved in an operation.

Note

You can optimize the read capacity consumption by making individual items as small as possible. The easiest way to do so is to minimize the length of the attribute names. You can also reduce item size by storing less frequently accessed attributes in a separate table.

The size of an item is the sum of the lengths of its attribute names and values.

The size of a Null or Boolean attribute value is (length of the attribute name + one byte).

An attribute of type List or Map requires 3 bytes of overhead, regardless of its contents. The size of an empty List or Map is (length of the attribute name + 3 bytes). If the attribute is non-empty, the size is (length of the attribute name + sum (length of attribute values) + 3 bytes).

DynamoDB reads data in blocks of 4 KB. For `GetItem`, which reads only one item, DynamoDB rounds the item size up to the next 4 KB. For example, if you get an item of 3.5 KB, DynamoDB rounds the items size to 4 KB. If you get an item of 10 KB, DynamoDB rounds the item size to 12 KB.

DynamoDB writes data in blocks of 1 KB. For `PutItem`, `UpdateItem`, and `DeleteItem`, which write only one item, 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.

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.

Most write operations in DynamoDB allow *conditional writes*, where you specify one or more conditions that must be met in order for the operation to succeed. Even if a conditional write fails, it still consumes provisioned throughput. 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.

For `BatchGetItem`, each item in the batch is read separately, so DynamoDB first rounds up the size of each item to the next 4 KB 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).

For `Query`, all items returned are treated as a single read operation. As a result, 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.

In the case of a `Scan` operation, DynamoDB considers the size of the items that are evaluated, not the size of the items returned by the scan. For a scan request, DynamoDB evaluates up to 1 MB of items and returns only the items that satisfy the scan condition.

Note

In computing the storage used by the table, DynamoDB adds 100 bytes of overhead to each item for indexing purposes. The `DescribeTable` operation returns a table size that includes this overhead. This overhead is also included when billing you for the storage costs. However, this extra 100 bytes is not used in computing the capacity unit calculation. For more information about pricing, go to [DynamoDB Pricing](#).

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.

The `PutItem` operation adds an item to the 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.

For an `UpdateItem` operation, 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).

When you issue a `DeleteItem` request, DynamoDB uses the size of the deleted item to calculate provisioned throughput consumption.

Read Operation and Consistency

For a read operation, 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.

Listing and Describing Tables

To obtain a list of all your tables, use the `ListTables` operation. A single `ListTables` call can return a maximum of 100 table names; if you have more than 100 tables, you can request that `ListTables` return paginated results, so that you can retrieve all of the table names.

To determine the structure of any table, use the `DescribeTable` operation. The metadata returned by `DescribeTable` includes the timestamp when it was created, its key schema, its provisioned throughput settings, its estimated size, and any secondary indexes that are present.

Note

If you issue a `DescribeTable` request immediately after a `CreateTable` request, DynamoDB might return a `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.

Guidelines for Working with Tables

Topics

- [Design For Uniform Data Access Across Items In Your Tables \(p. 59\)](#)
- [Understand Partition Behavior \(p. 61\)](#)
- [Use Burst Capacity Sparingly \(p. 62\)](#)
- [Distribute Write Activity During Data Upload \(p. 62\)](#)
- [Understand Access Patterns for Time Series Data \(p. 63\)](#)
- [Cache Popular Items \(p. 64\)](#)
- [Consider Workload Uniformity When Adjusting Provisioned Throughput \(p. 64\)](#)
- [Test Your Application At Scale \(p. 65\)](#)

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

When it stores data, DynamoDB divides a table's items into multiple partitions, and distributes the data primarily based upon the hash key element. The provisioned throughput associated with a table is also divided evenly among the partitions, with no sharing of provisioned throughput across partitions.

$$\text{Total Provisioned Throughput} / \text{Partitions} = \text{Throughput Per Partition}$$

Consequently, to achieve the full amount of request throughput you have provisioned for a table, keep your workload spread evenly across the hash key values. Distributing requests across hash key values distributes the requests across partitions.

For example, if a table has a very small number of heavily accessed hash key elements, possibly even a single very heavily used hash key element, 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 hash key element 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 hash keys to achieve your throughput level; nor does it mean that the percentage of accessed hash keys needs to be high. However, do be aware that when your workload accesses more distinct hash keys, 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 hash keys accessed to total hash keys in a table grows.

Choosing a Hash Key

The following table compares some common hash key schema for provisioned throughput efficiency:

Hash 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
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 hash key values, consider distributing your write operations across more distinct hash values. In other words, structure the primary key elements to avoid one "hot" (heavily requested) hash key value that slows overall performance.

For example, consider a table with a hash and range type primary key. The hash key represents the item's creation date, rounded to the nearest day. The range key is an item identifier. On a given day, say *2014-07-09*, all of the new items will be written to that same hash 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 Hash Key Values

One way to increase the write throughput of this application would be to randomize the writes across multiple hash 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 hash 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 hash key, the writes to the table on each day are spread evenly across all of the hash 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 hash key `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 hash 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 ASCII values for the characters in the order ID, modulo $200 + 1$. The hash 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 hash keys, and thus across the partitions. You can easily perform a `GetItem` operation on a particular item, because you can calculate the hash key 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" hash key taking all of the workload.

Understand Partition Behavior

DynamoDB manages table partitioning for you automatically, adding new partitions as your table grows in size. You can estimate the number of partitions that DynamoDB will create for your table, and compare that estimate against your scale and access patterns. This can help you determine the best table design for your application needs.

The number of partitions in a table is based on the table's storage requirements, and also its provisioned throughput requirements.

Number of partitions required, based solely on a table's size

- $numPartitions_{tableSize} = tableSizeInBytes / 10 \text{ GB}$

The size of the table is one factor in determining the number of partitions needed. The other factor is the table's provisioned throughput requirements. For any one partition, DynamoDB can allocate a maximum of 3000 read capacity units or 1000 write capacity units.

Number of partitions required, based solely on a table's provisioned read and write throughput settings

- $numPartitions_{throughput} = (readCapacityUnits / 3000) + (writeCapacityUnits / 1000)$

For example, suppose that you provisioned a table with 1000 read capacity units and 500 write capacity units. In this case, $numPartitions_{throughput}$ would be:

$$(1000 / 3000) + (500 / 1000) = 0.8333$$

Therefore, a single partition could accommodate all of the table's provisioned throughput requirements.

However, if you provisioned 1000 read capacity units and 1000 write capacity units, then $numPartitions_{throughput}$ would exceed a single partition's throughput capacity:

$$(\frac{1000}{3000}) + (\frac{1000}{1000}) = 1.333$$

In this case, the table would require two partitions, each with 500 read capacity units and 500 write capacity units.

The total number of partitions allocated by DynamoDB is the larger of the table's size or its provisioned throughput requirements:

Total number of partitions allocated by DynamoDB

- $numPartitions_{total} = \max(numPartitions_{tableSize}, numPartitions_{throughput})$

A single partition can hold approximately 10 GB of data. If your table size grows beyond 10 GB, DynamoDB will spread your data across additional partitions, and will also distribute your table's read and write throughput accordingly. Therefore, as your table grows in size, less throughput will be provisioned per partition.

Suppose that you have a table that has grown to 500 GB in size. This would mean that the table now occupies approximately 50 partitions:

$$500 \text{ GB} / 10 \text{ GB} = 50 \text{ partitions}$$

Now suppose that you have allocated 100,000 read capacity units to the table. You could determine the amount of read capacity per partition as follows:

$$100,000 \text{ read capacity units} / 50 \text{ partitions} = 2000 \text{ read capacity units per partition}$$

Note

In the future, these details of partition sizes and throughput allocation per partition may change.

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 reserves a portion of your unused capacity for later "bursts" of throughput usage. DynamoDB currently reserves up to 5 minutes (300 seconds) of unused read and write capacity. During an occasional burst of read or write activity, this reserved throughput 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 hash and range type primary key in which UserID is the hash attribute and the MessageID is the range attribute. 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 hash key values. You are taking one hash key at a time and uploading all its items before going to the next hash key items. 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 hash keys. In this case, by directing an uneven amount of upload work toward items all with the same hash key, 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 hash key first. Then you repeat the pattern for the next set of range keys 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 hash key, 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 and reserves 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 hash and range type primary key with Customer ID as the hash attribute and date/time as the range attribute. 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 [Example Tables and Data \(p. 609\)](#), 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 \(p. 62\)](#). 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 \(p. 61\)](#)), 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 \(p. 61\)](#)); 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 hash key, 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 hash key values, but in a non-uniform fashion (accessing a small number of hash keys 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 \(p. 61\)](#).)

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 \(p. 63\)](#).

Working with Tables Using the AWS SDK for Java Document API

Topics

- [Creating a Table \(p. 66\)](#)
- [Updating a Table \(p. 67\)](#)
- [Deleting a Table \(p. 68\)](#)
- [Listing Tables \(p. 68\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for Java Document API \(p. 69\)](#)

You can use the AWS SDK for Java Document API 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. For more information, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#). The following Java code snippet creates an example table that uses a numeric type attribute Id as its primary key.

The following are the steps to create a table using the AWS SDK for Java Document 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 Java code snippet demonstrates the preceding steps.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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));
```

```
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.

```
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 `describeTable` method of the client to get table information at any time.

```
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 the read capacity units and write capacity units anytime. However, you can decrease these values only four times in a 24 hour period. For additional guidelines and limitations, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

The following are the steps to update a table using the AWS SDK for Java Document 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 Java code snippet demonstrates the preceding steps.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
Table table = dynamoDB.getTable("ProductCatalog");  
  
ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()  
    .withReadCapacityUnits(15L)  
    .withWriteCapacityUnits(12L);  
  
table.updateTable(provisionedThroughput);  
  
table.waitForActive();
```

Deleting a Table

The following are the steps 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 Java code snippet demonstrates the preceding steps.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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 Java example 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 [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    static String tableName = "ExampleTable";

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

        createExampleTable();
        listMyTables();
        getTableInformation();
        updateExampleTable();

        deleteExampleTable();
    }

    static void createExampleTable() {

        try {

            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));
```

```
CreateTableRequest request = new CreateTableRequest()
.withTableName(tableName)
.withKeySchema(keySchema)
.withAttributeDefinitions(attributeDefinitions)
.withProvisionedThroughput(new ProvisionedThroughput()
    .withReadCapacityUnits(5L)
    .withWriteCapacityUnits(6L));
```

```
System.out.println("Issuing CreateTable request for " + tableName);
```

```
Table table = dynamoDB.createTable(request);
```

```
System.out.println("Waiting for " + tableName
    + " to be created...this may take a while...");
```

```
table.waitForActive();
```

```
getTableInformation();
```

```
} catch (Exception e) {
    System.err.println("CreateTable request failed for " + tableName);
    System.err.println(e.getMessage());
}
```

```
}
```

```
static void listMyTables() {
```

```
TableCollection<ListTablesResult> tables = dynamoDB.listTables();
Iterator<Table> iterator = tables.iterator();
```

```
System.out.println("Listing table names");
```

```
while (iterator.hasNext()) {
    Table table = iterator.next();
    System.out.println(table.getTableName());
}
}
```

```
static void getTableInformation() {
```

```
System.out.println("Describing " + tableName);
```

```
TableDescription tableDescription = dynamoDB.getTable(tableName).de
scribe();
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 Using the AWS SDK for .NET Low-Level API

Topics

- [Creating a Table \(p. 72\)](#)
- [Updating a Table \(p. 73\)](#)
- [Deleting a Table \(p. 74\)](#)
- [Listing Tables \(p. 74\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for .NET Low-Level API \(p. 75\)](#)

You can use the AWS SDK for .NET low-level API (protocol-level API) to create, update, and delete tables, list all the tables in your account, or get information about a specific table. These operations map to the corresponding DynamoDB API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

The following are the common steps for table operations using the .NET low-level API.

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.

Creating a Table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values. For more information, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

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"
        }
    },
    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.

```
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.

```
var res = client.DescribeTable(new DescribeTableRequest{TableName = "Product Catalog"});
```

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 the read capacity units and write capacity units anytime. You can also decrease read capacity units anytime. However, you can decrease write capacity units only four times in a 24 hour period. Any change you make must be at least 10% different from the current values. For additional guidelines and limitations, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

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.

```
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.

```
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.

```
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 uses the AWS SDK for .NET low-level API to create, update, and delete 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 [Using the AWS SDK for .NET \(p. 368\)](#).

```
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"
    },
    new KeySchemaElement
    {
        AttributeName = "ReplyDateTime",
        KeyType = "RANGE"
    }
},
    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.ReadCapacity
Units,
                tableDescription.ProvisionedThroughput.WriteCapa
cityUnits);

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.ReadCapacity
Units);
    Console.WriteLine("Provision Throughput (writes/sec): {0}",
                      description.ProvisionedThroughput.WriteCapacity
Units);
}

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 Tables Using the AWS SDK for PHP Low-Level API

Topics

- [Creating a Table \(p. 79\)](#)
- [Updating a Table \(p. 80\)](#)
- [Deleting a Table \(p. 81\)](#)
- [Listing Tables \(p. 81\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for PHP Low-Level API \(p. 82\)](#)

You can use the AWS SDK for PHP to create, update, and delete tables, list all the tables in your account, or get information about a specific table. These operations map to the corresponding DynamoDB API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

The following are the common steps for table operations using the AWS SDK for PHP API.

1. Create an instance of the `DynamoDbClient` client class.
2. Provide the parameters for a DynamoDB operation to the client instance, including any optional parameters.
3. Load the response from DynamoDB into a local variable for your application.

Creating a Table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values. For more information, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#). The following PHP code sample creates an `ExampleTable` that uses a numeric type attribute `Id` as its primary key.

The following are the steps to create a table using the AWS SDK for PHP.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `createTable` operation to the client instance.

You must provide the table name, its primary key, attribute type definitions, and the provisioned throughput values.

3. Load the response into a local variable, such as `$response`, for use in your application.

The following PHP code snippet demonstrates the preceding steps. The code 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` method.

```
use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2'  #replace with your desired region
));

$tableName = 'ExampleTable';

echo "# Creating table $tableName..." . PHP_EOL;

$result = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Id',
            'AttributeType' => 'N'
        )
    ),
    'KeySchema' => array(
        array(
            'AttributeName' => 'Id',
            'KeyType' => 'HASH'
        )
    )
));
```

```
        )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 5,
    'WriteCapacityUnits' => 6
)
));
print_r($result->getPath('TableDescription'));
```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE before you can put data into the table. You can use the client's `waitUntil` function to wait until the table's status becomes ACTIVE. For more information, see the [DescribeTable](#) operation.

The following code snippet demonstrates a sleep operation to wait for the table to be in the ACTIVE state.

```
$client->waitUntilTableExists(array('TableName' => $tableName));
echo "table $tableName has been created." . PHP_EOL;
```

Updating a Table

You can update the provisioned throughput values of an existing table. Depending on your application requirements, you might need to update these values.

Note

You can increase the read capacity units and write capacity units anytime. However, you can decrease these values only four times in a 24 hour period. For additional guidelines and limitations, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

The following are the steps to update a table using the AWS SDK for PHP API.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `updateTable` operation to the client instance.

You must provide the table name and the new provisioned throughput values.

3. Load the response into a local variable, such as `$response`, for use in your application.

Immediately after a successful request, the table will be in the UPDATING state until the new values are set. The new provisioned throughput values are available when the table returns to the ACTIVE state.

The following PHP code snippet demonstrates the preceding steps.

```
$tableName = 'ExampleTable';

echo "Updating provisioned throughput settings on $tableName..." . PHP_EOL;

$result = $client->updateTable(array(
    'TableName' => $tableName,
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 6,
        'WriteCapacityUnits' => 7
))
```

```
) .  
  
    // Wait until update completes  
    $client->waitForTableExists(array('TableName' => $tableName));  
  
    echo "New provisioned throughput settings:" . PHP_EOL;  
  
    echo "Read capacity units: " . $result['TableDescription']['ProvisionedThroughput']['ReadCapacityUnits'] . PHP_EOL;  
    echo "Write capacity units: " . $result['TableDescription']['ProvisionedThroughput']['WriteCapacityUnits'] . PHP_EOL;
```

Deleting a Table

The following are the steps to delete a table using the AWS SDK for PHP.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `deleteTable` operation to the client instance.

You must provide the table name for the table to delete.

3. Load the response into a local variable, such as `$response`, for use in your application.

Immediately after a successful request, the table will be in the `DELETING` state until the table and all of the values in the table are removed from the server.

The following PHP code snippet demonstrates the preceding steps.

```
$tableName = 'ExampleTable';  
  
echo "Deleting the table..." . PHP_EOL;  
  
$result = $client->deleteTable(array(  
    'TableName' => $tableName  
));  
  
$client->waitForTableNotExists(array('TableName' => $tableName));  
echo "The table has been deleted." . PHP_EOL;
```

Listing Tables

To list tables in your account using the AWS SDK for PHP, create an instance of the `DynamoDbClient` class and execute the `listTables` operation. 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. You can also set the `ExclusiveStartTableName` parameter. After fetching the first page of results, DynamoDB returns a `LastEvaluatedTableName` value. Use the `LastEvaluatedTableName` value for the `ExclusiveStartTableName` parameter to get the next page of results.

The following PHP code snippet demonstrates how to list all of the tables in your account by using the `LastEvaluatedTableName` value for the `ExclusiveStartTableName` parameter, using a `Limit` value of 2 table names per page.

```
$tables = array();

// Walk through table names, two at a time

do {
    $response = $client->listTables(array(
        'Limit' => 2,
        'ExclusiveStartTableName' => isset($response) ? $response['LastEvaluatedTableName'] : null
    ));

    foreach ($response['TableNames'] as $key => $value) {
        echo "$value" . PHP_EOL;
    }

    $tables = array_merge($tables, $response['TableNames']);
}

while ($response['LastEvaluatedTableName']);

// Print total number of tables

echo "Total number of tables: ";
print_r(count($tables));
echo PHP_EOL;
```

Example: Create, Update, Delete, and List Tables Using the AWS SDK for PHP Low-Level API

The following PHP code example uses the AWS SDK for PHP 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. At the end, the example deletes the table. However, the delete operation is commented-out so you can keep the table and data until you are ready to delete it.

Note

For step-by-step instructions to run the following code example, see [Running PHP Examples \(p. 371\)](#).

```
<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

$tableName = 'ExampleTable';

echo "# Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
```

```
'AttributeDefinitions' => array(
    array(
        'AttributeName' => 'Id',
        'AttributeType' => 'N'
    )
),
'KeySchema' => array(
    array(
        'AttributeName' => 'Id',
        'KeyType' => 'HASH'
    )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 5,
    'WriteCapacityUnits' => 6
)
));
);

print_r($response->getPath('TableDescription'));

$client->waitForTableExists(array('TableName' => $tableName));
echo "table $tableName has been created." . PHP_EOL;

#####
# Updating the table

echo "# Updating the provisioned throughput of table $tableName." . PHP_EOL;

$response = $client->updateTable(array(
    'TableName' => $tableName,
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 6,
        'WriteCapacityUnits' => 7
    )
));
;

// Wait until update completes
$client->waitForTableExists(array('TableName' => $tableName));

echo "New provisioned throughput settings:" . PHP_EOL;

echo "Read capacity units: " . $response['TableDescription']['ProvisionedThroughput']['ReadCapacityUnits'] . PHP_EOL;
echo "Write capacity units: " . $response['TableDescription']['ProvisionedThroughput']['WriteCapacityUnits'] . PHP_EOL;

#####
# Deleting the table

echo "# Deleting table $tableName..." . PHP_EOL;

$response = $client->deleteTable(array(
    'TableName' => $tableName
));
;

$client->waitForTableNotExists(array('TableName' => $tableName));
echo "The table has been deleted." . PHP_EOL;
```

```
#####
# Collect all table names in the account

echo "# Listing all the tables in the account..." . PHP_EOL;

$tables = array();

// Walk through table names, two at a time

do {
    $response = $client->listTables(array(
        'Limit' => 2,
        'ExclusiveStartTableName' => isset($response) ? $response['LastEvaluatedTableName'] : null
    ));

    foreach ($response['TableNames'] as $key => $value) {
        echo "$value" . PHP_EOL;
    }

    $tables = array_merge($tables, $response['TableNames']);
}

while ($response['LastEvaluatedTableName']);

// Print total number of tables

echo "Total number of tables: ";
print_r(count($tables));
echo PHP_EOL;

?>
```

Working with Items in DynamoDB

Topics

- [Overview \(p. 85\)](#)
- [Reading an Item \(p. 86\)](#)
- [Writing an Item \(p. 87\)](#)
- [Batch Operations \(p. 88\)](#)
- [Atomic Counters \(p. 88\)](#)
- [Conditional Writes \(p. 88\)](#)
- [Reading and Writing Items Using Expressions \(p. 91\)](#)
- [Guidelines for Working with Items \(p. 106\)](#)
- [Working with Items Using the AWS SDK for Java Document API \(p. 110\)](#)
- [Working with Items Using the AWS SDK for .NET Low-Level API \(p. 132\)](#)
- [Working with Items Using the AWS SDK for PHP Low-Level API \(p. 167\)](#)

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 [DynamoDB Data Types \(p. 6\)](#).

DynamoDB provides APIs to read and write items. For these operations, you need to specify the items and attributes that you want to work with. When you write an item, you can specify one or more conditions that must evaluate to true. For example, you might want the write to succeed only if an item with the same key does not already exist.

This section describes how to work with items in Amazon DynamoDB. This includes reading and writing items, conditional updates, and atomic counters. This section also includes guidelines for working with items, and example code that uses the AWS SDKs.

Overview

An item consists of one or more attributes. Each attribute consists of a name, a data type, and a value. When you read or write an item, the only attributes that are required are those that make up the primary key.

Important

For the primary key, you must provide *all* of its attributes. For example, with a hash type primary key, you only need to specify the hash attribute. For a hash-and-range type primary key, you

must specify *both* the hash attribute and the range attribute. For more information, see [Primary Key \(p. 5\)](#).

Except for the primary key, there is no predefined schema for the items in a table. For example, to store product information, you can create a *ProductCatalog* table and store various product items in it such as books and bicycles. The following table shows two items, a book and a bicycle, that you can store in the *ProductCatalog* table. Note that the example uses JSON-like syntax to show the attribute value.

Id (Primary Key)	Other Attributes
101	<pre>{ Title = "Book 101 Title" ISBN = "111-1111111111" Authors = "Author 1" Price = "-2" Dimensions = "8.5 x 11.0 x 0.5" PageCount = "500" InPublication = true ProductCategory = "Book" }</pre>
201	<pre>{ Title = "18-Bicycle 201" Description = "201 description" BicycleType = "Road" Brand = "Brand-Company A" Price = "100" Gender = "M" Color = ["Red", "Black"] ProductCategory = "Bike" }</pre>

An item can have any number of attributes, although there is a limit of 400 KB on the item size. An item size is the sum of lengths of its attribute names and values (binary and UTF-8 lengths); it helps if you keep the attribute names short. For more information about attributes and data types, see [DynamoDB Data Model \(p. 3\)](#).

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.

You need to specify the *entire* primary key, not just part of it. For example, if a table has a hash and range type primary key, you must supply a value for the hash attribute and a value for the range attribute.

The following are the default behaviors for `GetItem`:

- `GetItem` performs an eventually consistent read.
- `GetItem` returns all of the item's attributes.
- `GetItem` does not return any information about how many provisioned capacity units it consumes.

You can override these defaults using `GetItem` parameters. For more information, see [GetItem](#) in the [Amazon DynamoDB API Reference](#).

Read Consistency

DynamoDB maintains multiple copies of each item to ensure durability. For every successful write request, DynamoDB ensures that the write is durable on multiple servers. However, it takes time for a write to propagate to all copies. In DynamoDB, data is *eventually consistent*. This means that if you write an item and then immediately try to read it, you might not see the results from the earlier write.

By default, a `GetItem` operation performs an eventually consistent read. You can optionally 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.

An eventually consistent `GetItem` request consumes only half the read capacity units as a strongly consistent request. Therefore, it is best to design applications so that they use eventually consistent reads whenever possible. Consistency across all copies of the data is usually reached within one second.

Writing an Item

To create, update, and delete items in a DynamoDB table, use the following operations:

- `PutItem` — creates a new item. If an item with the same key already exists in the table, it is replaced with the new item. You must provide the table name and the item that you want to write.
- `UpdateItem` — if the item does not already exist, this operation creates a new item; otherwise, it modifies an existing item's attributes. You must specify the table name and the key of the item you want to modify. For each of the attributes that you want to update, you must provide new values for them.
- `DeleteItem` — deletes an item. You must specify the table name and the key of the item you want to delete.

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 hash and range type primary key, you must supply a value for the hash attribute and a value for the range attribute.

If you want to avoid accidentally overwriting or deleting an existing item, you can use a conditional expression with any of these operations. A conditional expression lets you check whether a condition is true (such as an item already existing in the table) before the operation can proceed. For more information, see [Conditional Write Operations \(p. 105\)](#).

In some cases, you might want DynamoDB to display certain attribute values before or after you modify them. For example, with an `UpdateItem` operation, you could request that attribute values be returned as they appeared before the update occurs. `PutItem`, `UpdateItem`, and `DeleteItem` have a `ReturnValues` parameter, which you can use to return the attribute values before or after they are modified.

By default, none of these operations return any information about how many provisioned capacity units they consume. You can use the `ReturnConsumedCapacity` parameter to obtain this information.

For detailed information about these operations, see [PutItem](#), [UpdateItem](#), and [DeleteItem](#) in the [Amazon DynamoDB API Reference](#).

Batch Operations

If your application needs to read multiple items, you can use the `BatchGetItem` API. A single `BatchGetItem` request can retrieve up to 1 MB of data, which can contain as many as 100 items. In addition, a single `BatchGetItem` request can retrieve items from multiple tables.

The `BatchWriteItem` operation lets you put or delete multiple items. `BatchWriteItem` can write up to 16 MB of data, consisting of up to 25 put or delete requests. The maximum size of an individual item is 400 KB in size. In addition, a single `BatchWriteItem` request can put or delete items in multiple tables. (Note that `BatchWriteItem` cannot update items. To update items, use the `UpdateItem` API.)

A batch consists of one or more requests. For each request, DynamoDB invokes the corresponding API for that request. For example, if a `BatchGetItem` request contains five items, DynamoDB implicitly performs five `GetItem` operations on your behalf. Similarly, if a `BatchWriteItem` request contains two put requests and four delete requests, DynamoDB implicitly performs two `PutItem` and four `DeleteItem` requests.

If an individual request in a batch should fail (for example, because you exceed the provisioned throughput settings on a table), this does not cause the entire batch to fail. Instead, the batch operation returns the keys and data from the individual failed request, so that you can retry the operation. In general, a batch operation does not fail unless *all* of the requests in the batch fail.

For detailed information about batch operations, see [BatchGetItem](#) and [BatchWriteItem](#) in the [Amazon DynamoDB API Reference](#).

Atomic Counters

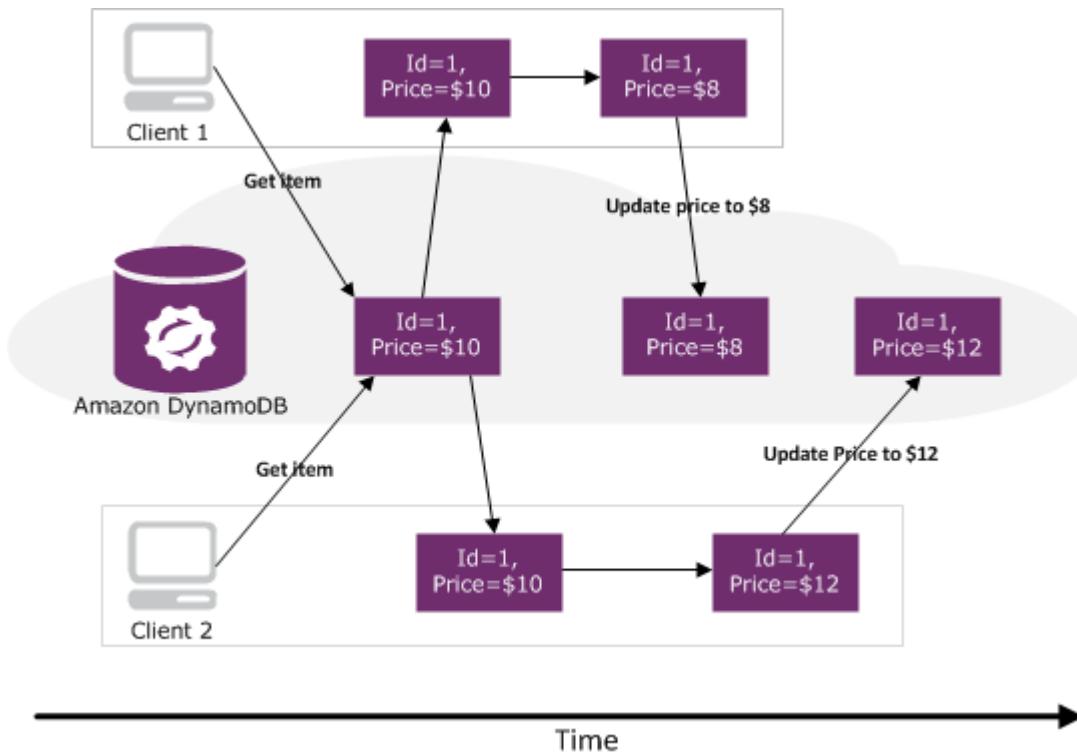
DynamoDB supports *atomic counters*, where you use the `UpdateItem` operation to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they were received.) For example, a web application might want to maintain a counter per visitor to their site. In this case, the application would need to increment this counter regardless of its current value.

Atomic counter updates are *not idempotent*. This means that the counter will increment each time you call `UpdateItem`. If you suspect that a previous request was unsuccessful, your application could retry the `UpdateItem` operation; however, this would risk updating the counter twice. This might be acceptable for a web site counter, because you can tolerate with slightly over- or under-counting the visitors. However, in a banking application, it would be safer to use a conditional update rather than an atomic counter.

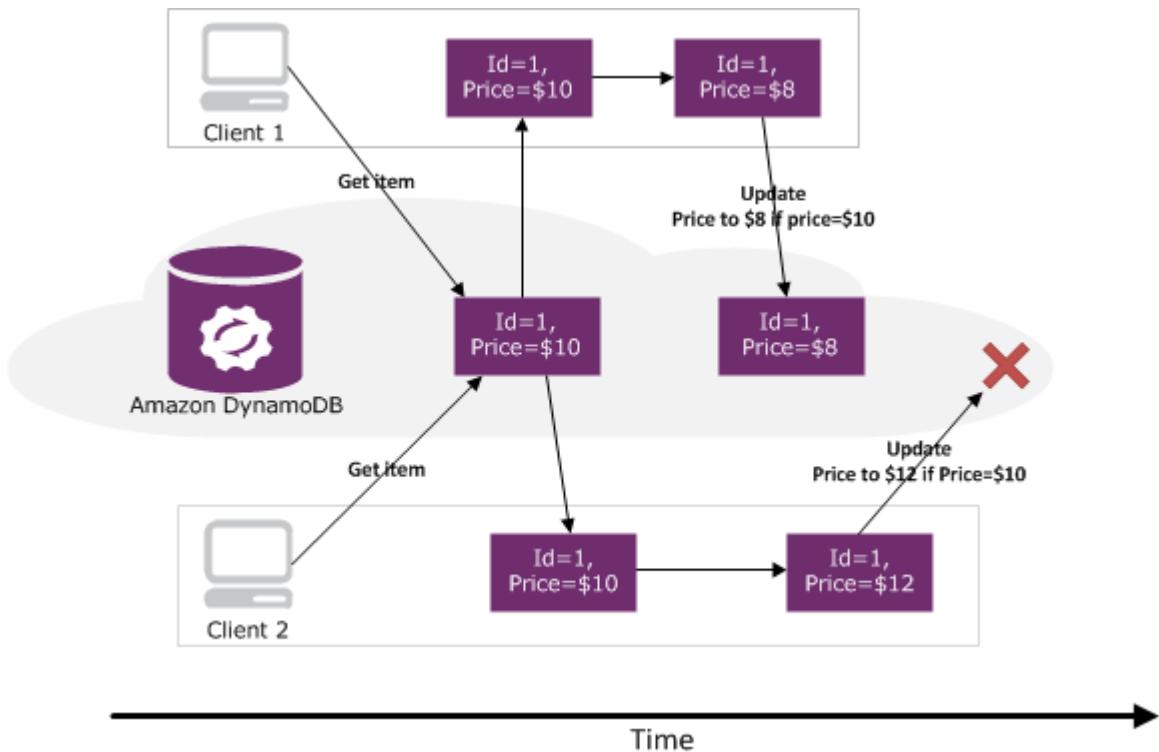
To update an atomic counter, use an `UpdateItem` operation with an attribute of type Number in the `UpdateExpression` parameter, and `SET` as the update action to perform. You can increment the counter using a positive number, or decrement it using a negative number. For more information, see [Incrementing and Decrementing Numeric Attributes \(p. 102\)](#)

Conditional Writes

In a multi-user environment, multiple clients can access the same item and attempt to modify its attribute values at the same time. However, each client might not realize that other clients might have modified the item already. This is shown in the following illustration in which Client 1 and Client 2 have retrieved a copy of an item (Id=1). Client 1 changes the price from \$10 to \$8. Later, Client 2 changes the same item price to \$12, and the previous change made by Client 1 is lost.



To help clients coordinate writes to data items, DynamoDB supports *conditional writes* for `PutItem`, `DeleteItem`, and `UpdateItem` operations. With a conditional write, an operation succeeds only if the item attributes meet one or more expected conditions; otherwise it returns an error. For example, the following illustration shows both Client 1 and Client 2 both retrieving a copy of an item (`Id=1`). Client 1 first attempts to updated the item price to \$8, with the expectation that the existing item price on the server will be \$10. This operation succeeds because the expectation is met. Client 2 then attempts to update price to \$12, with the expectation that the existing item price on the server will be \$10. This expectation cannot be met, because the price is now \$8; therefore, Client 2's request fails.



Note that conditional writes are *idempotent*. This means that you can send the same conditional write request multiple times, but it will have no further effect on the item after the first time DynamoDB performs the specified update. For example, suppose you issue a request to update the price of a book item by 10%, with the expectation that the price is currently \$20. However, before you get a response, a network error occurs and you don't know whether your request was successful or not. Because a conditional update is an idempotent operation, you can send the same request again, and DynamoDB will update the price only if the current price is still \$20.

To request a conditional `PutItem`, `DeleteItem`, or `UpdateItem`, you specify the condition(s) in the `ConditionExpression` parameter. `ConditionExpression` is a string containing attribute names, conditional operators and built-in functions. The entire expression must evaluate to true; otherwise the operation will fail.

Tip

For more information, see [Conditional Write Operations \(p. 105\)](#).

If you specify the `ReturnConsumedCapacity` parameter, DynamoDB will return the number of write capacity units that were consumed during the conditional write. (Note that write operations only consume write capacity units; they never consume read capacity units.) Setting `ReturnConsumedCapacity` to `TOTAL` returns the write capacity consumed for the table and all of its global secondary indexes; `INDEXES` returns only the write capacity consumed by the global secondary indexes; `NONE` means that you do not want any consumed capacity statistics returned.

If a `ConditionExpression` fails during a conditional write, DynamoDB will still consume one write capacity unit from the table. A failed conditional write will return a `ConditionalCheckFailedException` instead of the expected response from the write operation. For this reason, you will not receive any information about the write capacity unit that was consumed. However, you can view the `ConsumedWriteCapacityUnits` metric for the table in Amazon CloudWatch to determine the provisioned write capacity that was consumed from the table. For more information, see [DynamoDB Metrics \(p. 521\)](#) in [Monitoring DynamoDB with CloudWatch \(p. 519\)](#).

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.

Reading and Writing Items Using Expressions

In 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. Some update examples are replacing the attribute with a new value, or adding new data to a list or a map. This section describes the different kinds of expressions that are available.

Note

For backward compatibility, DynamoDB also supports conditional parameters that do not use expressions. For more information, see [Legacy Conditional Parameters \(p. 659\)](#). New applications should use expressions rather than the legacy parameters.

Topics

- [Case Study: A ProductCatalog Item \(p. 91\)](#)
- [Accessing Item Attributes with Projection Expressions \(p. 92\)](#)
- [Using Placeholders for Attribute Names and Values \(p. 94\)](#)
- [Specifying Conditions with Condition Expressions \(p. 96\)](#)
- [Modifying Items and Attributes with Update Expressions \(p. 100\)](#)

Case Study: A *ProductCatalog* Item

In this section, we will consider an item in the *ProductCatalog* table. (This table is shown in [Example Tables and Data](#) in the *Amazon DynamoDB Developer Guide*, along with some sample items.) Here is a representation of the item:

```
{  
    Id: 205,  
    Title: "20-Bicycle 205",  
    Description: "205 description",  
    BicycleType: "Hybrid",  
    Brand: "Brand-Company C",  
    Price: 500,  
    Gender: "B",  
    Color: ["Red", "Black"],  
    ProductCategory: "Bike",  
    InStock: true,  
    QuantityOnHand: null,  
    RelatedItems: [  
        341,  
        472,  
        649  
    ],  
    Pictures: {  
        FrontView: "http://example.com/products/205_front.jpg",  
        RearView: "http://example.com/products/205_rear.jpg",  
        SideView: "http://example.com/products/205_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."  
    ]  
}
```

Note the following:

- The hash key value (`Id`) is 205. There is no range 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.

Accessing Item Attributes with Projection Expressions

To read data from a table, you use API operations such as `GetItem`, `Query` or `Scan`. DynamoDB returns all of the item attributes by default. To get just some of the attributes, rather than all of them, use a projection expression.

Note

The examples in the following sections are based on the *ProductCatalog* item from [Case Study: A ProductCatalog Item \(p. 91\)](#).

Topics

- [Projection Expressions \(p. 92\)](#)
- [Document Paths \(p. 93\)](#)

Projection Expressions

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.

Following are some examples of projection expressions:

- A single top-level attribute.

Title

- Three top-level attributes. Note that DynamoDB will retrieve the entire `Color` set.

Title, Price, Color

- Four top-level attributes. Note that 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 \(p. 94\)](#).

Document Paths

In addition to top-level attributes, expressions can access individual elements in any document type attribute. To do this, you must provide the element's location, or document path, within the item. The *document path* tells DynamoDB where to find the attribute, even if it is deeply nested among multiple lists and maps.

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.

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:

- `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 index in a list dereference 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 Path Examples

Following are some examples of projection expressions using document paths.

- The third element from the `RelatedItems` list. (Remember that list elements are zero-based.)

`RelatedItems[2]`

- The item's price, color, and a front-view picture of the product.

Price, Color, 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 in any 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 \(p. 94\)](#).

Using Placeholders for Attribute Names and Values

Topics

- [Expression Attribute Names \(p. 94\)](#)
- [Expression Attribute Values \(p. 96\)](#)

This section introduces placeholders, or substitution variables, that you can use in DynamoDB expressions.

Expression Attribute Names

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 \(p. 649\)](#).)

For example, the following projection expression would be invalid because SESSION is a reserved word:

- Classroom, Session, StartTime

To work around this, you can define an expression attribute name. An *expression attribute name* is a placeholder that you use in the expression, as an alternative to the actual attribute name. An expression attribute name must begin with a #, followed by one alphabetic character, and then by zero or more alphanumeric characters.

In the preceding expression, you can replace Session with an expression attribute name such as #s. The # (pound sign) is required and indicates that this is a placeholder for an attribute name. The revised expression would now look like this:

- Classroom, #s, StartTime

Tip

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.

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, consider the following item in a DynamoDB table:

```
{  
    Id: 1234,  
    My.Scalar.Message: "Hello",  
    MyMap: {  
        MyKey: "My key value",  
        MyOtherKey: 10"  
    }  
}
```

Suppose that you wanted to access `My.Scalar.Message` using the following projection expression:

```
My.Scalar.Message
```

DynamoDB would return an empty result, rather than the expected `Hello` string. 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 `#msm`) as a substitute for `My.Scalar.Message`. You could then use the following projection expression:

```
#msm
```

DynamoDB would then return the desired result: `Hello`

Tip

A dot in an expression represents a document path separator.

If an attribute name contains dot characters, define an expression attribute name for it. You can then use that name in an expression.

Now suppose that you wanted to access the embedded attribute `MyMap.MyKey`, using the following projection expression:

```
MyMap.MyKey
```

The result would be `My key value`, 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 `#mmmk` as a substitute for `MyMap.MyKey`? DynamoDB would return an empty result, instead of the expected string. 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 `#mmmk`, it determines that `MyMap.MyKey` refers to a scalar attribute—which is not what was intended.

The correct approach would be to define two expression attribute names, one for each element in the document path:

- `#mm` — `MyMap`
- `#mk` — `MyKey`

You could then use the following projection expression:

```
#mm.#mk
```

DynamoDB would then return the desired result: `My key value`

Tip

A dot in an expression attribute name represents a valid character within an attribute's name.

To access a nested attribute, define an expression attribute name for each element in the document path. You can then use these names in an expression, with each name separated by a dot.

Expression attribute names are also 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:

- `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`

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 `:`, followed by one alphabetic character, and then by zero or more alphanumeric characters.

For example, suppose you have an application that shows products costing less than a certain value, with the actual value to be entered by the user. You can define an expression attribute value, such as `:p`, as a placeholder. The `:` is required, and indicates that this is a placeholder for an attribute value. Such an expression would look like this:

- `Price < :p`

At runtime, the application can prompt the user for the desired price. This price will be used in the expression, and DynamoDB will retrieve the desired results.

If you define an expression attribute value, you must use it consistently throughout the entire expression. Also, you cannot omit the `:` symbol.

Specifying Conditions with Condition Expressions

To read items from a table, you use API operations such as `Query` or `Scan`. These API actions let you provide your own conditions for selection criteria and filtering. DynamoDB will evaluate and return only those items that match your conditions.

To write an item, you use API operations such as `PutItem`, `UpdateItem` and `DeleteItem`. These API actions give you control over how and under what conditions an item can be modified.

Note

The examples in the following sections are based on the *ProductCatalog* item from [Case Study: A ProductCatalog Item \(p. 91\)](#).

Topics

- [Condition Expressions \(p. 97\)](#)
- [Condition Expression Reference \(p. 97\)](#)

Condition Expressions

A *condition expression* represents restrictions to put in place when you read and write items in a table. A condition expression is a free-form string that can contain attribute names, document paths, logical operators and functions. For a complete list of elements allowed in a condition expression, see [Condition Expression Reference \(p. 97\)](#)

Following are some examples of condition expressions. Note that some of these expressions use placeholders for attribute names and values. For more information, see [Using Placeholders for Attribute Names and Values \(p. 94\)](#).

- All of the items that have a `RearView` picture.

```
attribute_exists(Pictures.RearView)
```

- Only the items that don't have one-star reviews. The expression attribute name, `#pr`, is a substitute for `ProductReviews`.

```
attribute_not_exists (#pr.FiveStar)
```

For more information about the `#` character, see [Expression Attribute Names \(p. 94\)](#).

- Simple scalar comparisons. `:p` represents a number and `:bt` represents a string.

```
Price <= :p
```

```
BicycleType = :bt
```

- Two conditions that must both be true. `#P` and `#PC` are placeholders for `Price` and `ProductCategory` attribute names. `:lo` and `:hi` represent values of type Number, and `:cat1` and `:cat2` represent values of type String.

```
(#P between :lo and :hi) and (#PC in (:cat1, :cat2))
```

You can use any attribute name in a 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`. 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 \(p. 94\)](#).

Condition Expression Reference

This section covers the fundamental building blocks of condition expressions in DynamoDB.

Note

The syntax for `ConditionExpression` is identical to that of the `FilterExpression` parameter. `FilterExpression` is used for querying and scanning data; for more information, see [Narrowing the Results with Filter Expressions \(p. 184\)](#).

Topics

- [Syntax for Condition Expressions \(p. 98\)](#)
- [Comparators \(p. 98\)](#)
- [Functions \(p. 99\)](#)
- [Logical Evaluations \(p. 99\)](#)
- [Parentheses \(p. 99\)](#)
- [Precedence in Conditions \(p. 99\)](#)

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)
    | begins_with (path, operand)
    | contains (path, operand)

```

Comparators

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`

Range and List Comparisons

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 path; for more information, see [Document Paths \(p. 93\)](#).

- `attribute_exists (path)` — true if the attribute at the specified `path` exists.
- `attribute_not_exists (path)` — true if the attribute at the specified `path` does not exist.
- `begins_with (path, operand)` — true if the attribute at the specified path begins with a particular operand.
- `contains (path, operand)` — true if the attribute at the specified path contains a particular operand. Note that the path and the operand must be distinct; that is, `contains (a, a)` will return an error.

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`

Modifying Items and Attributes with Update Expressions

To delete an item from a table, use the `DeleteItem` operation. You must provide the key of the item you want to delete.

To update an existing item in a table, 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. For more information, see [Update Expressions \(p. 100\)](#).

The `DeleteItem` and `UpdateItem` operations support conditional writes, where you provide a *condition expression* to indicate the conditions that must be met in order for the operation to succeed. For more information, see [Conditional Write Operations \(p. 105\)](#).

If DynamoDB modifies an item successfully, it acknowledges this with an HTTP 200 status code (OK). No further data is returned in the reply; however, you can request that the item or its attributes are returned. You can request these as they appeared before or after an update. For more information, see [Return Values \(p. 105\)](#).

Note

The examples in the following sections are based on the *ProductCatalog* item from [Case Study: A ProductCatalog Item \(p. 91\)](#).

Update Expressions

An *update expression* specifies the attributes you want to modify, along with new values for those attributes. An update expression also specifies *how* to modify the attributes—for example, setting a scalar value, or deleting elements in a list or a map. It is a free-form string that can contain attribute names, document paths, operators and functions. It also contains keywords that indicate how to modify attributes.

The `PutItem`, `UpdateItem` and `DeleteItem` operations require a primary key value, and will only modify the item with that key. If you want to perform a *conditional update*, you must provide an update expression *and* a condition expression. The condition expression specifies the condition(s) that must be met in order for the update to succeed. The following is a syntax summary for update expressions:

```
update-expression ::=  
    SET set-action , ...  
    | REMOVE remove-action , ...  
    | ADD add-action , ...  
    | DELETE delete-action , ...
```

An update expression consists of *sections*. Each section begins with a `SET`, `REMOVE`, `ADD` or `DELETE` keyword. You can include any of these sections in an update expression in any order. However, each section keyword can appear only once. You can modify multiple attributes at the same time. Following are some examples of update expressions:

- `SET list[0] = :val1`
- `REMOVE #m.nestedField1, #m.nestedField2`
- `ADD aNumber :val2, anotherNumber :val3`
- `DELETE aSet :val4`

The following example shows a single update expression with multiple sections:

- SET list[0] = :val1 REMOVE #m.nestedField1, #m.nestedField2 ADD aNumber :val2, anotherNumber :val3 DELETE aSet :val4

You can use any attribute name in an update 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 \(p. 94\)](#).

To specify a literal value in an update expression, you use expression attribute values. For more information, see [Expression Attribute Values \(p. 96\)](#).

Topics

- [SET \(p. 101\)](#)
- [REMOVE \(p. 103\)](#)
- [ADD \(p. 104\)](#)
- [DELETE \(p. 104\)](#)

SET

Use the `SET` action in an update expression to add one or more attributes and values to an item. If any of these attribute already exist, they are replaced by the new values. However, note that you can also use `SET` to add or subtract from an attribute that is of type `Number`. To `SET` multiple attributes, separate them by commas.

In the following syntax summary:

- The `path` element is the document path to the item. For more information, see [Document Paths \(p. 93\)](#).
- An `operand` element can be either a document path to an item, or a function. For more information, see [Functions for Updating Attributes \(p. 102\)](#).

```
set-action ::=  
    path = value  
  
value ::=  
    operand  
    | operand '+' operand  
    | operand '-' operand  
  
operand ::=  
    path | function
```

Following are some examples of update expressions using the `SET` action.

- The following example updates the `Brand` and `Price` attributes. The expression attribute value `:b` is a string and `:p` is a number.

```
SET Brand = :b, Price = :p
```

- The following example updates an attribute in the `RelatedItems` list. The expression attribute value `:ri` is a number.

```
SET RelatedItems[0] = :ri
```

- The following example updates some nested map attributes. The expression attribute name `#pr` is `ProductReviews`; the attribute values `:r1` and `:r2` are strings.

```
SET #pr.FiveStar[0] = :r1, #pr.FiveStar[1] = :r2
```

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.

The following example decreases the `Price` value of an item. The expression attribute value `:p` is a number.

- `SET Price = Price - :p`

To increase the `Price`, use the `+` operator instead.

Using `SET` with List Elements

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 array.

If you add multiple elements in a single `SET` operation, the elements are sorted in order by element number. For example, consider the following list:

```
MyNumbers: { [ "Zero", "One", "Two", "Three", "Four" ] }
```

The list contains elements `[0], [1], [2], [3], [4]`. Now, let's use the `SET` action to add two new elements:

```
set MyNumbers[8]="Eight", MyNumbers[10] = "Ten"
```

The list now contains elements `[0], [1], [2], [3], [4], [5], [6]`, with the following data at each element:

```
MyNumbers: { [ "Zero", "One", "Two", "Three", "Four", "Eight", "Ten" ] }
```

Note

The new elements are added to the end of the list and will be assigned the next available element numbers.

Functions for Updating Attributes

The `SET` action supports the following functions:

- `if_not_exists (path, operand)` – If the item does not contain an attribute at the specified `path`, then `if_not_exists` evaluates to `operand`; otherwise, it evaluates to `path`. You can use this function to avoid overwriting an attribute already present in the item.
- `list_append (operand, operand)` – This function evaluates to a list with a new element added to it. You can append the new element to the start or the end of the list by reversing the order of the operands.

Important

These function names are case-sensitive.

Following are some examples of using the `SET` action with these functions.

- If the attribute already exists, the following example does nothing; otherwise it sets the attribute to a default value.

```
SET Price = if_not_exists(Price, 100)
```

- The following example adds a new element to the *FiveStar* review list. The expression attribute name `#pr` is *ProductReviews*; the attribute value `:r` is a one-element list. If the list previously had two elements, [0] and [1], then the new element will be [2].

```
SET #pr.FiveStar = list_append(#pr.FiveStar, :r)
```

- The following example adds another element to the *FiveStar* review list, but this time the element will be appended to the start of the list at [0]. All of the other elements in the list will be shifted by one.

```
SET #pr.FiveStar = list_append(:r, #pr.FiveStar)
```

REMOVE

Use the `REMOVE` action in an update expression to remove one or more attributes from an item. To perform multiple `REMOVE` operations, 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
```

Following is an example of an update expression using the `REMOVE` action. Several attributes are removed from the item:

- `REMOVE Title, RelatedItems[2], Pictures.RearView`

Using `REMOVE` with List Elements

When you remove an existing list element, the remaining elements are shifted. For example, consider the following list:

- `MyNumbers: { ["Zero", "One", "Two", "Three", "Four"] }`

The list contains elements [0], [1], [2], [3], and [4]. Now, let's use the `REMOVE` action to remove two of the elements:

- `REMOVE MyNumbers[1], MyNumbers[3]`

The remaining elements are shifted to the right, resulting in a list with elements [0], [1], and [2], with the following data at each element:

- `MyNumbers: { ["Zero", "Two", "Four"] }`

Note

If you use `REMOVE` to delete a nonexistent item past the last element of the list, nothing happens: There is no data to be deleted. For example, the following expression has no effect on the `MyNumbers` list:

- `REMOVE MyNumbers[11]`

ADD

Important

The `ADD` action only supports Number and set data types. In general, we recommend using `SET` rather than `ADD`.

Use the `ADD` action in an update expression to do either of the following:

- If the attribute does not already exist, add the new attribute and its value(s) to the 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.

To perform multiple `ADD` operations, 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
```

Following are some examples of update expressions using the `add` action.

- The following example increments a number. The expression attribute value `:n` is a number, and this value will be added to `Price`.

```
ADD Price :n
```

- The following example adds one or more values to the `Color` set. The expression attribute value `:c` is a string set.

```
ADD Color :c
```

DELETE

Important

The `DELETE` action only supports set data types.

Use the `DELETE` action in an update expression to delete an element from a set. To perform multiple `DELETE` operations, 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 `value` element is the element(s) in the set that you want to delete.

```
delete-action ::=  
    path value
```

The following example deletes an element from the Color set using the DELETE action. The expression attribute value :c is a string set.

```
DELETE Color :c
```

Conditional Write Operations

To perform a conditional delete, 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.

Suppose that you want to delete an item, but only if there are no related items. You can use the following expression to do this:

- Condition expression: `attribute_not_exists(RelatedItems)`

To perform a conditional update, use an `UpdateItem` operation with an update expression *and* a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

Suppose that you want to increase the price of an item, but only if the result does not exceed a maximum price. You can use the following expressions to do this:

- Update expression: `SET Price = Price + :amt`
- Condition expression: `Price <= :maxprice`

Now suppose you want to set a front view picture for an item, but only if that item doesn't already have such a picture—you want to avoid overwriting any existing element. You can use the following expressions to do this:

- Update expression: `SET Pictures.FrontView = :myURL`

(Assume that `:myURL` is the location of a picture of the item, such as `http://example.com/picture.jpg`.)

- Condition expression: `attribute_not_exists(Pictures.FrontView)`

Return Values

When you perform a `DeleteItem` or `UpdateItem` operation, DynamoDB can optionally return some or all of the item in the response. To do this, you set the `ReturnValues` parameter. The default value for `ReturnValues` is `NONE`, so no data will be returned. You can change this behavior as described below.

Deleting an Item

In a `DeleteItem` operation, you can set `ReturnValues` to `ALL_OLD`. Doing this will cause DynamoDB to return the entire item, as it appeared before the delete operation occurred.

Updating an Item

In an `UpdateItem` operation, you can set `ReturnValues` to one of the following:

- `ALL_OLD` – The entire item is returned, as it appeared before the update occurred.

- ALL_NEW – The entire item is returned, as it appears after the update.
- UPDATED_OLD – Only the value(s) that you updated are returned, as they appear before the update occurred.
- UPDATED_NEW – Only the value(s) that you updated are returned, as they appear after the update.

Guidelines for Working with Items

Topics

- [Use One-to-Many Tables Instead Of Large Set Attributes \(p. 106\)](#)
- [Use Multiple Tables to Support Varied Access Patterns \(p. 107\)](#)
- [Compress Large Attribute Values \(p. 108\)](#)
- [Store Large Attribute Values in Amazon S3 \(p. 108\)](#)
- [Break Up Large Attributes Across Multiple Items \(p. 109\)](#)

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 \(p. 597\)](#), 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 [Getting Started with DynamoDB \(p. 13\)](#) 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 \(p. 597\)](#))

- 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.

Use Multiple Tables to Support Varied Access Patterns

If you frequently access large items in a DynamoDB table and you do not always use all of an item's larger attributes, you can improve your efficiency and make your workload more uniform by storing your smaller, more frequently accessed attributes as separate items in a separate table.

For example, consider the *ProductCatalog* table described in the [Getting Started with DynamoDB \(p. 13\)](#) section. Items in this table contain basic product information, such as product name and description. This information changes infrequently, but it is used every time an application displays a product for the user.

If your application also needed to keep track of rapidly changing product attributes, such as price and availability, you could store this information in a separate table called *ProductAvailability*. This approach would minimize the throughput cost of updates. To illustrate, suppose that a *ProductCatalog* item was 3 KB in size, and the price and availability attributes for that item were 300 bytes. In this case, an update of these rapidly changing attributes would cost three, the same cost as updating any other product attributes. Now suppose that price and availability information were stored in a *ProductAvailability* table instead. In this case, updating the information would cost only one write capacity unit.

Note

For an explanation of capacity units, see [Provisioned Throughput in Amazon DynamoDB \(p. 10\)](#).

If your application also needed to store product data that is displayed less frequently, you could store this information in a separate table called *ExtendedProductCatalog*. Such data might include product dimensions, a track listing for music albums, or other attributes that are not accessed as often as the basic product data. This way, the application would only consume throughput when displaying basic product information to the users, and would only consume additional throughput if the user requests the extended product details.

The following are example instances of the tables in the preceding discussion. Note that all the tables have an *Id* attribute as the primary key.

ProductCatalog

<u>Id</u>	Title	Description
21	"Famous Book"	"From last year's best sellers list, ..."
302	"Red Bicycle"	"Classic red bicycle..."
58	"Music Album"	"A new popular album from this week..."

ProductAvailability

<u>Id</u>	Price	QuantityOnHand
21	"\$5.00 USD"	3750
302	"\$125.00 USD"	8
58	"\$5.00 USD"	"infinity (digital item)"

ExtendedProductCatalog

<u>Id</u>	<u>AverageCustomerRating</u>	<u>TrackListing</u>
21	5	
302	3.5	
58	4	{"Track1#3:59", "Track2#2:34", "Track3#5:21", ...}

Here are several advantages and considerations for splitting the attributes of an item into multiple items in different tables:

- The throughput cost of reading or writing these attributes is reduced. The cost of updating a single attribute of an item is based on the full size of the item. If the items are smaller, you will incur less throughput when you access each one.
- If you keep your frequently accessed items small, your I/O workload will be distributed more evenly. Retrieving a large item can consume a great deal of read capacity all at once, from the same partition of your table; this can make your workload uneven, which can cause throttling. For more information, see [Avoid Sudden Bursts of Read Activity \(p. 189\)](#)
- For single-item read operations, such as `GetItem`, throughput calculations are rounded up to the next 4 KB boundary. If your items are smaller than 4 KB and you retrieve the items by primary key only, storing item attributes as separate items may not reduce throughput. Even so, the throughput cost of range operations such as `Query` and `Scan` are calculated differently: the sizes of all returned items are totaled, and that final total is rounded up to the next 4 KB boundary. For these operations, you might still reduce your throughput cost by moving the large attributes into separate items. For more information, see [Capacity Units Calculations for Various Operations \(p. 57\)](#).

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 [Getting Started with DynamoDB \(p. 13\)](#) 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 \(p. 129\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API \(p. 162\)](#)

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 \(p. 597\)](#). 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 [Getting Started with DynamoDB \(p. 13\)](#) 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, go to 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 hash schema table, and use batch API calls 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 [Getting Started with DynamoDB \(p. 13\)](#) 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* hash schema table.

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>	<small>-kdc - rev nois</small>
"DynamoDB#Thread1"	"2012-03-15T20:42:54.023Z"		3	1
"DynamoDB#Thread2"	"2012-03-21T20:41:23.192Z"	"short message"		

ReplyChunks

<u>Id</u>	<u>Message</u>
"DynamoDB#Thread1#2012-03-15T20:42:54.023Z#1#1"	"first part of long message text..."
"DynamoDB#Thread1#2012-03-15T20:42:54.023Z#1#3"	"...third part of long message text"

Id	Message
"DynamoDB#Thread1#2012-03-15T20:42:54.023Z#1#2"	"...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 hash key value.

Chunking large data items avoids this problem by using a separate table with a hash 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 hash and range key, with the hash portion 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.

Working with Items Using the AWS SDK for Java Document API

Topics

- [Putting an Item \(p. 110\)](#)
- [Getting an Item \(p. 113\)](#)
- [Batch Write: Putting and Deleting Multiple Items \(p. 115\)](#)
- [Batch Get: Getting Multiple Items \(p. 116\)](#)
- [Updating an Item \(p. 118\)](#)
- [Deleting an Item \(p. 120\)](#)
- [Example: CRUD Operations Using the AWS SDK for Java Document API \(p. 120\)](#)
- [Example: Batch Operations Using AWS SDK for Java Document API \(p. 125\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API \(p. 129\)](#)

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 that the AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

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 \(p. 118\)](#).

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 bicycle item to the `ProductCatalog` table. (This is the same item that is described in [Case Study: A ProductCatalog Item \(p. 91\)](#).)

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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/205_front.jpg");  
pictures.put("RearView", "http://example.com/products/205_rear.jpg");  
pictures.put("SideView", "http://example.com/products/205_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", 205)  
    .withString("Title", "20-Bicycle 205")  
    .withString("Description", "205 description")  
    .withString("BicycleType", "Hybrid")  
    .withString("Brand", "Brand-Company C")  
    .withNumber("Price", 500)  
    .withString("Gender", "B")  
    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red", "Black")))  
  
    .withString("ProductCategory", "Bike")  
    .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.

```
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:

```
{  
    "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 hash primary key.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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 [Data Read and Consistency Considerations \(p. 9\)](#).)

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 \(p. 92\)](#) and [Document Paths \(p. 93\)](#).

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 low-level `GetItem` API inputs. The code example below creates a `GetItemSpec`, and uses that spec as input to the `getItem` method.

```
GetItemSpec spec = new GetItemSpec()  
    .withPrimaryKey("Id", 205)  
    .withProjectionExpression("Id, Title, RelatedItems[0], Reviews.FiveStar")  
    .withConsistentRead(true);  
  
Item item = table.getItem(spec);  
  
System.out.println(item.toJSONPretty());
```

Tip

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" : 205,  
    "Title" : "20-Bicycle 205"  
}
```

GetItem and JSON Documents

In the [PutItem and JSON Documents \(p. 112\)](#) 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"]}}}
```

Tip

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 API 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 \(p. 597\)](#).

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.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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 hash attribute value",  
                "Amazon S3", "Some range attribute 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 \(p. 126\)](#).

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.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
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");  
  
Map<String, TableKeysAndAttributes> requestItems = new HashMap<String,  
TableKeysAndAttributes>();  
requestItems.put(forumTableName, forumTableKeysAndAttributes);  
requestItems.put(threadTableName, threadTableKeysAndAttributes);  
  
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.

```
TableKeysAndAttributes forumTableKeysAndAttributes = new TableKeysAndAttrib
```

```
utes("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 `Authors` multi-valued attribute 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`.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

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.

```

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.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

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`).

```
Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", false);

DeleteItemOutcome outcome = table.deleteItem("Id", 103,
    String conditionExpression,
    null, // ExpressionAttributeNames - not used in this example
    expressionAttributeValues);
```

Example: CRUD Operations Using the AWS SDK for Java Document API

The following AWS SDK for Java Document API code example illustrates CRUD operations on an item. The example creates an item, retrieves it, performs various updates, and finally deletes the item.

Note

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB.

tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

For a code example that demonstrates CRUD operations using the object persistence model, see [Example: CRUD Operations \(p. 395\)](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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.model.ReturnValue;

public class DocumentAPIItemCRUDExample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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 {

    Map<String, String> expressionAttributeNames = new HashMap<String,
String>();
    expressionAttributeNames.put("#na", "NewAttribute");

    Map<String, Object> expressionAttributeValues = new HashMap<String,
Object>();
    expressionAttributeValues.put(":val1", "Some value");

    UpdateItemSpec updateItemSpec = new UpdateItemSpec()
        .withPrimaryKey("Id", 121)
        .withUpdateExpression("set #na = :val1")
        .withNameMap(expressionAttributeNames)
        .WithValueMap(expressionAttributeValues)
        .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 {

        Map<String, String> expressionAttributeNames = new HashMap<String,
String>();
        expressionAttributeNames.put("#a", "Authors");
        expressionAttributeNames.put("#na", "NewAttribute");

        Map<String, Object> expressionAttributeValues = new HashMap<String,
Object>();
        expressionAttributeValues
            .put(":val1",
                new HashSet<String>(Arrays.asList("Author YY",
                    "Author ZZ")));
        expressionAttributeValues.put(":val2", "someValue");

        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey("Id", 120)
            .withUpdateExpression(
                "add #a :val1 set #na=:val2")
            .withNameMap(expressionAttributeNames)
            .WithValueMap(expressionAttributeValues)
            .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 {

        Map<String, String> expressionAttributeNames = new HashMap<String,
String>();
        expressionAttributeNames.put("#p", "Price");

        // Specify the desired price (25.00) and also the condition (price
= //
        // 20.00)

        Map<String, Object> expressionAttributeValues = new HashMap<String,
Object>();
        expressionAttributeValues.put(":val1", 25);
        expressionAttributeValues.put(":val2", 20);

        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey("Id", 120)
            .withReturnValues(ReturnValue.ALL_NEW)
            .withUpdateExpression("set #p = :val1")
            .withConditionExpression("#p = :val2")
            .withNameMap(expressionAttributeNames)
            .withValueMap(expressionAttributeValues);

        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out
            .println("Printing item after conditional update to new attrib
ute...");
        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 {

    Map<String, String> expressionAttributeNames = new HashMap<String,
String>();
    expressionAttributeNames.put("#ip", "InPublication");

    Map<String, Object> expressionAttributeValues = new HashMap<String,
Object>();
    expressionAttributeValues.put(":val", false);

    DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
        .withPrimaryKey("Id", 120)
        .withConditionExpression("#ip = :val")
        .withNameMap(expressionAttributeNames)
        .withValueMap(expressionAttributeValues)
        .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 \(p. 126\)](#)
- [Example: Batch Get Operation Using the AWS SDK for Java Document API \(p. 127\)](#)

This section provides examples of batch write and batch get operations using the AWS SDK for Java Document API.

Note

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

For a code example that demonstrates batch write operations using the object persistence model, see [Example: Batch Write Operations \(p. 397\)](#).

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, the DynamoDB `batchWriteItem` API 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 [Getting Started with DynamoDB \(p. 13\)](#) 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 \(p. 614\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for Java \(p. 365\)](#).

```
package com.amazonaws.codesamples;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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(forumT
ableName) //Forum
            .withItemsToPut(new Item()
                .withPrimaryKey( "Name" , "Amazon RDS" )
                .withNumber("Threads" , 0));

        // Add a new item, and delete an existing item, from Thread
        TableWriteItems threadTableWriteItems = new Table
        WriteItems(threadTableName)
            .withItemsToPut(new Item()
                .withPrimaryKey("ForumName" , "Amazon RDS" , "Subject" , "Amazon RDS
Thread 1")
                    .withString("Message" , "ElasticCache 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(forumTable
WriteItems , threadTableWriteItems);

        do {

            // Check for unprocessed keys which could happen if you exceed
            provisioned throughput

            Map<String, List<WriteRequest>> unprocessedItems = outcome.ge
tUnprocessedItems();

            if (outcome.getUnprocessedItems().size() == 0) {
                System.out.println("No unprocessed items found");
            } else {
                System.out.println("Retrieving the unprocessed items");
                outcome = dynamoDB.batchWriteItemUnprocessed(unpro
cessedItems);
            }

            } 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 item key list for each item to get. The example processes the response by printing the items retrieved.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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 TableKey
sAndAttributes(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");

            Map<String, TableKeysAndAttributes> requestItems = new
HashMap<String, TableKeysAndAttributes>();
            requestItems.put(forumTableName, forumTableKeysAndAttributes);
            requestItems.put(threadTableName, threadTableKeysAndAttributes);
        }
    }
}
```

```
System.out.println("Making the request.");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(forumTableKey
sAndAttributes,
                           threadTableKeysAndAttributes);

do {

    for (String tableName : outcome.getTableItems().keySet()) {
        System.out.println("Items in table " + tableName);
        List<Item> items = outcome.getTableItems().get(tableName);

        for (Item item : items) {
            System.out.println(item.toJSONString());
        }
    }

    // Check for unprocessed keys which could happen if you exceed
provisioned
    // throughput or reach the limit on response size.

    Map<String, KeysAndAttributes> unprocessedKeys = outcome.getUn
processedKeys();

    if (outcome.getUnprocessedKeys().size() == 0) {
        System.out.println("No unprocessed keys found");
    } else {
        System.out.println("Retrieving the unprocessed keys");
        outcome = dynamoDB.batchGetItemUnprocessed(unprocessedKeys);
    }
}

} while (outcome.getUnprocessedKeys().size() > 0);

} 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 AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

If you followed the [Getting Started with DynamoDB \(p. 13\)](#) 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 \(p. 614\)](#).

For step-by-step instructions to test the following sample, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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("Ex
tendedMessage")));

    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.finish();
    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);
    }

    return new String(baos.toByteArray(), "UTF-8");
}
```

Working with Items Using the AWS SDK for .NET Low-Level API

Topics

- [Putting an Item \(p. 133\)](#)
- [Getting an Item \(p. 135\)](#)
- [Updating an Item \(p. 136\)](#)
- [Atomic Counter \(p. 138\)](#)
- [Deleting an Item \(p. 138\)](#)

- [Batch Write: Putting and Deleting Multiple Items \(p. 139\)](#)
- [Batch Get: Getting Multiple Items \(p. 141\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Low-Level API \(p. 144\)](#)
- [Example: Batch Operations Using AWS SDK for .NET Low-Level API \(p. 151\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API \(p. 162\)](#)

You can use the AWS SDK for .NET low-level API (protocol-level API) to perform typical create, read, update, and delete (CRUD) operations on an item in a table. The low-level API for item operations map to the corresponding DynamoDB API (see [Using the DynamoDB API \(p. 477\)](#)).

Note that the .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. The .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables. These higher level APIs can reduce the amount of code you have to write.

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 \(p. 136\)](#).

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.

```
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 multi-valued string attribute.

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.

```

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 about the parameters and the API, 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 \(p. 141\)](#).

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 [Data Read and Consistency Considerations \(p. 9\)](#).

```
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 about the parameters and the API, 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` API uses the following guidelines:

- If the item does not exist, the `UpdateItem` API adds a new item using the primary key that is specified in the input.
- If the item exists, the `UpdateItem` API 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 \(p. 133\)](#). 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, the `UpdateItem` API 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.

```
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 about the parameters and the API, 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.

```
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.

```
var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue
    { N = "201" } } },

    // Optional parameters.
    ReturnValues = "ALL_OLD",
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#IP", "InPublication"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {"":inpub",new AttributeValue {BOOL = false}}
    },
    ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);
```

For more information about the parameters and the API, 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 API 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 \(p. 151\)](#).

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" } }
                    },
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "Amazon S3" } }
                    }
                }
            }
        },
        { table2Name,
            new KeysAndAttributes
            {
                Keys = new List<Dictionary<string, AttributeValue>>()
                {
                    new Dictionary<string, AttributeValue>()
                    {
                        { "ForumName", new AttributeValue { S = "DynamoDB" } },
                        { "Subject", new AttributeValue { S = "DynamoDB Thread 1" } }
                    },
                    new Dictionary<string, AttributeValue>()
                    {
                        { "ForumName", new AttributeValue { S = "DynamoDB" } },
                        { "Subject", new AttributeValue { S = "DynamoDB Thread 2" } }
                    },
                    new Dictionary<string, AttributeValue>()
                    {
                        { "ForumName", new AttributeValue { S = "Amazon S3" } },
                        { "Subject", new AttributeValue { S = "Amazon S3 Thread 1" } }
                    }
                }
            }
        }
    };
}

var response = client.BatchGetItem(request);

// Check the response.
var result = response.BatchGetItemResult;
var responses = result.Responses; // The attribute list in the response.

var table1Results = responses[table1Name];
Console.WriteLine("Items in table {0}" + table1Name);
foreach (var item1 in table1Results.Items)
{

```

```

        PrintItem(item1);
    }

var table2Results = responses[table2Name];
Console.WriteLine("Items in table {1}" + table2Name);
foreach (var item2 in table2Results.Items)
{
    PrintItem(item2);
}
// Any unprocessed keys? could happen if you exceed ProvisionedThroughput or
// some other error.
Dictionary<string, KeysAndAttributes> unprocessedKeys = result.UnprocessedKeys;
foreach (KeyValuePair<string, KeysAndAttributes> pair in unprocessedKeys)
{
    Console.WriteLine(pair.Key, pair.Value);
}

```

Specifying Optional Parameters

You can also provide optional parameters using the `BatchGetItemRequest` object as shown in the following C# code 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.

```

AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string tableName = "Forum";

var request = new BatchGetItemRequest
{
    RequestItems = new Dictionary<string, KeysAndAttributes>()
    {
        { tableName,
            new KeysAndAttributes
            {
                Keys = new List<Dictionary<string, AttributeValue>>()
                {
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "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 about the parameters and the API, 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 Getting Started 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 \(p. 623\)](#).

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write.

For a code example that demonstrates CRUD operations using the document model classes, see [Example: CRUD Operations Using the AWS SDK for .NET Document Model \(p. 419\)](#). For a code example that demonstrates CRUD operations using the object persistence model, see [Example: CRUD Operations Using the AWS SDK for .NET Object Persistence Model \(p. 461\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples

{
    class LowLevelItemCRUDExample
    {
        private static string tableName = "ProductCatalog";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
    }
```

```
try

{
    CreateItem();

    RetrieveItem();

    // Perform various updates.

    UpdateMultipleAttributes();
    UpdateExistingAttributeConditionally();

    // Delete item.

    DeleteItem();

    Console.WriteLine("To continue, press Enter");

    Console.ReadLine();
}

catch (Exception e)
{
    Console.WriteLine(e.Message);

    Console.WriteLine("To continue, press Enter");

    Console.ReadLine();
}

}

private static void CreateItem()
{
    var request = new PutItemRequest
    {
        TableName = tableName,
        Item = new Dictionary<string, AttributeValue>()
```

```
{  
    "Id", new AttributeValue { N = "1000" }},  
    "Title", new AttributeValue { S = "Book 201 Title" }},  
    "ISBN", new AttributeValue { S = "11-11-11-11" }},  
    "Authors", new AttributeValue { SS = new List<string>{ "Author1",  
"Author2" }}},  
    "Price", new AttributeValue { N = "20.00" }},  
    "Dimensions", new AttributeValue { S = "8.5x11.0x.75" }},  
    "InPublication", new AttributeValue { BOOL = false } }  
}  
};  
client.PutItem(request);  
}  
  
private static void RetrieveItem()  
{  
    var request = new GetItemRequest  
    {  
        TableName = tableName,  
        Key = new Dictionary<string, AttributeValue>()  
    {  
        "Id", new AttributeValue { N = "1000" } }  
    },  
        ProjectionExpression = "Id, ISBN, Title, Authors",  
        ConsistentRead = true  
    };  
    var response = client.GetItem(request);  
  
    // Check the response.  
}
```

```
var attributeList = response.Item; // attribute list in the response.

Console.WriteLine("\nPrinting item after retrieving it
....");

PrintItem(attributeList);

}

private static void UpdateMultipleAttributes()
{
    var request = new UpdateItemRequest
    {
        Key = new Dictionary<string, AttributeValue>()
        {
            { "Id", new AttributeValue { N = "1000" } }
        },
        // Perform the following updates:
        // 1) Add two new authors to the list
        // 1) Set a new attribute
        // 2) Remove the ISBN attribute
        ExpressionAttributeNames = new Dictionary<string,string>()
        {
            { "#A", "Authors" },
            { "#NA", "NewAttribute" },
            { "#I", "ISBN" }
        },
        ExpressionAttributeValues = new Dictionary<string, Attribute
Value>()
        {
            { ":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 con-
ditionally .....");

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. 152)
- Example: Batch Get Operation Using the AWS SDK for .NET Low-Level API (p. 157)

This section provides examples of batch operations, batch write and batch get, that DynamoDB supports.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables.

For code examples that demonstrate batch operations using the object persistence model, see [Batch Operations Using AWS SDK for .NET Object Persistence Model \(p. 457\)](#) and [Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model \(p. 464\)](#).

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, the DynamoDB `BatchWriteItem` API 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 Getting Started, 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 \(p. 623\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchWrite
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
    }
}
```

```
static void Main(string[] args)
{
    try
    {
        TestBatchWrite();
    }
    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }

    catch (Exception e) { Console.WriteLine(e.Message); }

    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}

private static void TestBatchWrite()
{
    var request = new BatchWriteItemRequest
    {
        ReturnConsumedCapacity = "TOTAL",
        RequestItems = new Dictionary<string, List<WriteRequest>>
    };

    {
        tableName, new List<WriteRequest>
    {

        new WriteRequest
        {

            PutRequest = new PutRequest
            {

```

```
        Item = new Dictionary<string, AttributeValue>

        {
            { "Name", new AttributeValue { S = "S3 forum" } },
            { "Threads", new AttributeValue { N = "0" } }
        }

    }

}

}

}

,

{

    tableName, 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 hash attr
value" } },
                { "Subject", new AttributeValue { S = "Some range attr
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.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables. The individual object instances then map to items in a table.

For code examples that demonstrate batch operations using the object persistence model, see [Batch Operations Using AWS SDK for .NET Object Persistence Model \(p. 457\)](#) and [Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model \(p. 464\)](#).

If you followed the Getting Started 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 \(p. 623\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchGet
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
```

```
private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
  
static void Main(string[] args)  
{  
    try  
    {  
        RetrieveMultipleItemsBatchGet();  
  
        Console.WriteLine("To continue, press Enter");  
        Console.ReadLine();  
    }  
    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }  
  
    catch (Exception e) { Console.WriteLine(e.Message); }  
}  
  
private static void RetrieveMultipleItemsBatchGet()  
{  
    var request = new BatchGetItemRequest  
    {  
        RequestItems = new Dictionary<string, KeysAndAttributes>()  
    };  
  
    { tableName,  
        new KeysAndAttributes  
        {  
            Keys = new List<Dictionary<string, AttributeValue>>()  
            {  
                new Dictionary<string, AttributeValue>()  
                {  
                    {  
                        "attributeName": "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}", ta
bleResponse.Key);

        foreach (var item1 in tableResults)

        {

            PrintItem(item1);

        }

    }

}
```

```
// Any unprocessed keys? could happen if you exceed Provisioned  
Throughput or some other error.  
  
Dictionary<string, KeysAndAttributes> unprocessedKeys = re  
sponse.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> attrib  
uteList)  
{  
  
    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()) + " ]")  
  
    );  
  
}  
  
Con  
sole.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.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables.

If you followed the Getting Started, 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 \(p. 623\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
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 replyIdHashAttribute = "Amazon DynamoDB#DynamoDB Thread 1";

            string replyDateTimeRangeAttribute = Convert.ToString(DateTime.UtcNow);

            try
            {
                CreateItem(replyIdHashAttribute, replyDateTimeRangeAttribute);

                RetrieveItem(replyIdHashAttribute, replyDateTimeRangeAttribute);

                // Delete item.

                DeleteItem(replyIdHashAttribute, replyDateTimeRangeAttribute);

                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 hashAttribute, string rangeAttribute)
    {
        MemoryStream compressedMessage = ToGzipMemoryStream("Some long ex-
tended message to compress.");

        var request = new PutItemRequest
        {
            TableName = tableName,
            Item = new Dictionary<string, AttributeValue>()
        {

            { "Id", new AttributeValue { S = hashAttribute } },
            { "ReplyDateTime", new AttributeValue { S = rangeAttribute } },
            { "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 hashAttribute, string rangeAttribute)
{
    var request = new GetItemRequest
    {
        TableName = tableName,
```

```
        Key = new Dictionary<string, AttributeValue>()

    {
        { "Id", new AttributeValue { S = hashAttribute } },
        { "ReplyDateTime", new AttributeValue { S = rangeAttribute } }
    },
    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 hashAttribute, string rangeAttribute)
{
    var request = new DeleteItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue { S = hashAttribute } },
        { "ReplyDateTime", new AttributeValue { S = rangeAttribute } }
    },
    };
    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, Compression
Mode.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, Compression
Mode.Decompress))

        using (StreamReader reader = new StreamReader(zipStream))
    {
        return reader.ReadToEnd();
    }
}

}
```

Working with Items Using the AWS SDK for PHP Low-Level API

Topics

- [Putting an Item \(p. 168\)](#)
- [Getting an Item \(p. 170\)](#)
- [Batch Write: Putting and Deleting Multiple Items \(p. 171\)](#)
- [Batch Get: Getting Multiple Items \(p. 172\)](#)
- [Updating an Item \(p. 174\)](#)
- [Atomic Counter \(p. 176\)](#)
- [Deleting an Item \(p. 176\)](#)
- [Example: CRUD Operations Using the AWS SDK for PHP Low-Level API \(p. 178\)](#)
- [Example: Batch Operations Using AWS SDK for PHP \(p. 180\)](#)

You can use AWS SDK for PHP API to perform typical create, read, update, and delete (CRUD) operations on an item in a table. The PHP API for item operations map to the underlying the DynamoDB API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

The following are the common steps that you follow to perform data CRUD operations using the PHP API.

1. Create an instance of the `DynamoDbClient` client.
2. Provide the parameters for a DynamoDB operation to the client instance, including any optional parameters.
3. Load the response from DynamoDB into a local variable for your application.

Putting an Item

The PHP `putItem` function uploads an item to 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` function. For more information, see [Updating an Item \(p. 174\)](#).

The following are the steps to upload an item to DynamoDB using the AWS SDK for PHP.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `putItem` operation to the client instance.

You must provide the table name and the item attributes, including primary key values.

3. Load the response into a local variable, such as `$response` to use in your application.

The following PHP code snippet demonstrates the preceding tasks. The code uploads an item to the `ProductCatalog` table.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->putItem(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'      => array('N'      => 104      ), // Primary Key
        'Title'   => array('S'      => 'Book 104 Title' ),
        'ISBN'    => array('S'      => '111-1111111111' ),
        'Price'   => array('N'      => 25  ),
        'Authors' => array('SS'    => array('Author1', 'Author2') )
    )
));
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters to the `putItem` function. For example, the following PHP 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 PHP SDK throws an `ConditionalCheckFailedException`. The code specifies the following optional parameters in for `putItem`:

- A `ConditionExpression` parameter that define conditions for the request, such as the condition that the existing item is replaced only if it has an ISBN attribute that equals a specific value.
- The `ALL_OLD` value for the `ReturnValue` parameter that provides all the attribute values for the item before the PutItem operation. In this case, the older item only had two authors and the new item values include three authors.

```
$client = DynamoDbClient::factory(array(
    "profile" => "default",
    "region" => "us-west-2" #replace with your desired region
));

$tableName = "ProductCatalog";

$result = $client->putItem ( array (
    "TableName" => $tableName,
    "Item" => array (
        "Id" => array (
            "N" => 104
        ), // Primary Key
        "Title" => array (
            "S" => "Book 104 Title"
        ),
        "ISBN" => array (
            "S" => "333-3333333333"
        ),
        "Price" => array (
            "N" => 2000
        ),
        "Authors" => array (
            "SS" => array (
                "Author1",
                "Author2",
                "Author3"
            )
        )
    ),
    "ExpressionAttributeNames" => array (
        "#I" => "ISBN" ) ,
    "ExpressionAttributeValues" => array (
        ":vall" => array("S" => "333-3333333333") ) ,
    "ConditionExpression" => "#I = :vall",
    "ReturnValues" => "ALL_OLD"
) ) ;

print_r ($result);
```

For more information, see [PutItem](#).

Getting an Item

The `getItem` function retrieves a single item. To retrieve multiple items, you can use the `batchGetItem` method (see [Batch Get: Getting Multiple Items \(p. 172\)](#)).

The following are the steps to retrieve an item.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `getItem` operation to the client instance.

You must provide the table name and primary key values.

3. Load the response into a local variable, such as `$response` to use in your application.

The following PHP code snippet demonstrates the preceding steps. The code gets the item that has the specified hash primary key.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
);

$response = $client->getItem(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'Id' => array( 'N' => 104 )
    )
));
print_r ($response['Item']);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `getItem` function. For example, the following PHP code snippet uses an optional method to retrieve only a specific list of attributes, and requests a strongly consistent return value. The code specifies the following optional parameters:

- A specific list of attribute names, including the `Id` and `Authors`.
- A Boolean value that requests a strongly consistent read value. Read results are eventually consistent by default. You can request read results to be strongly consistent. To learn more about read consistency, see [Data Read and Consistency Considerations \(p. 9\)](#).

```
$client = DynamoDbClient::factory(array
```

```
        'profile' => 'default',
        'region' => 'us-west-2' #replace with your desired region
    ));

$response = $client->getItem(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'Id' => array('N' => 104),
    ),
    'ProjectionExpression' => 'Id, Authors',
    'ConsistentRead' => true
));

print_r ($response['Item']);
```

For more information about the parameters and the API, see [GetItem](#).

Batch Write: Putting and Deleting Multiple Items

The AWS SDK for PHP `batchWriteItem` function enables you to put or delete several items from multiple tables in a single request.

The following are the common steps that you follow to get multiple items.

1. Create an instance of the `DynamoDbClient` class.
2. Execute the `batchWriteItem` operation by providing the associative array parameter with the list of put and write requests.

The following PHP code snippet demonstrates the preceding steps. The code performs the following write operations:

- Put an item in the Forum table.
- Put and delete an item from the Thread table.

Note that the `key:value` pair specified in the array parameter to the `batchWriteItem` uses syntax required by the underlying DynamoDB API. For more information, see [BatchWriteItem](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableNameOne = "Forum";
$tableNameTwo = "Thread";

$response = $client->batchWriteItem(array(
    "RequestItems" => array(
```

```

$tableNameOne => array(
    array(
        "PutRequest" => array(
            "Item" => array(
                "Name" => array('S' => "Amazon S3 Forum"),
                "Threads" => array('N' => 0)
            )
        )
    ),
    $tableNameTwo => array(
        array(
            "PutRequest" => array(
                "Item" => array(
                    "ForumName" => array('S' => "Amazon S3 Forum"),
                    "Subject" => array('S' => "My sample question"),
                    "Message"=> array('S' => "Message Text."),
                    "KeywordTags"=>array('SS' => array("Amazon S3", "Bucket"))
                )
            )
        )
    )
));

```

Batch Get: Getting Multiple Items

The AWS SDK for PHP `batchGetItem` function enables you to retrieve multiple items from one or more tables. To retrieve a single item, you can use the `getItem` method.

The following are the common steps that you follow to get multiple items.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `batchGetItem` operation to the client instance as `RequestItems`.

You must provide the table names and primary key values.

3. Load the response into a local variable, such as `$response` to use in your application.

The following PHP code snippet demonstrates the preceding steps. The code retrieves two items from the Forum table and three items from the Thread table.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

date_default_timezone_set("UTC");
$sevenDaysAgo = date("Y-m-d H:i:s", strtotime("-7 days"));
$twentyOneDaysAgo = date("Y-m-d H:i:s", strtotime("-21 days"));

$response = $client->batchGetItem(array(
    "RequestItems" => array(
        "Forum" => array(
            "Keys" => array(
                array( // Key #2
                    "Name" => array( 'S' => "DynamoDB" )
                )
            )
        ),
        "Reply" => array(
            "Keys" => array(
                array( // Key #1
                    "Id" => array( 'S' => "DynamoDB#DynamoDB Thread 2" ),
                    "ReplyDateTime" => array( 'S' => $sevenDaysAgo ),
                ),
                array( // Key #2
                    "Id" => array( 'S' => "DynamoDB#DynamoDB Thread 2" ),
                    "ReplyDateTime" => array( 'S' => $twentyOneDaysAgo ),
                )
            )
        )
    )
));
print_r($response[ 'Responses' ]);

```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `batchGetItem` function. For example, you can specify a list of attributes to retrieve as shown in the following PHP code snippet. The code retrieves two items from the `Forum` table and uses the `ProjectionExpression` parameter to retrieve the count of threads in each table:

```

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->batchGetItem(array(
    "RequestItems" => array(
        "Forum" => array(
            "Keys" => array(
                array( // Key #1
                    "Name" => array( 'S' => "Amazon S3" )
                )
            )
        )
    )
));

```

```
        array( // Key #2
            "Name" => array( 'S' => "DynamoDB" )
        )
    ),
    "ProjectionExpression" => "Threads"
),
)
);
print_r($response);
```

For more information about the parameters and the API, see [BatchGetItem](#).

Updating an Item

Use the `updateItem` function to update existing attribute values, add new attributes to the existing collection, or delete attributes from the existing collection.

The `updateItem` function uses the following guidelines:

- If an item does not exist, the `updateItem` function adds a new item using the primary key that is specified in the input.
- If an item exists, the `updateItem` function applies the updates as follows:
 - Replaces the existing attribute values with the values in the update.
 - If the attribute you provide in the input does not exist, it adds a new attribute to the item.
 - 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` function ([Putting an Item \(p. 168\)](#)) also updates items. For example, if you use `putItem` to upload an item and the primary key exists, the operation replaces the entire item. If there are attributes in the existing item and those attributes are not specified in the input, the `putItem` operation deletes those attributes. However, the `updateItem` API only updates the specified input attributes so that any other existing attributes of that item remain unchanged.

The following are the steps to update an existing item using the AWS SDK for PHP.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `updateItem` operation to the client instance as an `UpdateExpression`.

You must provide the table name, primary key, and attribute names and values to update.

3. Load the response into a local variable, such as `$response` to use in your application.

The following PHP code snippet demonstrates the preceding tasks. The example updates a book item in the `ProductCatalog` table. It adds a new author to the `Authors` multi-valued attribute and deletes the existing `ISBN` attribute. It also reduces the price by one.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->updateItem(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'Id' => array(
            'N' => 201
        )
    ),
    'ExpressionAttributeValues' => array (
        ':val1' => array(
            'S' => 'Author YY',
            'S' => 'Author ZZ'),
        ':val2' => array('N' => '1')
    ),
    'UpdateExpression' => 'set Authors = :val1, Price = Price - :val2 remove
ISBN'
));
print_r($response);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `updateItem` function including an expected value that an attribute must have if the update is to occur. If the condition you specify is not met, then the AWS SDK for PHP throws a `ConditionalCheckFailedException`. For example, the following PHP code snippet conditionally updates a book item price to 25. It specifies the following optional parameters:

- A `ConditionExpression` parameter that sets the condition that the price should be updated only if the existing price is 20.00.
- A `ALL_NEW` value for the `ReturnValues` parameter that specifies the response should include all of the item's current attribute values after the update.

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->updateItem(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'Id' => array(
            'N' => 201
        )
    ),
    'ExpressionAttributeValues' => array (
        ':val1' => array('N' => 22),
        ':val2' => array('N' => 20)
    ),
    'UpdateExpression' => 'set Price = :val1 + :val2 if Price < :val1'
));
print_r($response);
```

```
'UpdateExpression' => 'set Price = :val1',
'ConditionExpression' => 'Price = :val2',
'ReturnValues' => 'ALL_NEW'
));
print_r($response);
```

For more information about the parameters and the API, 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 appropriate `UpdateExpression`.

The following code snippet demonstrates this, incrementing the `Quantity` attribute by one.

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->updateItem(array(
    "TableName" => "ProductCatalog",
    "Key" => array(
        "Id" => array(
            'N' => 201
        )
    ),
    "ExpressionAttributeValues" => array (
        ":val1" => array('N' => '1')
    ),
    "UpdateExpression" => "set Quantity = Quantity + :val1",
    "ReturnValues" => 'ALL_NEW'
));
print_r($response["Attributes"]);
```

Deleting an Item

The `deleteItem` function deletes an item from a table.

The following are the common steps that you follow to delete an item using the AWS SDK for PHP.

1. Create an instance of the `DynamoDbClient` class (the client).
2. Provide the parameters for the `deleteItem` operation to the client instance.

You must provide the table name and primary key values.

3. Load the response into a local variable, such as `$response` to use in your application.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively,

you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->deleteItem(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'Id' => array(
            'N' => 101
        )
    )
));
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `deleteItem` function. For example, the following PHP code snippet specifies the following optional parameters:

- An `Expected` parameter specifying that the Book item with Id value "103" in the ProductCatalog table be deleted only if the book is no longer in publication. Specifically, delete the book if the `InPublication` attribute is false.
- A `RETURN_ALL_OLD` enumeration value for the `ReturnValues` parameter requests that the response include the item that was deleted and its attributes before the deletion.

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableName = "ProductCatalog";

$response = $client->deleteItem ( array (
    'TableName' => $tableName,
    'Key' => array (
        'Id' => array (
            'N' => 103
        )
    ),
    'ExpressionAttributeValues' => array(
        ':val1' => array('BOOL' => false)
    ),
    'ConditionExpression' => 'InPublication = :val1',
    'ReturnValues' => 'ALL_OLD'
) );
```

For more information about the parameters and the API, see [DeleteItem](#).

Example: CRUD Operations Using the AWS SDK for PHP Low-Level API

The following PHP code example illustrates CRUD (create, read, update, and delete) operations on an item. The example creates an item, retrieves it, performs various updates, and finally deletes the item. However, the delete operation is commented-out so you can keep the data until you are ready to delete it.

Note

For step-by-step instructions to test the following code example, see [Running PHP Examples \(p. 371\)](#).

```
<?php

use Aws\AwsClient;
use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    "profile" => "default",
    "region" => "us-west-2" // replace with your desired region
));

$tableName = "ProductCatalog";

// ##### Adding data to the table

echo "# Adding data to table $tableName..." . PHP_EOL;

$response = $client->putItem(array(
    "TableName" => $tableName,
    "Item" => array(
        "Id" => array("N" => "120"),
        "Title" => array("S" => "Book 120 Title"),
        "ISBN" => array("S" => "120-1111111111"),
        "Authors" => array("SS" => array("Author12", "Author22")),
        "Price" => array("N" => "20"),
        "Category" => array("S" => "Book"),
        "Dimensions" => array("S" => "8.5x11.0x.75"),
        "InPublication" => array("BOOL" => false),
    ),
    "ReturnConsumedCapacity" => "TOTAL"
));
echo "Consumed capacity: " . $response ["ConsumedCapacity"] ["CapacityUnits"]
. PHP_EOL;

$response = $client->putItem(array(
    "TableName" => $tableName,
    "Item" => array(
        "Id" => array("N" => "121"),
        "Title" => array("S" => "Book 121 Title"),
        "ISBN" => array("S" => "121-1111111111"),
        "Authors" => array("SS" => array("Author21", "Author22")),
        "Price" => array("N" => "20"),
        "Category" => array("S" => "Book"),
    ),
    "ReturnConsumedCapacity" => "TOTAL"
));
echo "Consumed capacity: " . $response ["ConsumedCapacity"] ["CapacityUnits"]
. PHP_EOL;
```

```

        "Dimensions" => array("S" => "8.5x11.0x.75"),
        "InPublication" => array("BOOL" => true),
    ) ,
    "ReturnConsumedCapacity" => "TOTAL"
) );

echo "Consumed capacity: " . $response ["ConsumedCapacity"] ["CapacityUnits"]
. PHP_EOL;

// ##### Getting an item from the table

echo PHP_EOL . PHP_EOL;
echo "# Getting an item from table $tableName..." . PHP_EOL;

$response = $client->getItem ( array (
    "TableName" => $tableName,
    "ConsistentRead" => true,
    "Key" => array (
        "Id" => array (
            "N" => "120"
        )
    ),
    "ProjectionExpression" => "Id, ISBN, Title, Authors"
) );
print_r ( $response ["Item"] );

// ##### Updating item attributes

echo PHP_EOL . PHP_EOL;
echo "# Updating an item and returning the whole new item in table $tableName..." .
PHP_EOL;

$response = $client->updateItem ( array (
    "TableName" => $tableName,
    "Key" => array (
        "Id" => array (
            "N" => 120 //was 121
        )
    ),
    "ExpressionAttributeNames" => array (
        "#NA" => "NewAttribute",
        "#A" => "Authors"
    ),
    "ExpressionAttributeValues" => array (
        ":val1" => array("S" => "Some Value"),
        ":val2" => array("SS" => array("Author YY", "Author ZZ"))
    ),
    "UpdateExpression" => "set #NA = :val1, #A = :val2",
    "ReturnValues" => "ALL_NEW"
) );
print_r ( $response ["Attributes"] );

// ##### Conditionally updating the Price attribute, only if it has not changed.

echo PHP_EOL . PHP_EOL;

```

```
echo "# Updating an item attribute only if it has not changed in table $tableName..." . PHP_EOL;

$response = $client->updateItem ( array (
    "TableName" => $tableName,
    "Key" => array (
        "Id" => array (
            "N" => "121"
        )
    ),
    "ExpressionAttributeNames" => array (
        "#P" => "Price"
    ),
    "ExpressionAttributeValues" => array(
        ":val1" => array("N" => "25"),
        ":val2" => array("N" => "20"),
    ),
    "UpdateExpression" => "set #P = :val1",
    "ConditionExpression" => "#P = :val2",
    "ReturnValues" => "ALL_NEW"
) ) ;

print_r ( $response [ "Attributes" ] );

// ##### Deleting an item

echo PHP_EOL . PHP_EOL;
echo "# Deleting an item and returning its previous values from in table $tableName..." . PHP_EOL;

$response = $client->deleteItem ( array (
    "TableName" => $tableName,
    "Key" => array (
        "Id" => array (
            "N" => "121"
        )
    ),
    "ReturnValues" => "ALL_OLD"
) );
print_r ( $response [ "Attributes" ] );

?>
```

Example: Batch Operations Using AWS SDK for PHP

Example: Batch Write Operation Using the AWS SDK for PHP

The following PHP code example uses batch write API to perform the following tasks:

- Put an item in the Forum table.
- Put and delete an item from the Thread table.

To learn more about the batch write operation, see [Batch Write: Putting and Deleting Multiple Items \(p. 171\)](#).

This code example assumes that you have followed the Getting Started ([Getting Started with DynamoDB \(p. 13\)](#)) and created the Forum and Thread tables. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

Note

For step-by-step instructions to test the following code example, see [Running PHP Examples \(p. 371\)](#).

```
<?php

use Aws\AwsClient;
use Aws\DynamoDb\AwsClient;
use Aws\DynamoDb\BatchWriteItemRequest;
use Aws\DynamoDb\BatchWriteItemResult;
use Aws\DynamoDb\DeleteRequest;
use Aws\DynamoDb\PutRequest;

require 'vendor/autoload.php';

$awsClient = AwsClient::factory([
    'profile' => 'default',
    'region'  => 'us-west-2' // replace with your desired region
]);

$tableNameOne = "Forum";
$tableNameTwo = "Thread";

$response = $awsClient->batchWriteItem([
    'RequestItems' => [
        $tableNameOne => [
            [
                'PutRequest' => [
                    'Item' => [
                        'Name' => [
                            'S' => "Amazon S3 Forum"
                        ],
                        'Threads' => [
                            'N' => "0"
                        ]
                    ]
                ]
            ],
            $tableNameTwo => [
                [
                    'PutRequest' => [
                        'Item' => [
                            'ForumName' => [
                                'S' => "Amazon S3 Forum"
                            ],
                            'Subject' => [
                                'S' => "My sample question"
                            ],
                            'Message' => [
                                'S' => "Message Text."
                            ],
                            'KeywordTags' => [
                                'SS' => [
                                    'S' => "Amazon S3",
                                    'S' => "Bucket"
                                ]
                            ]
                        ]
                    ]
                ],
                [
                    'DeleteRequest' => [
                        'Key' => [
                            'ForumName' => [
                                'S' => "Some hash value"
                            ],
                            'Subject' => [
                                'S' => "Some range key"
                            ]
                        ]
                    ]
                ]
            ]
        ]
    ]
]);
```

```
print_r($response);  
?>
```

Query and Scan Operations in DynamoDB

Topics

- [Narrowing the Results with Filter Expressions \(p. 184\)](#)
- [Capacity Units Consumed by Query and Scan \(p. 185\)](#)
- [Paginating the Results \(p. 185\)](#)
- [Count and ScannedCount \(p. 186\)](#)
- [Limit \(p. 186\)](#)
- [Read Consistency for Query and Scan \(p. 186\)](#)
- [Query and Scan Performance \(p. 186\)](#)
- [Parallel Scan \(p. 187\)](#)
- [Guidelines for Query and Scan \(p. 189\)](#)
- [Querying Tables in DynamoDB \(p. 192\)](#)
- [Scanning Tables in DynamoDB \(p. 215\)](#)

In addition to using primary keys to access and manipulate items, Amazon DynamoDB also provides two APIs for searching the data: [Query](#) and [Scan](#).

- **Query**

A [Query](#) operation finds items in a table using only primary key attribute values. You must provide a hash key attribute name and a distinct value to search for. You can optionally provide a range key attribute name and value, and use a comparison operator to refine the search results. By default, a [Query](#) returns all of the data attributes for items with the specified primary key(s); however, you can use the `ProjectionExpression` parameter so that the [Query](#) only returns some of the attributes, rather than all of them.

[Query](#) supports a specific set of comparison operators for choosing key values. You must specify the hash key attribute name and value as an equality condition. You can optionally specify a second condition, referring to the range key attribute; this condition allows you to choose from several conditional operators. For information about the available comparison operators, go to [Query](#) in the Amazon DynamoDB API Reference and refer to the `KeyConditions` parameter.

Tip

If your table has one or more secondary indexes, you can `Query` those indexes in the same way that you query a table. For more information, see [Improving Data Access with Secondary Indexes in DynamoDB \(p. 241\)](#).

A single `Query` 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. (For more information on filters, see [Narrowing the Results with Filter Expressions \(p. 184\)](#).)

A `Query` operation always returns a result set, but if no matching items are found, the result set will be empty.

For items with a given hash key, DynamoDB stores those items in sorted order by range key. In a `Query`, DynamoDB retrieves the items in sorted order, and then processes the items using `KeyConditions` and any filter expressions that may be present. Only then are the `Query` results sent back to the client.

`Query` results are always sorted by the range key. If the data type of the range key is `Number`, the results are returned in numeric order; otherwise, the results are returned in order of ASCII character code values. By default, the sort order is ascending. To reverse the order, set the `ScanIndexForward` parameter to `false`.

- **Scan**

A `Scan` operation examines every item in a table or a secondary index. By default, a `Scan` returns all of the data attributes for every item in the table or index. You can use the `ProjectionExpression` parameter so that the `Scan` only returns some of the attributes, rather than all of them.

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. (For more information on filters, see [Narrowing the Results with Filter Expressions \(p. 184\)](#).)

A `Scan` operation always returns a result set, but if no matching items are found, the result set will be empty.

Narrowing the Results with Filter Expressions

With a `Query` or a `Scan` operation, you can specify an optional filter expression to refine the results returned to you. A *filter expression* lets you apply conditions to the data, after it is queried or scanned, but before it is returned to you. Only the items that meet your conditions are returned to you.

Following are some examples of filter expressions. Note that these expressions use placeholders (such as `#V` and `:name`) instead of actual values. For more information, see [Expression Attribute Names \(p. 94\)](#) and [Expression Attribute Values \(p. ?\)](#).

- Query the `Thread` table for a particular `ForumName` (hash key) and `Subject` (range key). Of the items that are found, return only the most popular discussion threads — for example, those threads with more than a certain number of `Views`.

```
#V > :num
```

Note that `Views` is a reserved word in DynamoDB (see [Reserved Words in DynamoDB \(p. 649\)](#)), so we use an expression attribute name as a substitution.

- Scan the `Thread` table and return only the items that were last posted to by a particular user.

```
LastPostedBy = :name
```

Note

The syntax for a `FilterExpression` is identical to that of a `ConditionExpression`. In addition, `FilterExpression` use the same comparators, functions, and logical operators as `ConditionExpression`. For more information, see [Condition Expression Reference \(p. 97\)](#).

A single `Query` or `Scan` operation can retrieve a maximum of 1 MB of data. This limit applies before any filter expression is applied to the results.

Capacity Units Consumed by Query and Scan

When you create a table, you specify your read and write capacity unit requirements. If you add a global secondary index to the table, you must also provide the throughput requirements for that index.

You can use `Query` and `Scan` operations on secondary indexes in the same way that you use these operations on a table. If you `Query` or `Scan` a local secondary index, then capacity units are consumed from the table's provisioned throughput. However, if you perform these operations on a global secondary index, capacity units are consumed from the provisioned throughput of the index. This is because a global secondary index has its own provisioned throughput settings, separate from those of its table.

For more information about how DynamoDB computes the capacity units consumed by your operation, see [Capacity Units Calculations for Various Operations \(p. 57\)](#).

Note

For `Query` and `Scan` operations, DynamoDB calculates the amount of consumed provisioned throughput 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 the `ProjectionExpression` parameter.

The number of capacity units consumed will also be the same whether or not you specify a `FilterExpression`.

Paginating the Results

DynamoDB *paginates* the results from `Query` and `Scan` operations. With pagination, `Query` and `Scan` results are divided into distinct pieces; an application can process the first page of results, then the second page, and so on. The data returned from a `Query` or `Scan` operation is limited to 1 MB; this means that if you scan a table that has more than 1 MB of data, you'll need to perform another `Scan` operation to continue to the next 1 MB of data in the table.

If you query or scan for specific attributes that match values that amount to more than 1 MB of data, you'll need to perform another `Query` or `Scan` request for the next 1 MB of data. To do this, take the `LastEvaluatedKey` value from the previous request, and use that value as the `ExclusiveStartKey` in the next request. This will let you progressively query or scan for new data in 1 MB increments.

When the entire result set from a `Query` or `Scan` has been processed, the `LastEvaluatedKey` is `null`. This indicates that the result set is complete (i.e. the operation processed the "last page" of data).

If `LastEvaluatedKey` is anything other than `null`, this does not necessarily mean that there is more data in the result set. The only way to know when you have reached the end of the result set is when `LastEvaluatedKey` is `null`.

Count and ScannedCount

The DynamoDB Query and Scan APIs use the *Count* parameter. *Count* is used for two distinct purposes:

- In a request, set the *Count* parameter to `true` if you want DynamoDB to provide the total number of items that match the filter expression, instead of a list of the matching items.
- In a response, DynamoDB returns a *Count* value for the number of matching items in a request. If the matching items for a filter expression or query condition is over 1 MB, *Count* contains a partial count of the total number of items that match the request. To get the full count of items that match, take the *LastEvaluatedKey* value from the previous request, and use that value as the *ExclusiveStartKey* in the next request. Repeat this until DynamoDB no longer returns a *LastEvaluatedKey*.

Query and Scan operations also return a *ScannedCount* value. The *ScannedCount* value is the total number of items that were queried or scanned, *before* any filter expression was applied to the results.

Limit

The DynamoDB Query and Scan APIs allow a *Limit* value to restrict the size of the results.

In a request, set the *Limit* parameter to the number of items that you want DynamoDB to process before returning results.

In a response, DynamoDB returns all the matching results within the scope of the *Limit* value. For example, if you issue a Query or a Scan request with a *Limit* value of 6 and without a filter expression, the operation returns the first six items in the table that match the request parameters. If you also supply a *FilterExpression*, the operation returns the items within the first six items in the table that match the filter requirements.

For either a Query or Scan operation, DynamoDB might return a *LastEvaluatedKey* value if the operation did not return all matching items in the table. To get the full count of items that match, take the *LastEvaluatedKey* from the previous request and use it as the *ExclusiveStartKey* in the next request. Repeat this until DynamoDB no longer returns a *LastEvaluatedKey*.

Read Consistency for Query and Scan

A `Query` result is an eventually consistent read, but you can request a strongly consistent read instead. An eventually consistent read might not reflect the results of a recently completed `PutItem` or `UpdateItem` operation. For more information, see [Data Read and Consistency Considerations \(p. 9\)](#).

A `Scan` result is an eventually consistent read, meaning that changes to data immediately before the scan takes place might not be included in the scan results. The result set will not contain any duplicate items.

Query and Scan Performance

Generally, a `Query` operation is more efficient than a `Scan` operation.

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 table's request rate. For more information, see [Guidelines for Query and Scan \(p. 189\)](#).

A `Query` operation searches for a specific range of keys that satisfy a given set of key conditions. If you specify a filter expression, then DynamoDB must perform the extra step of removing data from the result set. A `Query` operation seeks the specified composite primary key, or range of keys, until one of the following events occur:

- The result set is exhausted.
- The number of items retrieved reaches the value of the `Limit` parameter, if specified.
- The amount of data retrieved reaches the maximum result set size limit of 1 MB.

`Query` performance depends on the amount of data retrieved, rather than the overall number of primary keys in a table or secondary index. The parameters for a `Query` operation (and consequently the number of matching keys) determine the performance of the query. For example, a query on a table that contains a large set of range key elements for a single hash key element can be more efficient than a query on another table that has fewer range key elements per hash key element, if the number of matching keys in the first table is fewer than in the second. The total number of primary keys, in either table, does not determine the efficiency of a `Query` operation. A filter expression can also impact the efficiency of a `Query`, because the items that don't match the filter must be removed from the result set. Avoid using a `Query` operation on a large table or secondary index with a filter that removes many results, if possible.

If a specific hash key element has a large range key element set, and the results cannot be retrieved in a single `Query` request, the `ExclusiveStartKey` continuation parameter allows you to submit a new query request from the last retrieved item without re-processing the data already retrieved.

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 secondary index, 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.

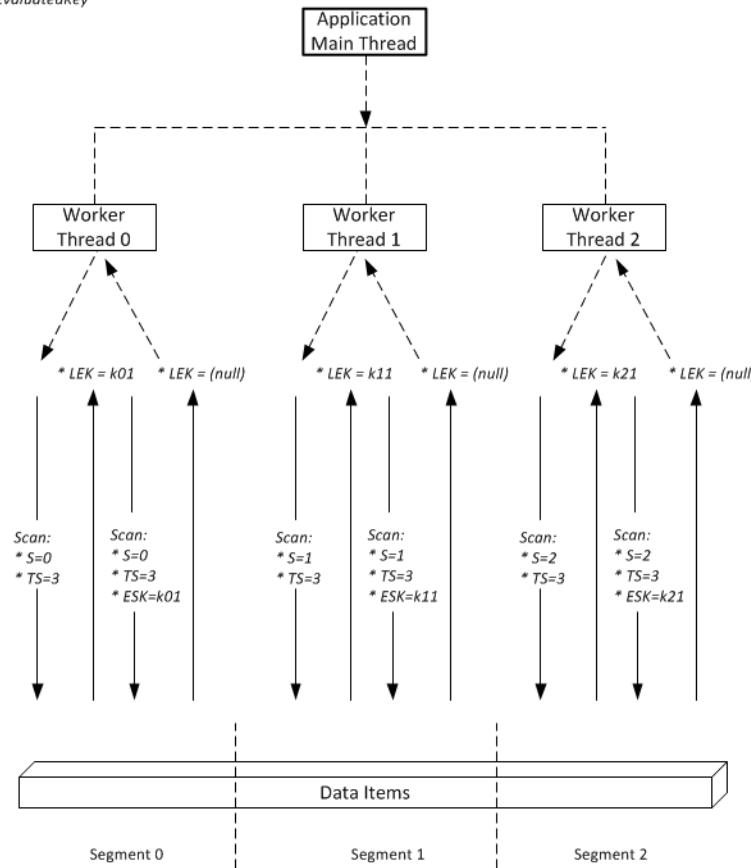
To address these issues, the `Scan` operation can logically divide a table or secondary index into multiple *segments*, with multiple application workers scanning the segments in parallel. Each worker can be a thread (in programming languages that support multithreading) or an operating system process. 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 a table's provisioned throughput; it is best to avoid such scans if the table 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. For more information, see "Reduce Page Size" in [Avoid Sudden Bursts of Read Activity \(p. 189\)](#).

Guidelines for Query and Scan

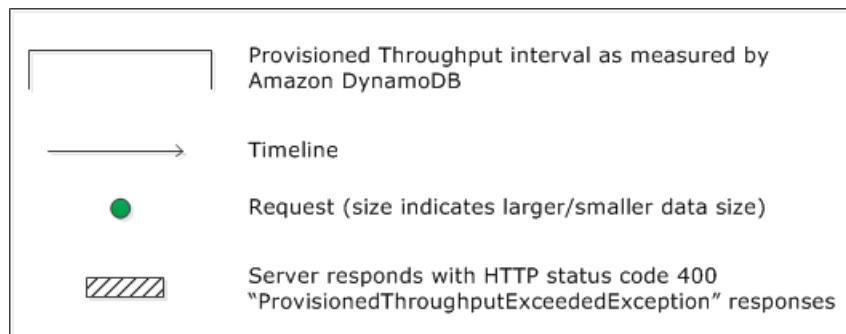
This section covers some best practices for query and scan operations.

Avoid Sudden Bursts of 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, 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. This represents a sudden burst of usage, compared to the configured read capacity for the table. This sudden use 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 burst of 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.

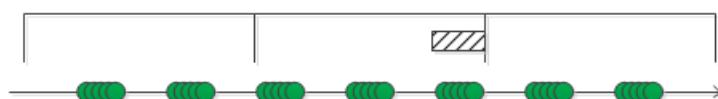
The following diagram illustrates the impact of a sudden burst of capacity unit usage by `Query` and `Scan` operations, and its impact on your other requests against the same table.



1. Good: Even distribution of requests and size



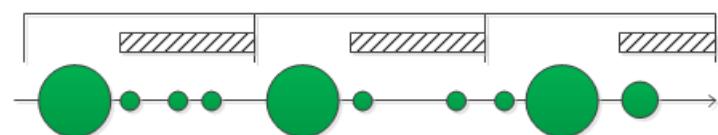
2. Not as Good: Frequent requests in bursts



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 Scan or Query 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 Scan or Query 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 \(p. 487\)](#).

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 4096, 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.

Querying Tables in DynamoDB

Topics

- [Querying Tables Using the AWS SDK for Java Document API \(p. 192\)](#)
- [Querying Tables Using the AWS SDK for .NET Low-Level API \(p. 198\)](#)
- [Querying Tables Using the AWS SDK for PHP Low-Level API \(p. 211\)](#)

This section shows basic queries and their results.

Querying Tables Using the AWS SDK for Java Document API

The `query` function enables you to query a table or a secondary index. You must provide a hash key value and an equality condition. If the table or index has a range key, you can refine the results by providing a range key value and a condition.

Note

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

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 hash key 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 [Example Tables and Data \(p. 609\)](#).

```
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. The `Id` and the `ReplyDateTime` make up the composite hash-and-range 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.

```
DynamoDB dynamoDB = new DynamoDB(  
    new AmazonDynamoDBClient(new ProfileCredentialsProvider()));  
  
Table table = dynamoDB.getTable("Reply");  
  
ItemCollection<QueryOutcome> items = table.query("Id", "Amazon DynamoDB#DynamoDB  
Thread 1");
```

```

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 range condition because DynamoDB evaluates the query condition that you specify against the range attribute 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 range key condition to retrieve only the replies in the past 15 days. The condition specifies a `ReplyDateTime` value and a comparison operator to use for comparing dates.
- The query filter condition specifies that only replies by a specific user are returned.
- The `withProjectionExpression` method to specify the attributes to retrieve for items in the query results.
- The `withConsistentRead` method as true to request a strongly consistent read. To learn more about read consistency, see [DynamoDB Data Model \(p. 3\)](#).

This snippet uses a `QuerySpec` object which gives access to all of the low-level Query input parameters.

```

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);

RangeKeyCondition rangeKeyCondition = new RangeKeyCondition("ReplyDateTime")
    .gt(twoWeeksAgoStr);

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", "User B");

QuerySpec spec = new QuerySpec()
    .withHashKey("Id", "Amazon DynamoDB#DynamoDB Thread 1")
    .withRangeKeyCondition(rangeKeyCondition)
    .withFilterExpression("PostedBy = :val")
    .WithValueMap(expressionAttributeValues)
    .withConsistentRead(true) ;

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 queries the Reply table. In the request, the `withMaxPageSize` method is used. The `Page` class provides an `Iterator` that allows the code to process the items on each page.

```
Table table = dynamoDB.getTable("Reply");

ItemCollection<QueryOutcome> items = table.query("Id", "Amazon DynamoDB#DynamoDB
    Thread 1")
    .withMaxPageSize(10); // 10 items per page

// 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 about table schemas, see [Example Tables and Data \(p. 609\)](#).

Note

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

For a code example that demonstrates query operations using the object persistence model, see [Example: Query and Scan \(p. 402\)](#).

```
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 range key conditions to narrow the query results and use other optional query parameters.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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.RangeKeyCondition;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;

public class DocumentAPIQuery {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new ProfileCredentialsProvider()));

    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;

        ItemCollection<QueryOutcome> items = table.query("Id", replyId);

        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().withHashKey("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;

        RangeKeyCondition rangeKeyCondition = new RangeKeyCondition("ReplyDateTime").le(twoWeeksAgoStr);

        ItemCollection<QueryOutcome> items = table.query("Id", replyId,
            rangeKeyCondition,
            null, //FilterExpression - not used in this example
            "Message, ReplyDateTime, PostedBy", //ProjectionExpression
            null, //ExpressionAttributeNames - not used in this example
            null); //ExpressionAttributeValues - not used in this example
    }
}

```

```

        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;

    RangeKeyCondition rangeKeyCondition = new RangeKeyCondition("ReplyDate
Time").between(startDate, endDate);

    ItemCollection<QueryOutcome> items = table.query("Id", replyId,
        rangeKeyCondition,
        null, //FilterExpression - not used in this example
        "Message, ReplyDateTime, PostedBy", //ProjectionExpression
        null, //ExpressionAttributeNames - not used in this example
        null); //ExpressionAttributeValues - not used in this example

    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;

    Map<String, Object> expressionAttributeValues = new HashMap<String,
Object>();
    expressionAttributeValues.put(":val", "User B");

    ItemCollection<QueryOutcome> items = table.query("Id", replyId,
        null, //RangeKeyCondition - not used in this example
        "PostedBy = :val", //FilterExpression
        "Message, ReplyDateTime, PostedBy", //ProjectionExpression
        null, //ExpressionAttributeNames - not used in this example

```

```
        expressionAttributeValues);

System.out.println("\nfindRepliesUsingAFilterExpression results:");
Iterator<Item> iterator = items.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next().toJSONPretty());
}
}
```

Querying Tables Using the AWS SDK for .NET Low-Level API

The `Query` function enables you to query a table or a secondary index. You must provide a hash key value and an equality condition. If the table or index has a range key, you can refine the results by providing a range key value and a condition.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables.

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 [Example Tables and Data \(p. 609\)](#).

```
Reply ( Id, ReplyDateTime, ... )
```

Each forum thread has a unique ID and can have zero or more replies. Therefore, the primary key is composed of both the `Id` (hash attribute) and `ReplyDateTime` (range attribute).

The following query retrieves all replies for a specific thread subject. The query requires both the table name and the `Subject` value.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

var request = new QueryRequest
{
    TableName = "Reply",
    KeyConditions = new Dictionary<string, Condition>()
```

```

    {
      {
        "Id",
        new Condition()
        {
          ComparisonOperator = "EQ",
          AttributeValueList = new List<AttributeValue>()
          {
            new AttributeValue { S = "DynamoDB#DynamoDB Thread 1" }
          }
        }
      }
    };
  }

var response = client.Query(request);
var result = response.QueryResult;

foreach (Dictionary<string, AttributeValue> item in response.QueryResult.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 range condition because Amazon DynamoDB evaluates the query condition that you specify against the range attribute 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 about the parameters and the API, 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 `RangeKeyCondition` parameter to retrieve only the replies in the past 15 days.
The condition specifies a `ReplyDateTime` value and a comparison operator to use for comparing dates.
- 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. To learn more about read consistency, see [DynamoDB Data Model \(p. 3\)](#).

```

DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
string twoWeeksAgoString = twoWeeksAgoDate.ToString(AWSSDKUtils.ISO8601DateFormat);

var request = new QueryRequest
{
  TableName = "Reply",
  KeyConditions = new Dictionary<string, Condition>()
  {
    {
      "Id",

```

```

        new Condition()
    {
        ComparisonOperator = "EQ",
        AttributeValueList = new List<AttributeValue>()
    {
        new AttributeValue { S = "DynamoDB#DynamoDB Thread 2" }
    }
}
},
{
{
    "ReplyDateTime",
    new Condition()
    {
        ComparisonOperator = "GT",
        AttributeValueList = new List<AttributeValue>()
    {
        new AttributeValue { S = twoWeeksAgoString }
    }
}
},
ProjectionExpression = "Subject, ReplyDateTime, PostedBy",
ConsistentRead = true
};

var response = client.Query(request);
var result = response.QueryResult;

foreach (Dictionary<string, AttributeValue> item
    in response.QueryResult.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.

```

Dictionary<string,AttributeValue> lastKeyEvaluated = null;
do
{
    var request = new QueryRequest
    {
        TableName = "Reply",
        KeyConditions = new Dictionary<string,Condition>()
    {

```

```

{
    "Id",
    new Condition()
    {
        ComparisonOperator = "EQ",
        AttributeValueList = new List<AttributeValue>()
        {
            new AttributeValue { S = "DynamoDB#DynamoDB Thread 2" }
        }
    }
},
// Optional parameters.
Limit = 10,
ExclusiveStartKey = lastKeyEvaluated
};

var response = client.Query(request);
// Process the query result.
foreach (Dictionary<string, AttributeValue> item in response.QueryResult.Items)
{
    PrintItem(item);
}
lastKeyEvaluated = response.QueryResult.LastEvaluatedKey;

} while (lastKeyEvaluated != null);

```

Example - Querying Using the AWS SDK for .NET

The following tables store information about a collection of forums. For more information about table schemas, see [Example Tables and Data \(p. 609\)](#).

Forum (<u>Name</u> , ...)
Thread (<u>ForumName</u> , <u>Subject</u> , Message, LastPostedBy, LastPostDateTime, ...)
Reply (<u>Id</u> , <u>ReplyDateTime</u> , 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 range key conditions to narrow query results and use other optional query parameters.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to

simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables. The individual object instances then map to items in a table.

For a code example that demonstrates query operations using the document model, see [Table.Query Method in the AWS SDK for .NET \(p. 429\)](#). For a code example that demonstrates query operations using the object persistence model, see [Example: Query and Scan in DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 470\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
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, threadSub
ject);
            }
        }

        private void FindRepliesForAThread(string forumName, string threadSubject)
        {
            var request = new QueryRequest()
            {
                TableName = forumName,
                IndexName = "PostId-Index",
                KeyConditionExpression = "PostId = :PostId",
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
                {
                    {":PostId", new AttributeValue() { S = "12345" } }
                },
                ConsistentRead = true
            };

            var response = client.Query(request);
            var items = response.Items;
            foreach (var item in items)
            {
                Console.WriteLine(item["PostId"].S + " " + item["Subject"].S);
            }
        }

        private void FindRepliesForAThreadSpecifyOptionalLimit(string forumName, string threadSubject)
        {
            var request = new QueryRequest()
            {
                TableName = forumName,
                IndexName = "PostId-Index",
                KeyConditionExpression = "PostId = :PostId",
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
                {
                    {":PostId", new AttributeValue() { S = "12345" } }
                },
                ConsistentRead = true,
                Limit = 2
            };

            var response = client.Query(request);
            var items = response.Items;
            foreach (var item in items)
            {
                Console.WriteLine(item["PostId"].S + " " + item["Subject"].S);
            }
        }
    }
}
```

```
        FindRepliesInLast15DaysWithConfig(forumName, threadSubject);

        FindRepliesPostedWithinTimePeriod(forumName, threadSubject);

    }

    Console.WriteLine("Example complete. 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 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",
    }
```

```
KeyConditions = new Dictionary<string, Condition>()

{
    "Id",
    new Condition()
    {
        ComparisonOperator = "EQ",
        AttributeValueList = new List<AttributeValue>()
        {
            new AttributeValue { S = replyId }
        }
    }
},
{
    "ReplyDateTime",
    new Condition
    {
        ComparisonOperator = "BETWEEN",
        AttributeValueList = new List<AttributeValue>()
        {
            new AttributeValue { S = start },
            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",  
            KeyConditions = new Dictionary<string, Condition>()  
        };
```

```
{  
    "Id",  
    new Condition  
    {  
        ComparisonOperator = "EQ",  
        AttributeValueList = new List<AttributeValue>()  
        {  
            new AttributeValue { S = replyId }  
        }  
    }  
,  
{  
    "ReplyDateTime",  
    new Condition  
    {  
        ComparisonOperator = "GT",  
        AttributeValueList = new List<AttributeValue>()  
        {  
            new AttributeValue { S = twoWeeksAgoString }  
        }  
    }  
,  
}  
,  
// Optional parameter.  
ProjectionExpression = "Id, ReplyDateTime, PostedBy",  
// Optional parameter.  
ConsistentRead = true  
};
```

```
var response = client.Query(request);

Console.WriteLine("No. of reads used (by query in FindRepliesIn
Last15DaysWithConfig) {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 FindRepliesForAThreadSpecifyOption
allLimit() ****");

    string replyId = forumName + "#" + threadSubject;

    Dictionary<string, AttributeValue> lastKeyEvaluated = null;

    do
    {
        var request = new QueryRequest
        {

            TableName = "Reply",

            ReturnConsumedCapacity = "TOTAL",

            KeyConditions = new Dictionary<string, Condition>()

        }
    }
}
```

```

    {
        "Id",
        new Condition
    {
        ComparisonOperator = "EQ",
        AttributeValueList = new List<AttributeValue>()
    {
        new AttributeValue { S = replyId }
    }
}
}

},
Limit = 2, // The Reply table has only a few sample items.
So the page size is smaller.

ExclusiveStartKey = lastKeyEvaluated
};

var response = client.Query(request);

Console.WriteLine("No. of reads used (by query in FindReplies
ForAThreadSpecifyLimit) {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",
        KeyConditions = new Dictionary<string, Condition>()
    };

    {
        "Id",
        new Condition
        {
            ComparisonOperator = "EQ",
            AttributeValueList = new List<AttributeValue>()
        };
        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.B == null ? "" : "B=[ " + value.B + " ]") +
            (value.M == null ? "" : "M={ " + value.M + " }"));
    }
}
```

```
        (value.SS == null ? "" : "SS=[ " + string.Join(",",
value.SS.ToArray()) + " ]") +
        (value.NS == null ? "" : "NS=[ " + string.Join(",",
value.NS.ToArray()) + " ]")
    );
}

Con
sole.WriteLine("*****");
}

}
```

Querying Tables Using the AWS SDK for PHP Low-Level API

The `query` function enables you to query a table or a secondary index. You must provide a hash key value and an equality condition. If the table or index has a range key, you can refine the results by providing a range key value and a condition.

The following steps guide you through querying using the AWS SDK for PHP.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `query` operation to the client instance.

You must provide the table name, any desired item's primary key values, and any optional query parameters. You can set up a condition as part of the query if you want to find a range of values that meet specific comparison results. You can limit the results to a subset to provide pagination of the results. Read results are eventually consistent by default. If you want, you can request that read results be strongly consistent instead.

3. Load the response into a local variable, such as `$response`, for use in your application.

Consider the following Reply table that stores replies for forum threads.

```
Reply ( Id, ReplyDateTime, ... )
```

Each forum thread has a unique ID and can have zero or more replies. Therefore, the primary key is made of both the `Id` (hash attribute) and `ReplyDateTime` (range attribute).

The following query retrieves all replies for a specific thread subject. The query requires the table name and the `Subject` value.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively,

you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$response = $client->query(array(
    'TableName' => 'Reply',
    'KeyConditions' => array(
        'Id' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array(
                array('S' => 'Amazon DynamoDB#DynamoDB Thread 1')
            )
        )
    )
));
print_r($response['Items']);
```

Specifying Optional Parameters

The `query` function 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 range condition. The condition is called a range condition because DynamoDB evaluates the query condition you specify against the range attribute of the primary key. You can specify other optional parameters to retrieve a specific list of attributes from items in the query result. For more information about the parameters, see [Query](#).

The following PHP example retrieves forum thread replies posted in the past 7 days. The sample specifies the following optional parameters:

- The range key attribute in the `KeyCondition` parameter to retrieve only the replies within the last 7 days.

The condition specifies `ReplyDateTime` value and a comparison operator to use for comparing dates.
- `ProjectionExpression` to specify the attributes to retrieve for items in the query results
- `ConsistentRead` parameter to perform a strongly consistent read. This overrides the default behavior of performing an eventually consistent reads. To learn more about read consistency, see [Data Read and Consistency Considerations \(p. 9\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

date_default_timezone_set("UTC");
$sevenDaysAgo = date('Y-m-d H:i:s', strtotime('-7 days'));

$response = $client->query(array(
```

```

'TableName' => 'Reply',
'KeyConditions' => array(
    'Id' => array(
        'ComparisonOperator' => 'EQ',
        'AttributeValueList' => array(
            array('S' => 'Amazon DynamoDB#DynamoDB Thread 2')
        )
    ),
    // optional range key condition
    'ReplyDateTime' => array(
        'ComparisonOperator' => 'GE',
        'AttributeValueList' => array(
            array('S' => $sevenDaysAgo)
        )
    )
),
// optional parameters
'ProjectionExpression' => 'Subject, ReplyDateTime, PostedBy',
'ConsistentRead' => true
));

print_r($response['Items']);

```

You can also optionally limit the page size, the number of items per page, by adding the `Limit` parameter. Each time you execute the `query` function, you get one page of results with the specified number of items. To fetch the next page you execute the `query` function again by providing primary key value of the last item in the previous page so 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. For retrieving subsequent pages you must update this property value to the primary key of the last item in the preceding page.

The following PHP example queries the `Reply` table for entries that are more than 14 days old. In the request it specifies the `Limit` and `ExclusiveStartKey` optional parameters.

```

<?php

use Aws\AwsClient;
use Aws\DynamoDb\Document;
use Aws\DynamoDb\Mapper;

$mapper = new Mapper($client);

echo "# Creating table $tableName..." . PHP_EOL;

$response = $mapper->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Id',
            'AttributeType' => 'N'
        )
    ),
    'KeySchema' => array(
        array(

```

```

        'AttributeName' => 'Id',
        'KeyType' => 'HASH'
    )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 5,
    'WriteCapacityUnits' => 6
)
));
);

print_r($response->getPath('TableDescription'));

$client->waitForTableExists(array('TableName' => $tableName));
echo "table $tableName has been created." . PHP_EOL;

#####
# Updating the table

echo "# Updating the provisioned throughput of table $tableName." . PHP_EOL;

$response = $client->updateTable(array(
    'TableName' => $tableName,
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 6,
        'WriteCapacityUnits' => 7
    )
));
;

// Wait until update completes
$client->waitForTableExists(array('TableName' => $tableName));

echo "New provisioned throughput settings:" . PHP_EOL;

echo "Read capacity units: " . $response['TableDescription']['ProvisionedThroughput']['ReadCapacityUnits'] . PHP_EOL;
echo "Write capacity units: " . $response['TableDescription']['ProvisionedThroughput']['WriteCapacityUnits'] . PHP_EOL;

#####
# Deleting the table

echo "# Deleting table $tableName..." . PHP_EOL;

$response = $client->deleteTable(array(
    'TableName' => $tableName
));
;

$client->waitForTableNotExists(array('TableName' => $tableName));
echo "The table has been deleted." . PHP_EOL;

#####
# Collect all table names in the account

echo "# Listing all the tables in the account..." . PHP_EOL;

$tables = array();

```

```
// Walk through table names, two at a time

do {
    $response = $client->listTables(array(
        'Limit' => 2,
        'ExclusiveStartTableName' => isset($response) ? $response['LastEvaluatedTableName'] : null
    ));

    foreach ($response['TableNames'] as $key => $value) {
        echo "$value" . PHP_EOL;
    }

    $tables = array_merge($tables, $response['TableNames']);
}

while ($response['LastEvaluatedTableName']);

// Print total number of tables

echo "Total number of tables: ";
print_r(count($tables));
echo PHP_EOL;

?>
```

Scanning Tables in DynamoDB

Topics

- [Scanning Tables Using the AWS SDK for Java Document API \(p. 215\)](#)
- [Scanning Tables Using the AWS SDK for .NET Low-Level API \(p. 224\)](#)
- [Scanning Tables Using the AWS SDK for PHP Low-Level API \(p. 234\)](#)

This section shows basic scans and their results.

Scanning Tables Using the AWS SDK for Java Document API

The `scan` method scans the entire table and you should therefore use queries to retrieve information. To learn more about performance related to scan and query operations, see [Query and Scan Operations in DynamoDB \(p. 183\)](#).

Note

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

The following are the steps to scan a table using the AWS SDK for Java Document API.

1. Create an instance of the `AmazonDynamoDBClient` 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.

```
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` (hash attribute) and `ReplyDateTime` (range attribute). The following Java code snippet scans the entire table. The `ScanRequest` instance specifies the name of the table to scan.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider());  
  
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 about the parameters and the API, 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.

```
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 a time until the `getLastEvaluatedKey` method of the result returns a value of null.

```
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

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

For a code example that demonstrates scan operations using the object persistence model, see [Example: Query and Scan \(p. 402\)](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;  
  
import java.util.HashMap;  
import java.util.Iterator;
```

```

import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new ProfileCredentialsProvider()));
    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

This section explains the AWS SDK for Java Document API. The AWS SDK for Java also provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB

tables. This approach can reduce the amount of code you have to write. For more information, see [Java: Object Persistence Model](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class DocumentAPIParallelScan {

    // total number of sample items
    static int scanItemCount = 300;

    // number of items each scan request should return
    static int scanItemLimit = 10;

    // number of logical segments for parallel scan
    static int parallelScanThreads = 16;

    // table that will be used for scanning
    static String parallelScanTestTableName = "ParallelScanTest";

    static DynamoDB dynamoDB = new DynamoDB(
        new AmazonDynamoDBClient(new ProfileCredentialsProvider()));

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

```

        // Clean up the table
        deleteTable(parallelScanTestTableName);
        createTable(parallelScanTestTableName, 10L, 5L, "Id", "N");

        // Upload sample data for scan
        uploadSampleProducts(parallelScanTestTableName, scanItemCount);

        // Scan the table using multiple threads
        parallelScan(parallelScanTestTableName, scanItemLimit, parallelScanThreads);
    }
    catch (AmazonServiceException ase) {
        System.err.println(ase.getMessage());
    }
}

private static void parallelScan(String tableName, int itemLimit, int num
berOfThreads) {
    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 hashKeyName, String hashKeyType) {

    createTable(tableName, readCapacityUnits, writeCapacityUnits,
        hashKeyName, hashKeyType, null, null);
}

private static void createTable(
    String tableName, long readCapacityUnits, long writeCapacityUnits,
    String hashKeyName, String hashKeyType,
    String rangeKeyName, String rangeKeyType) {

    try {
        System.out.println("Creating table " + tableName);

        List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();

        keySchema.add(new KeySchemaElement()

```

```

        .withAttributeName(hashKeyName)
        .withKeyType(KeyType.HASH));

    List<AttributeDefinition> attributeDefinitions = new ArrayList<AttributeDefinition>();
    attributeDefinitions.add(new AttributeDefinition()
        .withAttributeName(hashKeyName)
        .withAttributeType(hashKeyType));

    if (rangeKeyName != null){
        keySchema.add(new KeySchemaElement()
            .withAttributeName(rangeKeyName)
            .withKeyType(KeyType.RANGE));
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName(rangeKeyName)
            .withAttributeType(rangeKeyType));
    }

    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 printItem(int segment, Map<String,AttributeValue> attributeList) {
    System.out.print("Segment " + segment + ", ");
    for (Map.Entry<String,AttributeValue> item : attributeList.entrySet())
    {
        String attributeName = item.getKey();
        AttributeValue value = item.getValue();
        System.out.print(attributeName + " "
            + (value.getS() == null ? "" : "S=[ " + value.getS() + "]");
        + (value.getN() == null ? "" : "N=[ " + value.getN() + "]");
        + (value.getB() == null ? "" : "B=[ " + value.getB() + "]");
        + (value.getSS() == null ? "" : "SS=[ " + value.getSS() + "]");
        + (value.getNS() == null ? "" : "NS=[ " + value.getNS() + "]");
        + (value.getBS() == null ? "" : "BS=[ " + value.getBS() + "]");
        + ", ");
    }
    // Move to next line
    System.out.println();
}

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 Using the AWS SDK for .NET Low-Level API

The `Scan` method scans the entire table and you should therefore use queries to retrieve information. To learn more about performance related to scan and query operations, see [Query and Scan Operations in DynamoDB \(p. 183\)](#).

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables.

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.

```
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` (hash attribute) and `ReplyDateTime` (range attribute). The following C# code snippet scans the entire table. The `ScanRequest` instance specifies the name of the table to scan.

```
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 about the parameters and the API, 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.

```

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 a time until the `LastEvaluatedKey` returns a null value.

```

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.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables. The individual object instances then map to items in a table.

For a code example that demonstrates scan operations using the document model classes, see [Example: Scan using the Table.Scan method \(p. 437\)](#). For a code example that demonstrates scan operations using the object persistence model, see [Example: Query and Scan in DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 470\)](#).

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
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> {  
  
                {"<key>:val", new AttributeValue { N = "0" }}  
  
            },  
  
        };  
  
        var response = client.Scan(lastKeyEvaluated);  
  
        foreach (var item in response.Items)  
  
    }  
  
}
```

```
        FilterExpression = "Price < :val",  
  
        ProjectionExpression = "Id, Title, Price"  
    };  
  
    var response = client.Scan(request);  
  
    foreach (Dictionary<string, AttributeValue> item  
            in response.Items)  
    {  
        Console.WriteLine("\nScanThreadTableUsePaging - print  
ing.....");  
        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.

Note

This section explains the .NET SDK low-level API. The .NET SDK also provides a set of document model classes (see [.NET: Document Model \(p. 410\)](#)) that wrap some of the low-level API to simplify your coding tasks. In addition, the .NET SDK also provides a high-level *object persistence model* (see [.NET: Object Persistence Model \(p. 441\)](#)), enabling you to map your client-side classes to DynamoDB tables. The individual object instances then map to items in a table.

For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET](#) (p. 368).

```
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"
                }
            },
            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.ReadCapacity
                          Units,
                          tableDescription.ProvisionedThroughput.WriteCapa
                          cityUnits);

        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 de-
leted", tableName);
        return;
    }
} while (status == "DELETING");
}
}
}
```

Scanning Tables Using the AWS SDK for PHP Low-Level API

The `Scan` method scans the entire table and you should therefore use queries to retrieve information. To learn more about performance related to scan and query operations, see [Query and Scan Operations in DynamoDB \(p. 183\)](#).

The following tasks guide you through scanning a table using the AWS SDK for PHP low-level API:

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `scan` operation to the client instance.

The only required parameter is the table name. You can set up a filter as part of the scan if you want to find a set of values that meet specific comparison results. You can limit the results to a subset to provide pagination of the results. Read results are eventually consistent.

3. Load the response into a local variable, such as `$response`, for use in your application.

Consider the following Reply table that stores replies for various forum threads.

```
Reply ( Id, ReplyDateTime, Message, PostedBy )
```

The table maintains all the replies for various forum threads. Therefore, the primary key is made of both the `Id` (hash attribute) and `ReplyDateTime` (range attribute). The following PHP code snippet scans the table.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#) topic.

For step-by-step instructions to run the following example, see [Running PHP Examples \(p. 371\)](#).

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
```

```

));
$response = $client->scan(array(
    'TableName' => 'Reply'
));
foreach ($response['Items'] as $key => $value) {
    echo 'Id: ' . $value['Id'][['S']] . PHP_EOL;
    echo 'ReplyDateTime: ' . $value['ReplyDateTime'][['S']] . PHP_EOL;
    echo 'Message: ' . $value['Message'][['S']] . PHP_EOL;
    echo 'PostedBy: ' . $value['PostedBy'][['S']] . PHP_EOL;
    echo PHP_EOL;
}

```

The scan operation response is a [GuzzleService\Resource\Model](#) object. You can perform operations on the object contents. For example, the following code snippet scans the ProductCatalog table, and prints the product Id and Title values.

```

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));
$response = $client->scan(array(
    "TableName" => "ProductCatalog"
));
foreach ($response['Items'] as $key => $value) {
    echo "<p><strong>Item Number:</strong>". $value['Id'][['N']] . "</p>";
    echo "<br><strong>Item Name: </strong>". $value['Title'][['S']] . "</p>";
}

```

Specifying Optional Parameters

The `scan` function supports several optional parameters. For example, you can optionally use a filter expression to filter the scan result. In a filter expression you specify a condition and an attribute name on which you want the condition evaluated. For more information about the parameters and the API, see [Scan](#).

The following PHP code scans the ProductCatalog table to find items that are priced less than 0. The sample specifies the following optional parameters:

- `FilterExpression` to retrieve only the items priced less than 0 (error condition).
- `ProjectionExpression` to specify the attributes to retrieve for items in the query results

The following PHP code snippet scans the ProductCatalog table to find all items priced less than 0.

```

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));
$response = $client->scan(array(

```

```

'TableName' => 'ProductCatalog',
'ProjectionExpression' => 'Id, Price',
'ExpressionAttributeValues' => array (
    ':vall' => array('N' => '0')) ,
'FilterExpression' => 'Price < :vall',
));

foreach ($response['Items'] as $key => $value) {
    echo 'Id: ' . $value['Id']['N'] . ' Price: ' . $value['Price']['N'] . PHP_EOL;
    echo PHP_EOL;
}

```

You can also optionally limit the page size, the number of items per page, by adding the optional `Limit` parameter. Each time you execute the `scan` function, you get one page of results with a specified number of items. To fetch the next page you execute the `scan` function again by providing primary key value of the last item in the previous page so the `scan` function 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. For retrieving subsequent pages you must update this property value to the primary key of the last item in the preceding page.

The following PHP code snippet scans the `ProductCatalog` table. In the request it specifies the `Limit` and `ExclusiveStartKey` optional parameters.

```

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableName = 'ProductCatalog';

# The Scan API is paginated. Issue the Scan request multiple times.
do {
    echo "Scanning table $tableName" . PHP_EOL;
    $request = array(
        'TableName' => $tableName,
        'ExpressionAttributeValues' => array (
            ':vall' => array('N' => '201')
        ) ,
        'FilterExpression' => 'Price < :vall',
        'Limit' => 2
    );

    # Add the ExclusiveStartKey if we got one back in the previous response
    if(isset($response) && isset($response['LastEvaluatedKey'])) {
        $request['ExclusiveStartKey'] = $response['LastEvaluatedKey'];
    }

    $response = $client->scan($request);

    foreach ($response['Items'] as $key => $value) {
        echo 'Id: ' . $value['Id']['N'] . PHP_EOL;
        echo 'Title: ' . $value['Title']['S'] . PHP_EOL;
        echo 'Price: ' . $value['Price']['N'] . PHP_EOL;
        echo PHP_EOL;
    }
}

```

```

}

# If there is no LastEvaluatedKey in the response, there are no more items
# matching this Scan
while(isset($response['LastEvaluatedKey']));

```

Example - Loading Data Using PHP

The following PHP code example prepares sample data to be used by subsequent examples. The program deletes and then re-creates the ProductCatalog table, then loads the table with data.

Note

For step-by-step instructions to run these code examples, see [Running PHP Examples \(p. 371\)](#).

```

<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region'  => 'us-west-2' // replace with your desired region
));

$tableName = 'ProductCatalog';

// Delete an old DynamoDB table

echo "Deleting the table...\n";

$response = $client->deleteTable(array(
    'TableName' => $tableName
));

$client->waitForTableNotExists(array('TableName' => $tableName));

echo "The table {$tableName} has been deleted.\n";

// Create a new DynamoDB table

echo "# Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Id',
            'AttributeType' => 'N'
        )
    ),
    'KeySchema' => array(
        array(
            'AttributeName' => 'Id',
            'KeyType'       => 'HASH'
        )
    ),
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 5,

```

```

        'WriteCapacityUnits' => 6
    )
));
$client->waitForTableExists(array('TableName' => $tableName));

echo "Table {$tableName} has been created.\n";

// Populate DynamoDB table

echo "# Populating Items to $tableName...\n";

for ($i = 1; $i <= 100; $i++) {
    $response = $client->putItem(array(
        'TableName' => $tableName,
        'Item' => array(
            'Id'      => array( 'N'      => $i ), // Primary Key
            'Title'   => array( 'S'      => "Book {$i} Title" ),
            'ISBN'    => array( 'S'      => '111-1111111111' ),
            'Price'   => array( 'N'      => 25 ),
            'Authors' => array( 'SS'    => array('Author1', 'Author2') )
        )
    )));
}

$response = $client->getItem(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'Id' => array( 'N' => $i )
    )
));
echo "Item populated: {$response['Item']['Title']['S']}\n";
sleep(1);
}

echo "{$tableName} is populated with items.\n";
?>

```

Example - Scan Using PHP

The following PHP code example performs a serial *Scan* on the ProductCatalog table.

Note

Before you run this program, you will need to populate the ProductTable with data. For more details, see [Example - Loading Data Using PHP \(p. 237\)](#).

For step-by-step instructions to run these code examples, see [Running PHP Examples \(p. 371\)](#).

```

<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region'  => 'us-west-2' // replace with your desired region

```

```

)) ;

$tableName = 'ProductCatalog';
$params = array (
    'TableName' => $tableName,
    'ExpressionAttributeValues' => array (
        ':val1' => array('S' => 'Book')
    ) ,
    'FilterExpression' => 'contains (Title, :val1)' ,
    'Limit' => 10
);

// Execute scan operations until the entire table is scanned
$count = 0;
do {
    $response = $client->scan ( $params );
    $items = $response->get ( 'Items' );
    $count = $count + count ( $items );

    // Do something with the $items
    foreach ( $items as $item ) {
        echo "Scanned item with Title \"{$item['Title']}\"\n";
    }

    // Set parameters for next scan
    $params ['ExclusiveStartKey'] = $response ['LastEvaluatedKey'];
} while ( $params ['ExclusiveStartKey'] ) ;

echo "{$tableName} table scanned completely. {$count} items found.\n";

?>

```

Example - Parallel Scan Using PHP

The following PHP code example demonstrates a parallel scan, running multiple `Scan` requests at the same time. Finally, the program prints a summary of run time statistics.

Note

Before you run this program, you will need to populate the `ProductTable` with data. For more details, see [Example - Loading Data Using PHP \(p. 237\)](#).

For step-by-step instructions to run these code examples, see [Running PHP Examples \(p. 371\)](#).

```

<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
);

$tableName = 'ProductCatalog';
$totalSegments = 5;
$params = array(
    'TableName' => $tableName,
    'Segment' => 1
);

```

```

'ExpressionAttributeValues' => array (
    ':val1' => array('S' => 'Book')
) ,
'FilterExpression' => 'contains (Title, :val1)',
'Limit' => 10,
'TotalSegments' => $totalSegments
);

// Create initial scan commands for each segment
$pendingScanCommands = array();
for ($segment = 0; $segment < $totalSegments; $segment++) {
    $params['Segment'] = $segment;
    $pendingScanCommands[] = $client->getCommand('Scan', $params);
}

// Execute scan operations in parallel until the entire table is scanned
while(count($pendingScanCommands) > 0) {
    // Executes the 5 scan operations in parallel using cURL multi handles
    $completedScanCommands = $client->execute($pendingScanCommands);
    $pendingScanCommands = array();

    // Process responses of each scan command
    foreach ($completedScanCommands as $scanCommand) {
        $response = $scanCommand->getResult();
        $segment = $scanCommand->getPath('Segment');
        $items = $response->get('Items');

        // Do something with the items
        foreach($items as $item) {
            echo "Scanned item with Title \"{$item['Title']}\" and Segment
\"{$segment}\"\n";
        }

        // If LastEvaluatedKey is present we should continue scanning this
        segment
        // Otherwise we've reached to end of this segment
        if ($response['LastEvaluatedKey']) {
            echo "LastEvaluatedKey found creating new scan command for Segment:
{$segment}.\n";
            $params['Segment'] = $segment;
            $params['ExclusiveStartKey'] = $response['LastEvaluatedKey'];
            $pendingScanCommands[] = $client->getCommand('Scan', $params);
        } else {
            echo "Segment: {$segment} scanned completely!\n";
        }
    }
}

echo "Table {$tableName} scanned completely.\n";

?>

```

Improving Data Access with Secondary Indexes in DynamoDB

Topics

- [Global Secondary Indexes \(p. 253\)](#)
- [Local Secondary Indexes \(p. 303\)](#)

For efficient access to data in a table, Amazon DynamoDB creates and maintains indexes for the primary key attributes. This allows applications to quickly retrieve data 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. With a secondary index, queries are no longer restricted to the table primary key; you can also retrieve the data using the alternate key defined by the secondary index. A table can have multiple secondary indexes, which gives your applications access to many different query patterns.

The data in a secondary index consists of attributes that are *projected*, or copied, from the table into the index. When you create a secondary index, you define the alternate key for the index, along with any other attributes that you want to be projected in the index. DynamoDB copies these attributes into the index, along with the primary key attributes from the 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 table, any indexes on the table are also updated to reflect these changes.

DynamoDB supports two types of secondary indexes:

- **Global secondary index** — an index with a hash and range key that can be different from those on the table. A global secondary index is considered "global" because queries on the index can span all of the data in a table, across all partitions.
- **Local secondary index** — an index that has the same hash key as the table, but a different range key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a table partition that has the same hash key.

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
Key Schema	The key of a global secondary index can be either a hash or a hash-and-range type key.

Characteristic	Global Secondary Index
	-col I a -ces -chn yra -nl xed
Key Attributes	The index hash key and range key (if present) can be any scalar table attributes.

Characteristic	Global Secondary Index	-cd I a -ces -dno y r a -nl xed
Size Restrictions Per Hash Key	There are no size restrictions for global secondary oF indexes.	hcae hsah , yek eh t lact ez is f o l la -ni dexed set i t sum e b 0 1 B G r o .sel

Characteristic	Global Secondary Index	-cd I a -ces -dro y r a -nl x ed
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. 260) .	

Characteristic	Global Secondary Index
	-col I a -ces -dno yra -nli xed -col I a -ces -dno yra -ni sed era -erc data t a eh t eras en t tant uo y -erc eta a elat uo Y -nac ton dda a -col I a -ces -dno yra -ni xed o t n a -xe -tsi gn i elat ron nac uo y - ed etel yna -col I a -ces -dno yra

Characteristic	Global Secondary Index
	<ul style="list-style-type: none">-colI a-ces-dhoyra-nIxed
Queries and Partitions	<p>A global secondary index lets you query over the entire table, across all partitions.</p> <ul style="list-style-type: none">-colI a-ces-dhoyra-nixedsteluo yyreqrevoaelgis-rap-itoints a-epsdif icy beh thsahyekeula/n ieh t.yed

Characteristic	Global Secondary Index
Read Consistency	Queries on global secondary indexes support eventual consistency only.

Characteristic	Global Secondary Index	-cd I a -ces -chn yra -nl xed
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 table. The same holds true for global secondary index updates due to table writes.	

Characteristic	Global Secondary Index
	-col I a -ces -dno yra -ni xed -req se i r o snacs n o a -col I a -ces -dno yra -ni xed -noc enus daer - ac - ap ytic stiu nor f eh t elat new uo y et i w o t a ,elat sti -col I a -ces -dno yra -ni seed era osla - pu ,det eant - pu setad -noc enus et i w - ac - ap

Characteristic	Global Secondary Index	-col I a -ces -dro y r a -n l x ed
		ytic st inu nor f eh t elat

Characteristic	Global Secondary Index
Projected Attributes	<p>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.</p>

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 `LimitExceeded` exception.

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 \(p. 597\)](#). The name must be unique for the table it is associated with, but you can use the same name for indexes that are associated with different tables.
- The key schema for the index. Every attribute in the index key schema must be a scalar data type, not a multi-value set. Other requirements for the key schema depend on the type of index:
 - For a global secondary index, the hash key can be any table attribute. A range key is optional, and it too can be any table attribute.
 - For a local secondary index, the hash key must be the same as the table's hash key, and the range key must be a non-key table attribute.
- Additional attributes, if any, to project from the table into the index. These attributes are in addition to the table key attributes, which are automatically projected into every index. You can project attributes of any data type, including scalar data types and multi-valued 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 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 parent 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](#) action. `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 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 the table, all of the indexes associated with that table are also deleted.

Global Secondary Indexes

Topics

- [Attribute Projections \(p. 256\)](#)
- [Querying a Global Secondary Index \(p. 258\)](#)
- [Scanning a Global Secondary Index \(p. 258\)](#)
- [Data Synchronization Between Tables and Global Secondary Indexes \(p. 258\)](#)
- [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 259\)](#)
- [Storage Considerations for Global Secondary Indexes \(p. 260\)](#)
- [Managing Global Secondary Indexes \(p. 260\)](#)

- [Guidelines for Global Secondary Indexes \(p. 271\)](#)
- [Working with Global Secondary Indexes Using the AWS SDK for Java Document API \(p. 273\)](#)
- [Working with Global Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 281\)](#)
- [Working with Global Secondary Indexes Using the AWS SDK for PHP Low-Level API \(p. 292\)](#)

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 hash key (`UserId`) and a range key (`GameTitle`). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown)

GameScores						
Userid (hash key)	GameTitle (range key)	TopScore	TopScoreDateTime	Wins	Losses	...
"101"	"Galaxy Invaders"	5842	"2013-09-15:17:24:31"	21	72	...
"101"	"Meteor Blasters"	1000	"2013-10-22:23:18:01"	12	3	...
"101"	"Starship X"	24	"2013-08-31:13:14:21"	4	9	...
"102"	"Alien Adventure"	192	"2013-07-12:11:07:56"	32	192	...
"102"	"Galaxy Invaders"	0	"2013-09-18:07:33:42"	0	5	...
"103"	"Attack Ships"	3	"2013-10-19:01:13:24"	1	8	...
"103"	"Galaxy Invaders"	2317	"2013-09-11:06:53:00"	40	3	...
"103"	"Meteor Blasters"	723	"2013-10-19:01:13:24"	22	12	...
"103"	"Starship X"	42	"2013-07-11:06:53:00"	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 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 hash key of `GameTitle` and a range key of `TopScore`. Since the 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> (hash key)	<i>TopScore</i> (range key)	<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 range key, *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 hash key, and can have an optional range key. The index key schema can be different from the table schema; you could have a table with a hash type primary key, and create a global secondary index with a hash-and-range index key — or vice-versa. The index key attributes can consist of any attributes from the table, as long as the data types are scalar rather than multi-value sets.

You can project other 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 parent table. For example, if you queried *GameTitleIndex*, as shown in the diagram above, the query would not be able to access any attributes other than *GameTitle* and *TopScore*.

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> (hash key)	<i>TopScore</i> (range key)	<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:

UserId	GameTitle
400	Comet Quest

Because you didn't specify the *TopScore* attribute, DynamoDB would not propagate this item to *GameTitleIndex*. Thus, if you queried *GameScores* for all the Comet Quest items, you would get the following four items:

<i>UserId</i> (hash key)	<i>GameTitle</i> (range key)	<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> (hash key)	<i>TopScore</i> (range key)	<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 hash and range keys 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 hash and range 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 table into the global secondary index, as shown in this diagram:

<i>GameTitle</i> (hash key)	<i>TopScore</i> (range key)	<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 source 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 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" ,  
    KeyConditions: {  
        GameTitle: {  
            ComparisonOperator: "EQ" ,  
            AttributeValueList: [  
                "Meteor Blasters"  
            ]  
        }  
    } ,  
    ProjectionExpression: "UserId, TopScore" ,  
    ScanIndexForward: false  
}
```

In this query:

- DynamoDB accesses `GameTitleIndex`, using the `GameTitle` hash 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` API to retrieve all of the data from a global secondary index. You must provide the 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 [Narrowing the Results with Filter Expressions \(p. 184\)](#).

Data Synchronization Between Tables and Global Secondary Indexes

DynamoDB automatically synchronizes each global secondary index with its parent 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 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 range 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 \(p. 259\)](#).

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 parent table. A `Query` operation on a global secondary index consumes read capacity units from the index, not the 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 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 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 [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

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 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. Even if no data needs to be written to a particular global secondary index, the table write will be throttled if that index has insufficient write capacity.

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 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 table primary key (hash and range key attributes)
- 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 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 hash key or range 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 \(p. 261\)](#)
- [Describing the Global Secondary Indexes on a Table \(p. 261\)](#)
- [Adding a Global Secondary Index To an Existing Table \(p. 261\)](#)
- [Modifying an Index Creation \(p. 264\)](#)
- [Deleting a Global Secondary Index From a Table \(p. 264\)](#)
- [Detecting and Correcting Index Key Violations \(p. 264\)](#)

Creating a Table With Global Secondary Indexes

To create a table with one or more global secondary indexes, use the `CreateTable` operation with the `GlobalSecondaryIndexUpdates` parameter. For maximum query flexibility, you can create up to 5 global secondary indexes per table.

You must specify one attribute for the index hash key; you can optionally specify another attribute for the index range 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 \(p. 253\)](#)), neither `TopScore` nor `TopScoreDateTime` are key attributes; you could create a global secondary index with a hash key of `TopScore` and a range 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 data type, not a multi-value set. You can project attributes of any data type into a global secondary index; this includes scalar data types and multi-valued sets. For a complete list of data types, see [DynamoDB Data Types \(p. 6\)](#).

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 \(p. 259\)](#).

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 `GlobalSecondaryIndexes` 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 hash key; you can optionally specify another attribute for the index range 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 hash and range 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 or delete one global secondary index per `UpdateTable` operation. However, if you run multiple `UpdateTable` operations simultaneously, you can create multiple indexes at a time. You can run up to five of these `UpdateTable` operations on a table at once, and each operation can create exactly one index.

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 of the violating item writes 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 \(p. 264\)](#).

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 \(p. 264\)](#).

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 \(p. 521\)](#).

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 \(p. 262\)](#).

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. However, you can delete more than one index at a time by running multiple `UpdateTable` operations simultaneously. You can run up to five of these `UpdateTable` operations on a table at once, and each operation can delete exactly one index.

While the global secondary index, is being deleted, there is no effect on any read or write activity in the parent table. You will not be able to add or delete other indexes on the table until the deletion is complete; however, 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 hash 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 hash key is 2048 bytes, and the maximum length of a range 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.

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 hash and range keys, 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 \(p. 265\)](#).
- `-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	- eR @riq
awsCredentialsFile	The fully qualified name of a file containing your AWS credentials. The credentials file must be in the following format: <pre>accessKey = access_key_id_goes_here secretKey = secret_key_goes_here</pre>	S E Y
dynamoDBRegion	The AWS region in which the table resides. For example: us-west-2.	S E Y
tableName	The name of the DynamoDB table to be scanned.	S E Y
gsiHashKeyName	The name of the index hash key attribute.	S E Y
gsiHashKeyType	The data type of the index hash key attribute—String, Number, or Binary: S N B	S E Y
gsiRangeKeyName	The name of the index range key attribute. Do not specify this parameter if the index only has a hash key.	N
gsiRangeKeyType	The data type of the index range key attribute—String, Number, or Binary: S N B Do not specify this parameter if the index only has a hash key.	O N
recordDetails	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.	N
recordGsiValueInViolationRecord	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.	O N

Parameter Name	Description	- eR aRiq
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 style="margin-left: 40px;">detectionOutputPath = //local/path/file-name.csv</pre> <pre style="margin-left: 40px;">detectionOutputPath = s3://bucket/file-name.csv</pre> <p>Information in the output file appears in CSV format (comma-separated values). If you do not set detectionOutputPath, then the output file is named violation_detection.csv and is written to your current working directory.</p>	N
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 numOfSegments 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. 187).</p>	N
numOfViolations	The upper limit of index key violations to write to the output file. If set to -1 (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.	N
numOfRecords	The number of items in the table to be scanned. If set to -1 (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.	N
readWriteIOPSPercent	Regulates the percentage of provisioned read capacity units that are consumed during the table scan. Valid values range from 1 to 100. The default value (25) means that Violation Detector will consume no more than 25% of the table's provisioned read throughput.	N

Parameter Name	Description	- eR aRiQ
correctionInputPath	<p>The full path of the Violation Detector correction input file. If you run Violation Detector in correction mode, the contents of this file are used to modify or delete data items in the table that violate the global secondary index.</p> <p>The format of the <code>correctionInputPath</code> file is the same as that of the <code>detectionOutputPath</code> file. This lets you process the output from detection mode as input in correction mode.</p>	N
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/file-name.csv</pre> <pre>correctionOutputPath = s3://bucket/file-name.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>	N

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 [Example Tables and Data \(p. 609\)](#). 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`. This index has a primary hash key of *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 Vi
olation 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 hash key value of the item in the table.
- **Table Range Key**—the range key value of the item in the table.
- **GSI Hash Key Value**—the hash 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 range 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 does not have a range key attribute, no range 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.

Guidelines for Global Secondary Indexes

Topics

- [Choose a Key That Will Provide Uniform Workloads \(p. 271\)](#)
- [Take Advantage of Sparse Indexes \(p. 271\)](#)
- [Use a Global Secondary Index For Quick Lookups \(p. 272\)](#)
- [Create an Eventually Consistent Read Replica \(p. 272\)](#)

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 hash and range keys so that the data is evenly spread across multiple partitions.

This same guidance is true for global secondary indexes. Choose hash and range 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 hash 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 hash key and its range 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 hash key of Champ and a range 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 hash and range keys. 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 as many global secondary indexes as you need 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 [Fine-Grained Access Control for DynamoDB \(p. 536\)](#).) 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.

Working with Global Secondary Indexes Using the AWS SDK for Java Document API

You can use the AWS SDK for Java Document API for Java 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

Global secondary indexes must be created at the same time you create a table. To do this, use the `CreateTable` API 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 hash key is Location and the range 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 range 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 hash key is Date and its range 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`.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider() ));  
  
// Attribute definitions  
ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<Attrib  
uteDefinition>();  
  
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));
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.RANGE));

// 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));
indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Precipitation")
    .withKeyType(KeyType.RANGE));

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 the `DescribeTable` API. 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 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. Call the `describe` method on the `Table` object.

The following Java code snippet demonstrates the preceding steps.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

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 the `Query` API 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 hash key and range key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a hash key of `Date` and a range 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 following Java code snippet demonstrates the preceding steps.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
```

```

        new ProfileCredentialsProvider())));
Table table = dynamoDB.getTable("WeatherData");

Index index = table.getIndex("PrecipIndex");

RangeKeyCondition rangeKeyCondition = new RangeKeyCondition("Precipitation")
    .eq(0);
ItemCollection<QueryOutcome> items = index.query("Date", "2013-08-10",
rangeKeyCondition);

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 hash key attribute is `IssueId` and the range key is `Title`. There are three global secondary indexes on this table:

- *CreateDateIndex*—the hash key is `CreateDate` and the range key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- *TitleIndex*—the hash key is `IssueId` and the range key is `Title`. No attributes other than the table keys are projected into the index.
- *DueDateIndex*—the hash key is `DueDate`, and there is no range 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 [Running Java Examples for DynamoDB \(p. 367\)](#).

```

package com.amazonaws.codesamples;
import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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.RangeKeyCondition;
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.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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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));
        tableKeySchema.add(new KeySchemaElement()
            .withAttributeName("Title")
            .withKeyType(KeyType.RANGE));

        // 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),
            new KeySchemaElement()
                .withAttributeName("IssueId")
                .withKeyType(KeyType.RANGE))
        .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),
            new KeySchemaElement()
                .withAttributeName("IssueId")
                .withKeyType(KeyType.RANGE))
        .withProjection(new Projection()
            .withProjectionType("KEYS_ONLY"));

    // DueDateIndex
    GlobalSecondaryIndex dueDateIndex = new GlobalSecondaryIndex()
        .withIndexName("DueDateIndex")
        .withProvisionedThroughput(ptIndex)
        .withKeySchema(new KeySchemaElement()
            .withAttributeName("DueDate")
            .withKeyType(KeyType.HASH))
        .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*****");
        System.out.print("Querying index " + indexName + "...");

        Index index = table.getIndex(indexName);

        ItemCollection<QueryOutcome> items = null;

        if (indexName == "CreateDateIndex") {
            System.out.println("Issues filed on 2013-11-01");
            RangeKeyCondition rangeKeyCondition = new RangeKeyCondition("Is
sueId").beginsWith("A-");
            items = index.query("CreateDate", "2013-11-01", rangeKeyCondition);

        } else if (indexName == "TitleIndex") {
            System.out.println("Compilation errors");
            RangeKeyCondition rangeKeyCondition = new RangeKeyCondition("Is
sueId").beginsWith("A-");
            items = index.query("Title", "Compilation error", rangeKeyCondition);

        } else if (indexName == "DueDateIndex") {
            System.out.println("Items that are due on 2013-11-30");
            items = index.query("DueDate", "2013-11-30");
        } 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 Using the AWS SDK for .NET Low-Level API

Topics

- [Create a Table With a Global Secondary Index \(p. 281\)](#)
- [Describe a Table With a Global Secondary Index \(p. 283\)](#)
- [Query a Global Secondary Index \(p. 284\)](#)
- [Example: Global Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 285\)](#)

You can use the AWS SDK for .NET low-level API (protocol-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 API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

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

Global secondary indexes must be created at the same time you create a table. To do this, use the `CreateTable` API 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 hash key is `Location` and the range 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 range 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 hash key is Date and its range 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"}},
    {new KeySchemaElement {
        AttributeName = "Date",
        KeyType = "RANGE"}}
};

// 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"}},
    {new KeySchemaElement{AttributeName = "Precipitation",KeyType = "RANGE"}}
};

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 the `DescribeTable` API. 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.

```

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.Projection
Type);

        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 the `Query` API 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 hash key and range key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a hash key of Date and a range 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 following C# code snippet demonstrates the preceding steps.

```

client = new AmazonDynamoDBClient();
String tableName = "WeatherData";
String indexName = "PrecipIndex";

QueryRequest queryRequest = new QueryRequest {
    TableName = tableName,
    IndexName = indexName,
    ScanIndexForward = true
};

Dictionary<String, Condition> keyConditions = new Dictionary<String, Condition>();

keyConditions.Add(
    "Date",
    new Condition
    {
        ComparisonOperator = "EQ",
        AttributeValueList = { new AttributeValue { S = "2013-08-01" } }
    }
);

keyConditions.Add(
    "Precipitation",
    new Condition
    {
        ComparisonOperator = "GT",
        AttributeValueList = { new AttributeValue { N = "0.0" } }
    }
);

```

```

);
queryRequest.KeyConditions = keyConditions;

var result = client.Query(queryRequest);

var items = result.QueryResult.Items;
foreach (var currentItem in items)
{
    foreach (string attr in currentItem.Keys)
    {
        Console.Write(attr + "----> ");
        if (attr == "Precipitation")
        {
            Console.WriteLine(currentItem[attr].N);
        }
        else
        {
            Console.WriteLine(currentItem[attr].S);
        }
    }
    Console.WriteLine();
}
}

```

Example: Global Secondary Indexes Using the AWS SDK for .NET Low-Level API

The following C# code example shows how to work with global secondary indexes. The example creates a table named Issues, which might be used in a simple bug tracking system for software development. The hash key attribute is IssueId and the range key is Title. There are three global secondary indexes on this table:

- *CreateDateIndex*—the hash key is CreateDate and the range key is IssueId. In addition to the table keys, the attributes Description and Status are projected into the index.
- *TitleIndex*—the hash key is IssueId and the range key is Title. No attributes other than the table keys are projected into the index.
- *DueDateIndex*—the hash key is DueDate, and there is no range 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 [Running .NET Examples for DynamoDB \(p. 369\)](#).

```

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"
                    }
                },
                {
                    new KeySchemaElement {
                        AttributeName = "Title",
                        KeyType = "RANGE"
                    }
                }
            };
        }
    }
}
```

```

};

// 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"
        },
        new KeySchemaElement {
            AttributeName = "IssueId", KeyType = "RANGE"
        }
    },
    Projection = new Projection
    {
        ProjectionType = "INCLUDE",
        NonKeyAttributes = { "Description", "Status" }
    }
};

// TitleIndex
var titleIndex = new GlobalSecondaryIndex()
{
    IndexName = "TitleIndex",
    ProvisionedThroughput = ptIndex,
    KeySchema = {
        new KeySchemaElement {
            AttributeName = "Title", KeyType = "HASH"
        },
        new KeySchemaElement {
            AttributeName = "IssueId", KeyType = "RANGE"
        }
    },
    Projection = new Projection
    {
        ProjectionType = "KEYS_ONLY"
    }
};

// DueDateIndex
var dueDateIndex = new GlobalSecondaryIndex()
{
    IndexName = "DueDateIndex",
    ProvisionedThroughput = ptIndex,
    KeySchema = {
        new KeySchemaElement {
            AttributeName = "DueDate",
            KeyType = "HASH"
        }
    }
};

```

```

        },
        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, dueD
ateIndex }
};
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 incor
rect",
"2013-11-01", "2013-11-04", "2013-11-30",
2, "In progress");

    putItem("A-103", "Test failure",
"Functional test of Project X produces errors",
"2013-11-01", "2013-11-02", "2013-11-10",
1, "In progress");
}

```

```

        putItem("A-104", "Compilation error",
            "Variable 'messageCount' was not initialized.",
            "2013-11-15", "2013-11-16", "2013-11-30",
            3, "Assigned");

        putItem("A-105", "Network issue",
            "Can't ping IP address 127.0.0.1. Please fix this.",
            "2013-11-15", "2013-11-16", "2013-11-19",
            5, "Assigned");

    }

    private static void putItem(
        String issueId, String title,
        String description,
        String createDate, String lastUpdateDate, String dueDate,
        Int32 priority, String status)
    {

        Dictionary<String, AttributeValue> item = new Dictionary<string,
        AttributeValue>();

        item.Add("IssueId", new AttributeValue { S = issueId });
        item.Add("Title", new AttributeValue { S = title });
        item.Add("Description", new AttributeValue { S = description });
        item.Add("CreateDate", new AttributeValue { S = createDate });
        item.Add("LastUpdateDate", new AttributeValue { S = lastUpdateDate
    });
        item.Add("DueDate", new AttributeValue { S = dueDate });
        item.Add("Priority", new AttributeValue { N = priority.ToString()
    });
        item.Add("Status", new AttributeValue { S = status });

        try
        {

            client.PutItem(new PutItemRequest
            {
                TableName = tableName,
                Item = item
            });

        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }

    private static void QueryIndex(string indexName)
    {
        Console.WriteLine
        ("*****\n");
        Console.WriteLine("Querying index " + indexName + "...");

        QueryRequest queryRequest = new QueryRequest
        {
            TableName = tableName,

```

```

        IndexName = indexName,
        ScanIndexForward = true
    };

    Dictionary<String, Condition> keyConditions = new Dictionary<String, Condition>();

    if (indexName == "CreateDateIndex")
    {
        Console.WriteLine("Issues filed on 2013-11-01\n");
        keyConditions.Add("CreateDate", new Condition
        {
            ComparisonOperator = "EQ",
            AttributeValueList = { new AttributeValue { S = "2013-11-
01" } }
        });
    }

    keyConditions.Add("IssueId", new Condition
    {
        ComparisonOperator = "BEGINS_WITH",
        AttributeValueList = { new AttributeValue { S = "A-" } }
    });

    else if (indexName == "TitleIndex")
    {
        Console.WriteLine("Compilation errors\n");

        keyConditions.Add("Title", new Condition
        {
            ComparisonOperator = "EQ",
            AttributeValueList = { new AttributeValue { S = "Compilation
error" } }
        });
    }

    keyConditions.Add("IssueId", new Condition
    {
        ComparisonOperator = "BEGINS_WITH",
        AttributeValueList = { new AttributeValue { S = "A-" } }
    });

    // Select
    queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";

}

else if (indexName == "DueDateIndex")
{
    Console.WriteLine("Items that are due on 2013-11-30\n");

    keyConditions.Add("DueDate", new Condition
    {
        ComparisonOperator = "EQ",
        AttributeValueList = { new AttributeValue { S = "2013-11-
30" } }
    });

    // Select
    queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
}

```

```

        }
    else
    {
        Console.WriteLine("\nNo valid index name provided");
        return;
    }

    queryRequest.KeyConditions = keyConditions;
    var result = client.Query(queryRequest);
    var items = result.Items;
    foreach (var currentItem in items)
    {
        foreach (string attr in currentItem.Keys)
        {
            if (attr == "Priority")
            {
                Console.WriteLine(attr + "----> " + currentItem[attr].N);

            }
            else
            {
                Console.WriteLine(attr + "----> " + currentItem[attr].S);

            }
        }
        Console.WriteLine();
    }

}

private static void DeleteTable(string tableName)
{
    Console.WriteLine("Deleting table " + tableName + "...");
    client.DeleteTable(new DeleteTableRequest { TableName = tableName
});;
    WaitForTableToDelete(tableName);
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
        }
    }
}

```

```
        status = res.Table.TableStatus;
    }
    catch (ResourceNotFoundException)
    {
        // DescribeTable is eventually consistent. So you might
        // get resource not found. So we handle the potential exception.
    }
} while (status != "ACTIVE");
}

private static void WaitForTableToDelete(string tableName)
{
    bool tablePresent = true;

    while (tablePresent)
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
        }
        catch (ResourceNotFoundException)
        {
            tablePresent = false;
        }
    }
}
}
```

Working with Global Secondary Indexes Using the AWS SDK for PHP Low-Level API

Topics

- [Create a Table With a Global Secondary Index \(p. 293\)](#)
- [Describe a Table With a Global Secondary Index \(p. 294\)](#)
- [Query a Global Secondary Index \(p. 295\)](#)
- [Example: Global Secondary Indexes Using the AWS SDK for PHP Low-Level API \(p. 296\)](#)

You can use the AWS SDK for PHP 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 API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

The following are the common steps for table operations using the PHP low-level API.

1. Create an instance of the `DynamoDbClient` class (the client).
2. Provide the parameters for the `query` operation to the client instance.

You must provide the table name, index name, any desired item's primary key values, and any optional query parameters. You can set up a condition as part of the query if you want to find a range of values that meet specific comparison results. You can limit the results to a subset to provide pagination of the results. Read results from a global secondary index are always eventually consistent.

3. Load the response into a local variable, such as `$response`, for use in your application.

Create a Table With a Global Secondary Index

Global secondary indexes must be created at the same time you create a table. To do this, use the `CreateTable` API and provide your specifications for one or more global secondary indexes. The following PHP code snippet creates a table to hold information about weather data. The hash key is `Location` and the range 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 PHP low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the parameters for the `createTable` operation to the client instance.

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 range key, the key schema for the index, and the attribute projection.

The following PHP code snippet demonstrates the preceding steps. The snippet creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index hash key is `Date` and its range 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 = DynamoDbClient::factory(array
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
);

$tableName = 'WeatherData';

$result = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Location',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'Date',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'Precipitation',
            'AttributeType' => 'N'
        )
    ),
    'GlobalSecondaryIndexes' => array(
        array(
            'IndexName' => 'PrecipIndex',
            'KeySchema' => array(
                array(
                    'AttributeName' => 'Date',
                    'KeyType' => 'HASH'
                )
            ),
            'Projection' => array(
                'NonKeyAttributes' => array(
                    'AttributeName' => 'Precipitation'
                ),
                'Type' => 'INCLUDE'
            ),
            'ProvisionedThroughput' => array(
                'ReadCapacityUnits' => 5,
                'WriteCapacityUnits' => 5
            )
        )
    )
));
```

```

        'AttributeType' => 'N'
    )
),
'KeySchema' => array(
    array(
        'AttributeName' => 'Location',
        'KeyType' => 'HASH'
    ),
    array(
        'AttributeName' => 'Date',
        'KeyType' => 'RANGE'
    )
),
'GlobalSecondaryIndexes' => array(
    array(
        'IndexName' => 'PrecipIndex',
        'ProvisionedThroughput' => array (
            'ReadCapacityUnits' => 5,
            'WriteCapacityUnits' => 5
        ),
        'KeySchema' => array(
            array(
                'AttributeName' => 'Date',
                'KeyType' => 'HASH'
            ),
            array(
                'AttributeName' => 'Precipitation',
                'KeyType' => 'RANGE'
            )
        ),
        'Projection' => array(
            'ProjectionType' => 'ALL'
        )
    )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 5,
    'WriteCapacityUnits' => 5
)
));

```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a Table With a Global Secondary Index

To get information about global 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 global secondary index information a table using the PHP low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the `TableName` parameter for the `createTable` operation to the client instance.

The following PHP code snippet demonstrates the preceding steps.

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableName = 'WeatherData';

$result = $client->describeTable(array(
    'TableName' => $tableName
));

foreach ($result['Table']['GlobalSecondaryIndexes'] as $key => $value) {
    echo "Info for index " . $value['IndexName'] . ':' . PHP_EOL;
    foreach ($value['KeySchema'] as $attribute => $keyschema) {
        echo "\t" .
            $value['KeySchema'][$attribute]['AttributeName'] . ':' .
            $value['KeySchema'][$attribute]['KeyType'] . PHP_EOL;
    }
    echo "\tThe projection type is: " . $value['Projection']['ProjectionType']
. PHP_EOL;
}
```

Query a Global Secondary Index

You can use the `Query` API 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 hash key and range key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a hash key of `Date` and a range 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 PHP low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the parameters for the `query` operation to the client instance.

You must provide the table name, the index name, the key conditions for the query, and the attributes that you want returned.

The following PHP code snippet demonstrates the preceding steps.

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableName = 'WeatherData';

$response = $client->query(array(
    'TableName' => $tableName,
    'IndexName' => 'PrecipIndex',
    'KeyConditions' => array(
        'Date' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValue' => '2012-08-10'
        )
    ),
    'ProjectionExpression' => 'Precipitation'
));
```

```

        'AttributeValueList' => array(
            array('S' => '2014-08-01')
        )
    ),
    'Precipitation' => array(
        'ComparisonOperator' => 'GT',
        'AttributeValueList' => array(
            array('N'=> '0.0')
        )
    )
),
'Select' => 'ALL_ATTRIBUTES',
'ScanIndexForward' => true,
));
}

foreach ($response['Items'] as $item) {
    echo "Date ---> " . $item['Date']['S'] . PHP_EOL;
    echo "Location ---> " . $item['Location']['S'] . PHP_EOL;
    echo "Precipitation ---> " . $item['Precipitation']['N'] . PHP_EOL;
    echo PHP_EOL;
}
}

```

Example: Global Secondary Indexes Using the AWS SDK for PHP Low-Level API

The following PHP 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 hash key attribute is `IssueId` and the range key is `Title`. There are three global secondary indexes on this table:

- *CreateDateIndex*—the hash key is `CreateDate` and the range key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- *TitleIndex*—the hash key is `IssueId` and the range key is `Title`. No attributes other than the table keys are projected into the index.
- *DueDateIndex*—the hash key is `DueDate`, and there is no range 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.

```

<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

$tableName = 'Issues';

echo "# Creating table $tableName..." . PHP_EOL;

```

```
$response = $client->createTable ( array (
    'TableName' => $tableName,
    'AttributeDefinitions' => array (
        array (
            'AttributeName' => 'IssueId',
            'AttributeType' => 'S'
        ),
        array (
            'AttributeName' => 'Title',
            'AttributeType' => 'S'
        ),
        array (
            'AttributeName' => 'CreateDate',
            'AttributeType' => 'S'
        ),
        array (
            'AttributeName' => 'DueDate',
            'AttributeType' => 'S'
        )
    ),
    'KeySchema' => array (
        array (
            'AttributeName' => 'IssueId',
            'KeyType' => 'HASH'
        ),
        array (
            'AttributeName' => 'Title',
            'KeyType' => 'RANGE'
        )
    ),
    'GlobalSecondaryIndexes' => array (
        array (
            'IndexName' => 'CreateDateIndex',
            'KeySchema' => array (
                array (
                    'AttributeName' => 'CreateDate',
                    'KeyType' => 'HASH'
                ),
                array (
                    'AttributeName' => 'IssueId',
                    'KeyType' => 'RANGE'
                )
            ),
            'Projection' => array (
                'ProjectionType' => 'INCLUDE',
                'NonKeyAttributes' => array (
                    'Description',
                    'Status'
                )
            )
        ),
        'ProvisionedThroughput' => array (
            'ReadCapacityUnits' => 1,
            'WriteCapacityUnits' => 1
        )
    ),
    array (
        'IndexName' => 'TitleIndex',
        'KeySchema' => array (

```

```

        array (
            'AttributeName' => 'Title',
            'KeyType' => 'HASH'
        ),
        array (
            'AttributeName' => 'IssueId',
            'KeyType' => 'RANGE'
        )
    ),
    'Projection' => array (
        'ProjectionType' => 'KEYS_ONLY'
    ),
    'ProvisionedThroughput' => array (
        'ReadCapacityUnits' => 1,
        'WriteCapacityUnits' => 1
    )
),
array (
    'IndexName' => 'DueDateIndex',
    'KeySchema' => array (
        array (
            'AttributeName' => 'DueDate',
            'KeyType' => 'HASH'
        )
    ),
    'Projection' => array (
        'ProjectionType' => 'ALL'
    ),
    'ProvisionedThroughput' => array (
        'ReadCapacityUnits' => 1,
        'WriteCapacityUnits' => 1
    )
)
),
'ProvisionedThroughput' => array (
    'ReadCapacityUnits' => 1,
    'WriteCapacityUnits' => 1
)
) );
) ) ;

echo " Waiting for table $tableName to be created." . PHP_EOL;
$client->waitForTableExists ( array (
    'TableName' => $tableName
) );
echo " Table $tableName has been created." . PHP_EOL;

// ######
// Add items to the table

echo "# Loading data into $tableName..." . PHP_EOL;

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'IssueId' => array (
            'S' => 'A-101'
        ),
        'Title' => array (

```

```

        'S' => 'Compilation error'
),
'Description' => array (
    'S' => 'Can\'t compile Project X - bad version number. What does
this mean?'
),
'CreateDate' => array (
    'S' => '2014-11-01'
),
'LastUpdateDate' => array (
    'S' => '2014-11-02'
),
'DueDate' => array (
    'S' => '2014-11-10'
),
'Priority' => array (
    'N' => '1'
),
'Status' => array (
    'S' => 'Assigned'
)
)
)
) );
);

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'IssueId' => array (
            'S' => 'A-102'
),
        'Title' => array (
            'S' => 'Can\'t read data file'
),
        'Description' => array (
            'S' => 'The main data file is missing, or the permissions are incor
rect'
),
        'CreateDate' => array (
            'S' => '2014-11-01'
),
        'LastUpdateDate' => array (
            'S' => '2014-11-04'
),
        'DueDate' => array (
            'S' => '2014-11-30'
),
        'Priority' => array (
            'N' => '2'
),
        'Status' => array (
            'S' => 'In progress'
)
)
)
);
);

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (

```

```
'IssueId' => array (
    'S' => 'A-103'
),
'Title' => array (
    'S' => 'Test failure'
),
'Description' => array (
    'S' => 'Functional test of Project X produces errors.'
),
'CreateDate' => array (
    'S' => '2014-11-01'
),
'LastUpdateDate' => array (
    'S' => '2014-11-02'
),
'DueDate' => array (
    'S' => '2014-11-10'
),
'Priority' => array (
    'N' => '1'
),
>Status' => array (
    'S' => 'In progress'
)
)
)
) );
}

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'IssueId' => array (
            'S' => 'A-104'
),
        'Title' => array (
            'S' => 'Compilation error'
),
        'Description' => array (
            'S' => 'Variable "messageCount" was not initialized.'
),
        'CreateDate' => array (
            'S' => '2014-11-15'
),
        'LastUpdateDate' => array (
            'S' => '2014-11-16'
),
        'DueDate' => array (
            'S' => '2014-11-30'
),
        'Priority' => array (
            'N' => '3'
),
        'Status' => array (
            'S' => 'Assigned'
)
)
)
) );
}

$response = $client->putItem ( array (
```

```

'TableName' => $tableName,
'Item' => array (
    'IssueId' => array (
        'S' => 'A-105'
    ),
    'Title' => array (
        'S' => 'Network issue'
    ),
    'Description' => array (
        'S' => 'Can\'t ping IP address 127.0.0.1. Please fix this.'
    ),
    'CreateDate' => array (
        'S' => '2014-11-15'
    ),
    'LastUpdateDate' => array (
        'S' => '2014-11-16'
    ),
    'DueDate' => array (
        'S' => '2014-11-19'
    ),
    'Priority' => array (
        'N' => '5'
    ),
    'Status' => array (
        'S' => 'Assigned'
    )
)
)
)
);

// ##### Query for issues filed on 2014-11-01

$response = $client->query ( array (
    'TableName' => $tableName,
    'IndexName' => 'CreateDateIndex',
    'KeyConditions' => array (
        'CreateDate' => array (
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array (
                array (
                    'S' => '2014-11-01'
                )
            )
        ),
        'IssueId' => array (
            'ComparisonOperator' => 'BEGINS_WITH',
            'AttributeValueList' => array (
                array (
                    'S' => 'A-'
                )
            )
        )
    )
)
);
)

echo '# Query for issues filed on 2014-11-01:' . PHP_EOL;
foreach ( $response ['Items'] as $item ) {
    echo ' - ' . $item ['CreateDate'] ['S'] . ' ' . $item ['IssueId'] ['S'] .

```

```

        . $item ['Description'] ['S'] . ' ' . $item ['Status'] ['S'] .
PHP_EOL;
}

print PHP_EOL;

// ##### Query for issues that are 'Compilation errors'

$response = $client->query ( array (
    'TableName' => $tableName,
    'IndexName' => 'TitleIndex',
    'KeyConditions' => array (
        'Title' => array (
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array (
                array (
                    'S' => 'Compilation error'
                )
            )
        ),
        'IssueId' => array (
            'ComparisonOperator' => 'GE',
            'AttributeValueList' => array (
                array (
                    'S' => 'A-'
                )
            )
        )
    )
) );
echo '# Query for issues that are compilation errors: ' . PHP_EOL;

foreach ( $response ['Items'] as $item ) {
    echo ' - ' . $item ['Title'] ['S'] . ' ' . $item ['IssueId'] ['S'] . PHP_EOL;
}

print PHP_EOL;

// ##### Query for items that are due on 2014-11-30

$response = $client->query ( array (
    'TableName' => $tableName,
    'IndexName' => 'DueDateIndex',
    'KeyConditions' => array (
        'DueDate' => array (
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array (
                array (
                    'S' => '2014-11-30'
                )
            )
        )
    )
) );

```

```
echo '# Querying for items that are due on 2014-11-30:' . PHP_EOL;
foreach ( $response ['Items'] as $item ) {

    echo ' - ' . $item ['DueDate'] ['S'] . ' ' . $item ['IssueId'] ['S'] . ' '
    .
    $item ['Title'] ['S'] . ' ' . $item ['Description'] ['S'] . ' ' .
    $item ['CreateDate'] ['S'] . ' ' . $item ['LastUpdateDate'] ['S'] . ' ' .

    $item ['Priority'] ['N'] . ' ' . $item ['Status'] ['S'] . PHP_EOL;
}

print PHP_EOL;

// #####
// Delete the table

echo "# Deleting table $tableName..." . PHP_EOL;
$client->deleteTable ( array (
    'TableName' => $tableName
) );
$client->waitForTableNotExists ( array (
    'TableName' => $tableName
) );
echo " Deleted table $tableName" . PHP_EOL;

?>
```

Local Secondary Indexes

Topics

- [Attribute Projections \(p. 305\)](#)
- [Creating a Local Secondary Index \(p. 307\)](#)
- [Querying a Local Secondary Index \(p. 307\)](#)
- [Scanning a Local Secondary Index \(p. 308\)](#)
- [Item Writes and Local Secondary Indexes \(p. 309\)](#)
- [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 309\)](#)
- [Storage Considerations for Local Secondary Indexes \(p. 311\)](#)
- [Item Collections \(p. 311\)](#)
- [Guidelines for Local Secondary Indexes \(p. 314\)](#)
- [Working with Local Secondary Indexes Using the Java Document API \(p. 316\)](#)
- [Working with Local Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 326\)](#)
- [Working with Local Secondary Indexes Using the AWS SDK for PHP Low-Level API \(p. 340\)](#)

Some applications only need to query data using the table's primary key; however, there may be situations where an alternate range key would be helpful. To give your application a choice of range 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 [Example Tables and Data \(p. 609\)](#). 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 (hash key)	Subject (range key)	LastPostDateTime	Replies	
"S3"	"aaa"	"2013-03-15:17:24:31"	12	...
	"bbb"	"2013-01-22:23:18:01"	3	...
	"ccc"	"2013-02-31:13:14:21"	4	...
	"ddd"	"2013-01-03:09:21:11"	9	...
"EC2"	"yyy"	"2013-02-12:11:07:56"	18	...
	"zzz"	"2013-01-18:07:33:42"	0	...
"RDS"	"rrr"	"2013-01-19:01:13:24"	3	...
	"sss"	"2013-03-11:06:53:00"	11	...
	"ttt"	"2013-02-22:12:19:44"	5	...
...

DynamoDB stores all of the items with the same hash key 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 hash key, the items are sorted by range key. If the range 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 range key for a given hash key. A local secondary index also contains a copy of some or all of the attributes from the 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 hash key as the table, but with a different range 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 range key and find the data quickly.

The following diagram shows a local secondary index named *LastPostIndex*. Note that the hash key is the same as that of the *Thread* table, but the range key is *LastPostDateTime*.

<i>LastPostIndex</i>		
<i>ForumName</i> (hash key)	<i>LastPostDateTime</i> (range key)	<i>Subject</i>
“S3”	“2013-01-03:09:21:11”	“ddd”
“S3”	“2013-01-22:23:18:01”	“bbb”
“S3”	“2013-02-31:13:14:21”	“ccc”
“S3”	“2013-03-15:17:24:31”	“aaa”
“EC2”	“2013-01-18:07:33:42”	“zzz”
“EC2”	“2013-02-12:11:07:56”	“yyy”
“RDS”	“2013-01-19:01:13:24”	“rrr”
“RDS”	“2013-02-22:12:19:44”	“ttt”
“RDS”	“2013-03-11:06:53:00”	“sss”
...

Every local secondary index must meet the following conditions:

- The hash key is the same as that of the source table.
- The range key consists of a single attribute.
- The range key attribute of the source table is projected into the index, where it acts as a non-key attribute.

In this example, the hash key is *ForumName* and the range key of the local secondary index is *LastPostDateTime*. In addition, the range key value from the source 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 hash and range attributes from its parent 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 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 hash key value. This figure includes all of the items in the table, plus all of the items in the indexes, that have the same hash key. For more information, see [Item Collections \(p. 311\)](#).

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

<i>ForumName</i> (hash key)	<i>LastPostDateTime</i> (range key)	<i>Subject</i>	<i>Replies</i>
"S3"	"2013-01-03:09:21:11"	"ddd"	9
"S3"	"2013-01-22:23:18:01"	"bbb"	3
"S3"	"2013-02-31:13:14:21"	"ccc"	4
"S3"	"2013-03-15:17:24:31"	"aaa"	12
"EC2"	"2013-01-18:07:33:42"	"zzz"	0
"EC2"	"2013-02-12:11:07:56"	"yyy"	18
"RDS"	"2013-01-19:01:13:24"	"rrr"	3
"RDS"	"2013-02-22:12:19:44"	"ttt"	5
"RDS"	"2013-03-11:06:53:00"	"sss"	11
...

A *projection* is the set of attributes that is copied from a table into a secondary index. The hash and range keys 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 hash and range 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 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 source 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 for the range key of the local secondary index. The attribute that you choose must be a scalar data type, not a multi-value set. For a complete list of data types, see [DynamoDB Data Types \(p. 6\)](#).

Important

For tables with local secondary indexes, there is a 10 GB size limit per hash key. A table with local secondary indexes can store any number of items, as long as the total size for any one hash key does not exceed 10 GB. For more information, see [Item Collection Size Limit \(p. 313\)](#).

You can project attributes of any data type into a local secondary index. This includes scalar data types and multi-valued sets. For a complete list of data types, see [DynamoDB Data Types \(p. 6\)](#).

For an example `CreateTable` request that includes a local secondary index, go to [CreateTable](#) in the Amazon DynamoDB API Reference.

Querying a Local Secondary Index

In a DynamoDB table, the combined hash key and range key value for each item must be unique. However, in a local secondary index, the range key value does not need to be unique for a given hash key value. If there are multiple items in the local secondary index that have the same range key value, a `Query` operation will return all of the items that have the same hash 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 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",  
    KeyConditions: {  
        ForumName: {  
            ComparisonOperator: "EQ",  
            AttributeValueList: [ "EC2" ]  
        },  
        LastPostDateTime: {  
            ComparisonOperator: "BETWEEN",  
            AttributeValueList: [  
                "2012-08-31T00:00:00.000Z",  
                "2012-11-31T00:00:00.000Z"  
            ]  
        }  
    },  
    ProjectionExpression: "Subject, LastPostDateTime, Replies, Tags",  
    ConsistentRead: false  
}
```

In this query:

- DynamoDB accesses `LastPostIndex`, using the `ForumName` hash 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 hash key and then by range key, 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 table. If you need to run this query often, you should project `Tags` into `LastPostIndex` to avoid fetching from the 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 the `Scan` API to retrieve all of the data from a local secondary index. You must provide the 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 [Narrowing the Results with Filter Expressions \(p. 184\)](#).

Item Writes and Local Secondary Indexes

DynamoDB automatically keeps all local secondary indexes synchronized with their respective 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 range key for the index. You also specify a data type for that attribute. This means that whenever you write an item to the 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* range 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 range 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 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 \(p. 315\)](#).

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 \(p. 309\)](#).

Important

For tables with local secondary indexes, there is a 10 GB size limit per hash key. A table with local secondary indexes can store any number of items, as long as the total size for any one hash key does not exceed 10 GB. For more information, see [Item Collection Size Limit \(p. 313\)](#).

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 <http://www.amazonaws.cn/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 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 [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

For index queries that read attributes that are not projected into the local secondary index, DynamoDB will need to fetch those attributes from the 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 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 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 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 bytes \times 4 items = 800 bytes; this is then rounded up to 4 KB.
- The size of each matching item in the table: (300 bytes, rounded up to 4 KB) \times 4 items = 16 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 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 table primary key (hash and range key attributes)
- 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 table.

If a table contains an item where a particular attribute is not defined, but that attribute is defined as an index range 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 \(p. 315\)](#).

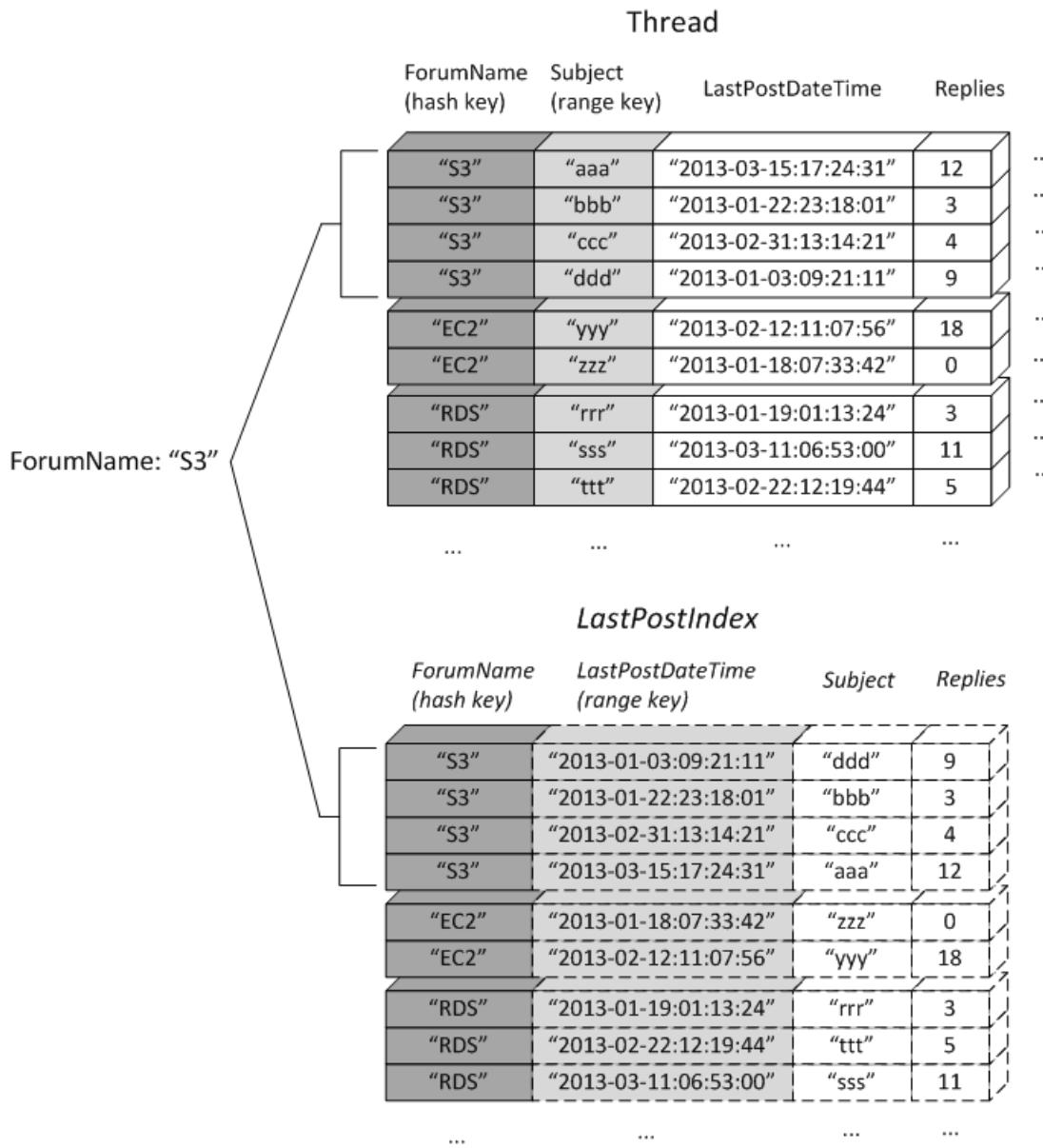
Item Collections

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 hash key, in a table and all of its local secondary indexes. In the examples used throughout this section, the hash key for the *Thread* table is *ForumName*, and the hash 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 hash key 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 `ItemCollectionSizeLimitExceededException` and you won't be able to add more items to the item collection or increase the sizes of items that are in the item collection. (Read and write operations that shrink the size of the item collection are still allowed.) You 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 hash key in question. When you delete these items from the table, DynamoDB will also remove any index entries that have the same hash key.
- 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 hash and range 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 hash key.

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 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 hash 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.

Guidelines for Local Secondary Indexes

Topics

- [Use Indexes Sparingly \(p. 314\)](#)
- [Choose Projections Carefully \(p. 314\)](#)
- [Optimize Frequent Queries To Avoid Fetches \(p. 315\)](#)
- [Take Advantage of Sparse Indexes \(p. 315\)](#)
- [Watch For Expanding Item Collections \(p. 315\)](#)

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 range 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 \(p. 309\)](#).

Take Advantage of Sparse Indexes

For any item in a table, DynamoDB will only write a corresponding index entry if the index range key attribute value is present in the item. If the range key attribute 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:

- Hash key: CustomerId
- Range 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* (hash) and *IsOpen* (range). 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* (hash) and *OrderOpenDate* (range). 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 hash key. An item collection cannot exceed 10 GB, so it's possible to run out of space for a particular hash key.

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 hash key. If you expect that the sum of table and index items for a particular hash key 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 \(p. 313\)](#).

Working with Local Secondary Indexes Using the Java Document API

Topics

- [Create a Table With a Local Secondary Index \(p. 316\)](#)
- [Describe a Table With a Local Secondary Index \(p. 318\)](#)
- [Query a Local Secondary Index \(p. 318\)](#)
- [Example: Local Secondary Indexes Using the Java Document API \(p. 319\)](#)

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 hash key is *Artist* and the range 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 attribute definitions for the index range 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 (*MusicCollection*) with a secondary index on the *AlbumTitle* attribute. The table hash and range key, plus the index range key, are the only attributes projected into the index.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
String tableName = "MusicCollection";
```

```

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));
tableKeySchema.add(new KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE));

createTableRequest.setKeySchema(tableKeySchema);

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH));
indexKeySchema.add(new KeySchemaElement().withAttributeName("AlbumTitle").withKeyType(KeyType.RANGE));

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.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
String tableName = "MusicCollection";  
  
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.getProjection  
Type());  
    if (projection.getProjectionType().toString().equals("INCLUDE")) {  
        System.out.println("\t\tThe non-key projected attributes are: " + projec  
tion.getNonKeyAttributes());  
    }  
}
```

Query a Local Secondary Index

You can use the `Query` API 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 range key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index range 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 \(p. 305\)](#).

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.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(  
    new ProfileCredentialsProvider()));  
  
String tableName = "MusicCollection";  
  
Table table = dynamoDB.getTable(tableName);  
Index index = table.getIndex("AlbumTitleIndex");  
  
ItemCollection<QueryOutcome> items = index.query("Artist", "Acme Band",  
    new RangeKeyCondition("AlbumTitle").eq("Songs About Life"));  
  
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 hash key attribute of `CustomerId` and a range key attribute of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex`—the range key is `OrderCreationDate`, and the following attributes are projected into the index:
 - `ProductCategory`
 - `ProductName`
 - `OrderStatus`
 - `ShipmentTrackingId`
- `IsOpenIndex`—the range 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 [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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.RangeKeyCondition;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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 hash and range key
        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 range keys
        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));
        tableKeySchema.add(new KeySchemaElement()
            .withAttributeName("OrderId")
            .withKeyType(KeyType.RANGE));

        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));
        indexKeySchema.add(new KeySchemaElement()
            .withAttributeName("OrderCreationDate")
            .withKeyType(KeyType.RANGE));

        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));
indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("IsOpen")
    .withKeyType(KeyType.RANGE));

// 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)
.withHashKey("CustomerId", "bob@example.com");

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.withRangeKeyCondition( new RangeKeyCondition("IsOpen")
.eq(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 01/31/2013.");
    System.out.println("Only the projected attributes are returned\n");

    Index index = table.getIndex(indexName);

    querySpec.withRangeKeyCondition(new RangeKeyCondition(
        "OrderCreationDate").gt(20130131));

    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");

    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", 20130101)
        .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", 20130221)
        .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", 20130304)
        .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", 20130111)
.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", 20130124)
.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", 20130221)
.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", 20130222)
.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", 20130309)
.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", 20130318)
        .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", 20130324)
        .withString("ProductCategory", "Golf")
        .withString("ProductName", "PGA Pro II")
        .withString("OrderStatus", "OUT FOR DELIVERY")
        .withString("ShipmentTrackingId", "383283");

    putItemOutcome = table.putItem(item);

}

}
```

Working with Local Secondary Indexes Using the AWS SDK for .NET Low-Level API

Topics

- [Create a Table With a Local Secondary Index \(p. 327\)](#)
- [Describe a Table With a Local Secondary Index \(p. 328\)](#)
- [Query a Local Secondary Index \(p. 329\)](#)
- [Example: Local Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 330\)](#)

You can use the AWS SDK for .NET low-level API (protocol-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 DynamoDB API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

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 the `CreateTable` API 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 hash key is `Artist` and the range 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 attribute definitions for the index range 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 (`MusicCollection`) with a secondary index on the `AlbumTitle` attribute. The table hash and range key, plus the index range key, are the only attributes projected into the index.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "MusicCollection";

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" });
tableKeySchema.Add(new KeySchemaElement() { AttributeName = "SongTitle", KeyType
= "RANGE" });

createTableRequest.KeySchema = tableKeySchema;

List<KeySchemaElement> indexKeySchema = new List<KeySchemaElement>();
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType
= "HASH" });
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "AlbumTitle", KeyType
= "RANGE" });

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<LocalSecondaryIn
dex>();
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.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "MusicCollection";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest()
{ TableName = tableName });
List<LocalSecondaryIndexDescription> localSecondaryIndexes =
    response.DescribeTableResult.Table.LocalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.
foreach (LocalSecondaryIndexDescription lsiDescription in localSecondaryIndexes)
{
    Console.WriteLine("Info for index " + lsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in lsiDescription.KeySchema)
    {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " +
kse.KeyType);
    }

    Projection projection = lsiDescription.Projection;

    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

    if (projection.ProjectionType.ToString().Equals("INCLUDE"))
    {
        Console.WriteLine("\t\tThe non-key projected attributes are:");

        foreach (String s in projection.NonKeyAttributes)
        {
            Console.WriteLine("\t\t" + s);
        }
    }
}
```

Query a Local Secondary Index

You can use the `Query` API 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 range key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index range 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 \(p. 305\)](#)

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.

```

QueryRequest queryRequest = new QueryRequest
{
    TableName = tableName,
    IndexName = "AlbumTitleIndex",
    ConsistentRead = true,
    Select = "ALL_ATTRIBUTES",
    ScanIndexForward = true
};

Dictionary<string, Condition> keyConditions = new Dictionary<string, Condition>();

keyConditions.Add("Artist",
    new Condition()
    {
        ComparisonOperator = "EQ",
        AttributeValueList = { new AttributeValue { S = "Acme Band" } }
    }
);

keyConditions.Add("AlbumTitle",
    new Condition()
    {
        ComparisonOperator = "EQ",
        AttributeValueList = { new AttributeValue { S = "Songs About Life" } }
    }
);

queryRequest.KeyConditions = keyConditions;
QueryResponse response = client.Query(queryRequest);

foreach (var attribs in response.QueryResult.Items)
{
    foreach (var attrib in attribs)
    {
        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 hash key attribute of `CustomerId` and a range key attribute of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex`—the range key is `OrderCreationDate`, and the following attributes are projected into the index:

- ProductCategory
- ProductName
- OrderStatus
- ShipmentTrackingId
- *IsOpenIndex*—the range 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 [Running .NET Examples for DynamoDB \(p. 369\)](#).

```
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 hash and range key
        { new AttributeDefinition() {AttributeName = "CustomerId", AttributeType = "S"} },
        { new AttributeDefinition() {AttributeName = "OrderId", AttributeType = "N"} },
        // Attribute definition for index range keys
        { 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" } }
    };

    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" } }
    };
}

```

```

        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)
    {
        Con
sole.WriteLine("\n*****\n");

        Console.WriteLine("Querying table " + tableName + "...");

        QueryRequest queryRequest
            = new QueryRequest()
        {
            TableName = tableName,
            ConsistentRead = true,

```

```

        ScanIndexForward = true,
        ReturnConsumedCapacity = "TOTAL"
    } ;

    Dictionary<string, Condition> keyConditions = new Dictionary<string, Condition>();

    keyConditions.Add(
        "CustomerId",
        new Condition
    {
        ComparisonOperator = "EQ",
        AttributeValueList =
            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;

    keyConditions.Add(
        "IsOpen",
        new Condition
    {
        ComparisonOperator = "EQ",
        AttributeValueList = { new AttributeValue { N = "1" } }
    }
);
}

// ProjectionExpression
queryRequest.ProjectionExpression = "OrderCreationDate, Product
Category, 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 re
turned\n");
    queryRequest.IndexName = indexName;

    keyConditions.Add(
        "OrderCreationDate",
        new Condition
    {
        ComparisonOperator = "GT",
        AttributeValueList = { new AttributeValue { N =
"20130131" } }
    }
);
}

```

```

        // Select
        queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
    }
    else
    {
        Console.WriteLine("\nNo index: All of Bob's orders, by OrderId:\n");
    }

    queryRequest.KeyConditions = keyConditions;

    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.Units + "\n");
}

private static void DeleteTable(string tableName)
{
    Console.WriteLine("Deleting table " + tableName + "...");
    client.DeleteTable(new DeleteTableRequest() { TableName = tableName });
    WaitForTableToDelete(tableName);
}

public static void LoadData()
{
    Console.WriteLine("Loading data into table " + tableName + "...");

    Dictionary<string, AttributeValue> item = new Dictionary<string, AttributeValue>();

    item["CustomerId"] = new AttributeValue { S = "alice@example.com" };
    item["OrderId"] = new AttributeValue { N = "1" };
    item["IsOpen"] = new AttributeValue { N = "1" };
    item["OrderCreationDate"] = new AttributeValue { N = "20130101" };
}

```

```

        item["ProductCategory"] = new AttributeValue { S = "Book" };
        item["ProductName"] = new AttributeValue { S = "The Great Outdoors" };
    };

    item["OrderStatus"] = new AttributeValue { S = "PACKING ITEMS" };
    /* no ShipmentTrackingId attribute */
    PutItemRequest putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue { S = "alice@example.com" };
};

item["OrderId"] = new AttributeValue { N = "2" };
item["IsOpen"] = new AttributeValue { N = "1" };
item["OrderCreationDate"] = new AttributeValue { N = "20130221" };

item["ProductCategory"] = new AttributeValue { S = "Bike" };
item["ProductName"] = new AttributeValue { S = "Super Mountain" };

item["OrderStatus"] = new AttributeValue { S = "ORDER RECEIVED" };

/* no ShipmentTrackingId attribute */
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue { S = "alice@example.com" };
};

item["OrderId"] = new AttributeValue { N = "3" };
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue { N = "20130304" };

item["ProductCategory"] = new AttributeValue { S = "Music" };
item["ProductName"] = new AttributeValue { S = "A Quiet Interlude" };
};

item["OrderStatus"] = new AttributeValue { S = "IN TRANSIT" };
item["ShipmentTrackingId"] = new AttributeValue { S = "176493" };
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue { S = "bob@example.com" };

item["OrderId"] = new AttributeValue { N = "1" };

```

```

/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue { N = "20130111" };

item["ProductCategory"] = new AttributeValue { S = "Movie" };
item["ProductName"] = new AttributeValue { S = "Calm Before The
Storm" };
item["OrderStatus"] = new AttributeValue { S = "SHIPPING DELAY" };

item["ShipmentTrackingId"] = new AttributeValue { S = "859323" };
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue { S = "bob@example.com" };

item["OrderId"] = new AttributeValue { N = "2" };
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue { N = "20130124" };

item["ProductCategory"] = new AttributeValue { S = "Music" };
item["ProductName"] = new AttributeValue { S = "E-Z Listening" };
item["OrderStatus"] = new AttributeValue { S = "DELIVERED" };
item["ShipmentTrackingId"] = new AttributeValue { S = "756943" };
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue { S = "bob@example.com" };

item["OrderId"] = new AttributeValue { N = "3" };
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue { N = "20130221" };

item["ProductCategory"] = new AttributeValue { S = "Music" };
item["ProductName"] = new AttributeValue { S = "Symphony 9" };
item["OrderStatus"] = new AttributeValue { S = "DELIVERED" };
item["ShipmentTrackingId"] = new AttributeValue { S = "645193" };
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue { S = "bob@example.com" };

```

```

        item["OrderId"] = new AttributeValue { N = "4" };
        item["IsOpen"] = new AttributeValue { N = "1" };
        item["OrderCreationDate"] = new AttributeValue { N = "20130222" };

        item["ProductCategory"] = new AttributeValue { S = "Hardware" };
        item["ProductName"] = new AttributeValue { S = "Extra Heavy Hammer" };
    };
    item["OrderStatus"] = new AttributeValue { S = "PACKING ITEMS" };
    /* no ShipmentTrackingId attribute */
    putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue { S = "bob@example.com" };

    item["OrderId"] = new AttributeValue { N = "5" };
    /* no IsOpen attribute */
    item["OrderCreationDate"] = new AttributeValue { N = "20130309" };

    item["ProductCategory"] = new AttributeValue { S = "Book" };
    item["ProductName"] = new AttributeValue { S = "How To Cook" };
    item["OrderStatus"] = new AttributeValue { S = "IN TRANSIT" };
    item["ShipmentTrackingId"] = new AttributeValue { S = "440185" };
    putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue { S = "bob@example.com" };

    item["OrderId"] = new AttributeValue { N = "6" };
    /* no IsOpen attribute */
    item["OrderCreationDate"] = new AttributeValue { N = "20130318" };

    item["ProductCategory"] = new AttributeValue { S = "Luggage" };
    item["ProductName"] = new AttributeValue { S = "Really Big Suitcase" };
};

    item["OrderStatus"] = new AttributeValue { S = "DELIVERED" };
    item["ShipmentTrackingId"] = new AttributeValue { S = "893927" };
    putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue { S = "bob@example.com" };
}

```

```

        item["OrderId"] = new AttributeValue { N = "7" };
        /* no IsOpen attribute */
        item["OrderCreationDate"] = new AttributeValue { N = "20130324" };

        item["ProductCategory"] = new AttributeValue { S = "Golf" };
        item["ProductName"] = new AttributeValue { S = "PGA Pro II" };
        item["OrderStatus"] = new AttributeValue { S = "OUT FOR DELIVERY"
    };
    item["ShipmentTrackingId"] = new AttributeValue { S = "383283" };
    putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
            status = res.Table.TableStatus;
        }
        catch (ResourceNotFoundException)
        {
            // DescribeTable is eventually consistent. So you might
            // get resource not found. So we handle the potential exception.
        }
    } while (status != "ACTIVE");
}

private static void WaitForTableToDelete(string tableName)
{
    bool tablePresent = true;

    while (tablePresent)
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {

```

```
        TableName = tableName
    });

    Console.WriteLine("Table name: {0}, status: {1}",
        res.Table.TableName,
        res.Table.TableStatus);
}
catch (ResourceNotFoundException)
{
    tablePresent = false;
}
}
}
}
```

Working with Local Secondary Indexes Using the AWS SDK for PHP Low-Level API

Topics

- [Create a Table With a Local Secondary Index \(p. 340\)](#)
- [Query a Local Secondary Index \(p. 342\)](#)
- [Example: Local Secondary Indexes Using the AWS SDK for PHP Low-Level API \(p. 343\)](#)

You can use the AWS SDK for PHP 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 DynamoDB API. For more information, see [Using the DynamoDB API \(p. 477\)](#).

The following are the common steps for table operations using the PHP low-level API.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `query` operation to the client instance.

You must provide the table name, index name, any desired item's primary key values, and any optional query parameters. You can set up a condition as part of the query if you want to find a range of values that meet specific comparison results. You can limit the results to a subset to provide pagination of the results. Read results are eventually consistent by default. If you want, you can request that read results be strongly consistent instead.

3. Load the response into a local variable, such as `$response`, for use in your application.

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` API and provide your specifications for one or more local secondary indexes. The following PHP code snippet creates a table to hold information about songs in a music collection. The hash key is `Artist` and the range 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 PHP low-level API.

1. Create an instance of the `DynamoDbClient` class.
2. Provide the parameters for the `createTable` operation to the client instance.

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 attribute definitions for the index range key, the key schema for the index, and the attribute projection.

The following PHP code snippet demonstrates the preceding steps. The snippet creates a table (*MusicCollection*) with a secondary index on the *AlbumTitle* attribute. The table hash and range key, plus the index range key, are the only attributes projected into the index.

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableName = 'MusicCollection';

$result = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Artist',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'SongTitle',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'AlbumTitle',
            'AttributeType' => 'S'
        )
    ),
    'KeySchema' => array(
        array(
            'AttributeName' => 'Artist',
            'KeyType' => 'HASH'
        ),
        array(
            'AttributeName' => 'SongTitle',
            'KeyType' => 'RANGE'
        )
    ),
    'LocalSecondaryIndexes' => array(
        array(
            'IndexName' => 'AlbumTitleIndex',
            'KeySchema' => array(
                array(
                    'AttributeName' => 'Artist',
                    'KeyType' => 'HASH'
                ),
                array(
                    'AttributeName' => 'AlbumTitle',
                    'KeyType' => 'RANGE'
                )
            ),
            'Projection' => array(
                'ProjectionType' => 'INCLUDE',
            )
        )
    )
));
```

```

        'NonKeyAttributes' => array('Genre', 'Year')
    )
)
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 5,
    'WriteCapacityUnits' => 5
)
));

```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Query a Local Secondary Index

You can use the `Query` API 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 range key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index range 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 \(p. 305\)](#)

The following are the steps to query a local secondary index using the P low-level API.

1. Create an instance of the `DynamoDbClient` class (the client).
2. Provide the parameters for the `query` operation to the client instance.

You must provide the table name, the index name, the key conditions for the query, and the attributes that you want returned.

The following PHP code snippet demonstrates the preceding steps.

```

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));

$tableName='MusicCollection';

$response = $client->query(array(
    'TableName' => $tableName,
    'IndexName' => 'AlbumTitleIndex',
    'KeyConditions' => array(
        'Artist' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array(
                array('S' => 'Acme Band')
            )
        ),
        'AlbumTitle' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array(
                array('S' => 'Songs About Life')
            )
        )
    )
));

```

```

        )
),
'Select' => 'ALL_ATTRIBUTES'
));

echo "Acme Band's Songs About Life:" . PHP_EOL;
foreach($response['Items'] as $item) {
    echo "    - " . $item['SongTitle'][ 'S' ] . PHP_EOL;
}

```

Example: Local Secondary Indexes Using the AWS SDK for PHP Low-Level API

The following PHP code example shows how to work with local secondary indexes. The example creates a table named *CustomerOrders* with a hash key attribute of CustomerId and a range key attribute of OrderId. There are two local secondary indexes on this table:

- *OrderCreationDateIndex*—the range key is OrderCreationDate, and the following attributes are projected into the index:
 - ProductCategory
 - ProductName
 - OrderStatus
 - ShipmentTrackingId
- *IsOpenIndex*—the range 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.

```

<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

$tableName = 'CustomerOrders';
echo "# Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'CustomerId',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'OrderId',
            'AttributeType' => 'N'
        ),
        array(

```

```

        'AttributeName' => 'OrderCreationDate',
        'AttributeType' => 'N'
    ),
    array(
        'AttributeName' => 'IsOpen',
        'AttributeType' => 'N'
    )
),
'KeySchema' => array(
    array(
        'AttributeName' => 'CustomerId',
        'KeyType' => 'HASH'
    ),
    array(
        'AttributeName' => 'OrderId',
        'KeyType' => 'RANGE'
    )
),
'LocalSecondaryIndexes' => array(
    array(
        'IndexName' => 'OrderCreationDateIndex',
        'KeySchema' => array(
            array(
                'AttributeName' => 'CustomerId',
                'KeyType' => 'HASH'
            ),
            array(
                'AttributeName' => 'OrderCreationDate',
                'KeyType' => 'RANGE'
            )
        ),
        'Projection' => array(
            'ProjectionType' => 'INCLUDE',
            'NonKeyAttributes' => array('ProductCategory', 'ProductName')
        )
    ),
    array(
        'IndexName' => 'IsOpenIndex',
        'KeySchema' => array(
            array(
                'AttributeName' => 'CustomerId',
                'KeyType' => 'HASH'
            ),
            array(
                'AttributeName' => 'IsOpen',
                'KeyType' => 'RANGE'
            )
        ),
        'Projection' => array(
            'ProjectionType' => 'ALL'
        )
    )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 5,
    'WriteCapacityUnits' => 5
)
));

```

```

echo " Waiting for table $tableName to be created." . PHP_EOL;
$client->waitForTableExists(array('TableName' => $tableName));
echo " Table $tableName has been created." . PHP_EOL;

#####
# Add items to the table

echo "# Loading data into $tableName..." . PHP_EOL;

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'alice@example.com'),
        'OrderId' => array ('N' => '1'),
        'IsOpen' => array ('N' => '1'),
        'OrderCreationDate' => array ('N' => '20140101'),
        'ProductCategory' => array ('S' => 'Book'),
        'ProductName' => array ('S' => 'The Great Outdoors'),
        'OrderStatus' => array ('S' => 'PACKING ITEMS')
    )
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'alice@example.com'),
        'OrderId' => array ('N' => '2'),
        'IsOpen' => array ('N' => '1'),
        'OrderCreationDate' => array ('N' => '20140221'),
        'ProductCategory' => array ('S' => 'Bike'),
        'ProductName' => array ('S' => 'Super Mountain'),
        'OrderStatus' => array ('S' => 'ORDER RECEIVED')
    )
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'alice@example.com'),
        'OrderId' => array ('N' => '3'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140304'),
        'ProductCategory' => array ('S' => 'Music'),
        'ProductName' => array ('S' => 'A Quiet Interlude'),
        'OrderStatus' => array ('S' => 'IN TRANSIT'),
        'ShipmentTrackingId' => array ('N' => '176493')
    )
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '1'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140111'),
        'ProductCategory' => array ('S' => 'Movie'),

```

```

        'ProductName' => array ('S' => 'Calm Before The Storm'),
        'OrderStatus' => array ('S' => 'SHIPPING DELAY'),
        'ShipmentTrackingId' => array ('N' => '859323')
    )
)
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '2'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140124'),
        'ProductCategory' => array ('S' => 'Music'),
        'ProductName' => array ('S' => 'E-Z Listening'),
        'OrderStatus' => array ('S' => 'DELIVERED'),
        'ShipmentTrackingId' => array ('N' => '756943')
    )
)
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '3'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140221'),
        'ProductCategory' => array ('S' => 'Music'),
        'ProductName' => array ('S' => 'Symphony 9'),
        'OrderStatus' => array ('S' => 'DELIVERED'),
        'ShipmentTrackingId' => array ('N' => '645193')
    )
)
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '4'),
        'IsOpen' => array ('N' => '1'),
        'OrderCreationDate' => array ('N' => '20140222'),
        'ProductCategory' => array ('S' => 'Hardware'),
        'ProductName' => array ('S' => 'Extra Heavy Hammer'),
        'OrderStatus' => array ('S' => 'PACKING ITEMS')
    )
)
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '5'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140309'),
        'ProductCategory' => array ('S' => 'Book'),
        'ProductName' => array ('S' => 'How To Cook'),
        'OrderStatus' => array ('S' => 'IN TRANSIT'),
        'ShipmentTrackingId' => array ('N' => '440185')
    )
)
);

```

```

        )
    ) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '6'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140318'),
        'ProductCategory' => array ('S' => 'Luggage'),
        'ProductName' => array ('S' => 'Really Big Suitcase'),
        'OrderStatus' => array ('S' => 'DELIVERED'),
        'ShipmentTrackingId' => array ('N' => '893927')
    )
) );

$response = $client->putItem ( array (
    'TableName' => $tableName,
    'Item' => array (
        'CustomerId' => array ('S' => 'bob@example.com'),
        'OrderId' => array ('N' => '7'),
        // no IsOpen attribute
        'OrderCreationDate' => array ('N' => '20140324'),
        'ProductCategory' => array ('S' => 'Golf'),
        'ProductName' => array ('S' => 'PGA Pro II'),
        'OrderStatus' => array ('S' => 'OUT FOR DELIVERY'),
        'ShipmentTrackingId' => array ('N' => '383283')
    )
) );

#####
# Query for Bob's 5 most recent orders in 2014, retrieving attributes which are
# projected into the index

$response = $client->query(array(
    'TableName' => $tableName,
    'IndexName' => 'OrderCreationDateIndex',
    'KeyConditions' => array(
        'CustomerId' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array(
                array('S' => 'bob@example.com')
            )
        )
    ),
    'OrderCreationDate' => array(
        'ComparisonOperator' => 'GE',
        'AttributeValueList' => array(
            array('N' => '20140101')
        )
    )
),
    'Select' => 'ALL_PROJECTED_ATTRIBUTES',
    'ScanIndexForward' => false,
    'ConsistentRead' => true,
    'Limit' => 5,
    'ReturnConsumedCapacity' => 'TOTAL'
)
);

```

```

));
echo "# Querying for Bob's 5 most recent orders in 2014:" . PHP_EOL;
foreach($response['Items'] as $item) {
    echo ' - ' . $item['CustomerId']['S'] . ' ' . $item['OrderCreationDate']['N']
    . '
        . $item['ProductName']['S'] . ' ' . $item['ProductCategory']['S'] .
PHP_EOL;
}
echo ' Provisioned Throughput Consumed: ' . $response['ConsumedCapacity']['CapacityUnits'] . PHP_EOL;

#####
# Query for Bob's 5 most recent orders in 2014, retrieving some attributes which
are not projected into the index

$response = $client->query(array(
    'TableName' => $tableName,
    'IndexName' => 'OrderCreationDateIndex',
    'KeyConditions' => array(
        'CustomerId' => array(
            'ComparisonOperator' => 'EQ',
            'AttributeValueList' => array(
                array('S' => 'bob@example.com')
            )
        ),
        'OrderCreationDate' => array(
            'ComparisonOperator' => 'GE',
            'AttributeValueList' => array(
                array('N' => '20140101')
            )
        )
    ),
    'Select' => 'SPECIFIC_ATTRIBUTES',
    'ProjectionExpression' => 'CustomerId, OrderCreationDate, ProductName,
ProductName, OrderStatus',
    'ScanIndexForward' => false,
    'ConsistentRead' => true,
    'Limit' => 5,
    'ReturnConsumedCapacity' => 'TOTAL'
));
echo "# Querying for Bob's 5 most recent orders in 2014:" . PHP_EOL;
foreach($response['Items'] as $item) {
    echo ' - ' . $item['CustomerId']['S'] . ' ' . $item['OrderCreationDate']['N']
    . '
        . $item['ProductName']['S'] . ' ' . $item['ProductCategory']['S'] . '
        . $item['OrderStatus']['S'] . PHP_EOL;
}
echo ' Provisioned Throughput Consumed: ' . $response['ConsumedCapacity']['CapacityUnits'] . PHP_EOL;

#####
# Query for Alice's open orders, fetching all attributes (which are already
projected into the index)

$response = $client->query(array(
    'TableName' => $tableName,

```

```
'IndexName' => 'IsOpenIndex',
'KeyConditions' => array(
    'CustomerId' => array(
        'ComparisonOperator' => 'EQ',
        'AttributeValueList' => array(
            array('S' => 'alice@example.com')
        )
    )
),
'Select' => 'ALL_ATTRIBUTES',
'ScanIndexForward' => false,
'ConsistentRead' => true,
'Limit' => 5,
'ReturnConsumedCapacity' => 'TOTAL'
));
echo "# Querying for Alice's open orders:" . PHP_EOL;
foreach($response['Items'] as $item) {
    echo ' - ' . $item['CustomerId']['S'] . ' ' . $item['OrderCreationDate']['N']
    .
    .
    .
    . $item['ProductName']['S'] . ' ' . $item['ProductCategory']['S'] . ' '
    . $item['OrderStatus']['S'] . PHP_EOL;
}
echo ' Provisioned Throughput Consumed: ' . $response['ConsumedCapacity']['CapacityUnits'] . PHP_EOL;

#####
# Delete the table

echo "# Deleting table $tableName..." . PHP_EOL;
$client->deleteTable(array('TableName' => $tableName));

$client->waitForTableNotExists(array('TableName' => $tableName));
echo " Deleted table $tableName..." . PHP_EOL;
?>
```

Best Practices for DynamoDB

This section is a summary of best practices for working with Amazon DynamoDB. Use this as a reference 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 \(p. 59\)](#)
- [Understand Partition Behavior \(p. 61\)](#)
- [Use Burst Capacity Sparingly \(p. 62\)](#)
- [Distribute Write Activity During Data Upload \(p. 62\)](#)
- [Understand Access Patterns for Time Series Data \(p. 63\)](#)
- [Cache Popular Items \(p. 64\)](#)
- [Consider Workload Uniformity When Adjusting Provisioned Throughput \(p. 64\)](#)
- [Test Your Application At Scale \(p. 65\)](#)

Item Best Practices

DynamoDB items are limited in size (see [Limits in DynamoDB \(p. 597\)](#)). 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 \(p. 106\)](#)
- [Use Multiple Tables to Support Varied Access Patterns \(p. 107\)](#)
- [Compress Large Attribute Values \(p. 108\)](#)
- [Store Large Attribute Values in Amazon S3 \(p. 108\)](#)
- [Break Up Large Attributes Across Multiple Items \(p. 109\)](#)

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.

- [Avoid Sudden Bursts of Read Activity \(p. 189\)](#)
- [Take Advantage of Parallel Scans \(p. 191\)](#)

Local Secondary Index Best Practices

Local secondary indexes let you define alternate range keys on a table. You can then issue Query requests against those range keys, in addition to the table's hash key. 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 \(p. 314\)](#)
- [Choose Projections Carefully \(p. 314\)](#)
- [Optimize Frequent Queries To Avoid Fetches \(p. 315\)](#)
- [Take Advantage of Sparse Indexes \(p. 315\)](#)
- [Watch For Expanding Item Collections \(p. 315\)](#)

Global Secondary Index Best Practices

Global secondary indexes let you define alternate key attributes for a table; these attributes don't have to be the same as the table's key attributes. You can issue Query requests against the global secondary index key fields, just as you would when querying 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 \(p. 271\)](#)
- [Take Advantage of Sparse Indexes \(p. 271\)](#)
- [Use a Global Secondary Index For Quick Lookups \(p. 272\)](#)
- [Create an Eventually Consistent Read Replica \(p. 272\)](#)

DynamoDB Streams Preview

Note

DynamoDB Streams is a new feature of DynamoDB, and is currently in preview mode.

Amazon DynamoDB Streams maintains a time ordered sequence of item level changes in any DynamoDB table in a log for a duration of 24 hours. Using the Streams APIs, developers can query the updates, receive the item level data before and after the changes, and use it to build creative extensions to their applications built on top of DynamoDB. For example, a developer building a global multi-player game using DynamoDB can leverage the Streams APIs to build a multi-master topology and keep the masters in sync by consuming the Streams for each master and replaying the updates in the remote masters. As another example, developers can use the Streams APIs to build mobile applications that automatically notify the mobile devices of all friends in a circle as soon as a user uploads a new picture.

Accessing the DynamoDB Streams Preview

The DynamoDB Streams preview is being made available in two different ways:

- A preview version of DynamoDB Local with support for the DynamoDB Streams API. This is the fastest way to begin using the DynamoDB Streams API. The preview version of DynamoDB Local is available for download using these links:
 - .tar.gz format: http://dynamodb-preview.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest_preview.tar.gz
 - .zip format: http://dynamodb-preview.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest_preview.zip

For information on using DynamoDB Local, see [DynamoDB Local \(p. 508\)](#) in this document.

- A set of preview endpoints for DynamoDB and DynamoDB Streams. Access to the preview endpoints is by request only. To request access, fill out the request form at this link:
 - <https://aws.amazon.com/dynamodb/preview/>

For the preview endpoints, the following limits are in effect:

- The number of tables per account is limited to 50.
- The provisioned throughput limits are:
 - Per table – maximum of 50 read capacity units and 50 write capacity units
 - Per account – maximum of 50 read capacity units and 50 write capacity units

- The `ListStreams` and `DescribeStream` API calls in DynamoDB Streams are limited to 10 per second.

These limits supersede the "Provisioned throughput limits" and "Tables per account" items listed in the [Limits in DynamoDB \(p. 597\)](#) section of this document.

To obtain SDKs, DynamoDB Local, and other preview releases of libraries and tools, go to the [DynamoDB Streams Developer Guide](#).

For More Information...

The documentation for the DynamoDB Streams preview is available at these links:

- [DynamoDB Streams Developer Guide](#)—detailed information about DynamoDB Streams, including links to the tools you'll need for writing applications.
- [DynamoDB Streams API Reference](#)—describes the API actions available in DynamoDB Streams.

DynamoDB Console

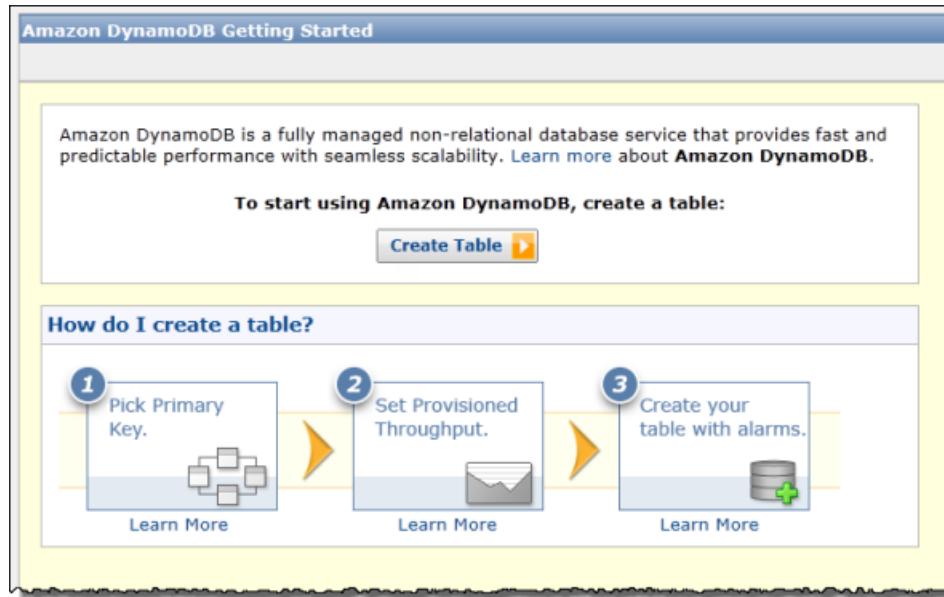
Topics

- [Working with Items and Attributes \(p. 356\)](#)
- [Monitoring Tables \(p. 362\)](#)
- [Setting Up CloudWatch Alarms \(p. 362\)](#)
- [Exporting and Importing Data \(p. 363\)](#)

The AWS Management Console for Amazon DynamoDB is available at <https://console.www.amazonaws.cn/dynamodb/home>. The console enables you to do the following:

- Create, update, and delete tables. The throughput calculator provides you with estimates of how many capacity units you will need to request based on the usage information you provide.
- View items stored in a tables, add, update, and delete items.
- Query a table.
- Set up alarms to monitor your table's capacity usage.
- View your table's top monitoring metrics on real-time graphs from CloudWatch.
- View alarms configured for each table and create custom alarms.

If your account doesn't already have tables in DynamoDB, the console displays an introductory screen that prompts you to create your first table. This screen also provides an overview of the process for creating a table, and links to relevant documentation and resources.



You can find detailed steps for creating your first table in the console in [Getting Started with DynamoDB \(p. 13\)](#). Once you have one or more tables, the console displays the tables as a list. You can select a table from the list to see additional information in the lower pane. In the lower pane, you also have the option to set up alarms and view CloudWatch metrics for the table.

The screenshot shows the 'Tables' page in the Amazon DynamoDB console. It lists four tables: Forum, ProductCatalog, Reply, and Thread. Each table row includes columns for Name, Status, Hash Key, Range Key, Read Throughput, and Write Throughput. Below the table list is a 'Table Items' section with tabs for 'Details', 'Monitoring' (which is selected), and 'Alarm Setup'. Under 'Monitoring', it shows 'CloudWatch alarms: No alarms configured' and 'Alarm History (past 24 hours): No alarms triggered'. It also displays 'CloudWatch metrics: Times are displayed in UTC.' with three line graphs: 'Consumed Read Capacity (Count)', 'Consumed Write Capacity (count)', and 'Get Latency (Milliseconds)'. A 'Time Range' dropdown is set to 'Last Hour'.

The **Explore Table** option enables you to view existing items in a table, add new items, update items, or delete items. You can also query the table. For more information, see [Working with Items and Attributes \(p. 356\)](#).

Amazon DynamoDB Explore Table: Reply			
List Tables		Browse Items	
<input checked="" type="radio"/> Scan <input type="radio"/> Query		Go	New Edit Copy to New Details Delete Export to .csv
	Id	ReplyDateTime	Message
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-17T16:45:04.833Z	DynamoDB Thread
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-24T16:45:04.833Z	DynamoDB Thread
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-01T16:45:04.833Z	DynamoDB Thread
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-07T16:45:04.833Z	DynamoDB Thread

Working with Items and Attributes

Topics

- [Adding an Item \(p. 356\)](#)
- [Deleting an Item \(p. 359\)](#)
- [Updating an Item \(p. 359\)](#)
- [Copying an Item \(p. 360\)](#)

You can use the DynamoDB console to manipulate items and attributes in a table.

Adding an Item

You can upload a large number of items programmatically. However, the console provides a way for you to upload an item without having to write any code.

To add an item

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. Select the **Reply** table and click **Explore Table**.

Amazon DynamoDB Tables				
Filter:		Explore Table	Create Table	Modify Throughput
Name	Status	Hash Key	Range Key	
Forum	ACTIVE	Name	-	
ProductCatalog	ACTIVE	Id	-	
Reply	ACTIVE	Id	ReplyDateTime	

3. On the **Browse Items** tab, click **New**.

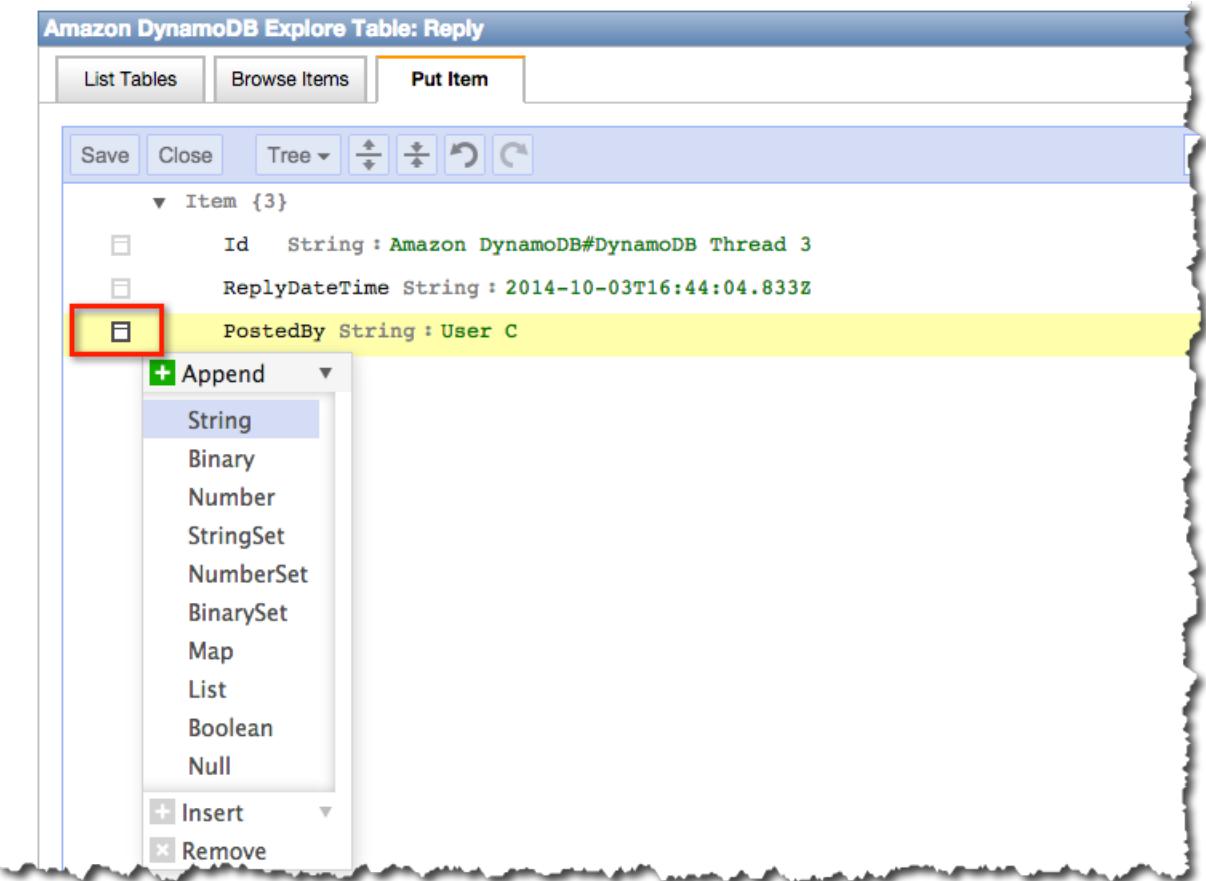
Amazon DynamoDB Explore Table: Reply			
List Tables		Browse Items	
<input checked="" type="radio"/> Scan <input type="radio"/> Query		Go	New
	Id	ReplyDateTime	Message
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-17T16:45:04.833Z	DynamoDB Thread
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-24T16:45:04.833Z	DynamoDB Thread
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-01T16:45:04.833Z	DynamoDB Thread
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-07T16:45:04.833Z	DynamoDB Thread

4. The **Put Item** tab shows data entry fields for you to enter the required primary key attribute values. If the table has any secondary indexes, then the console also shows data entry fields for index key values.

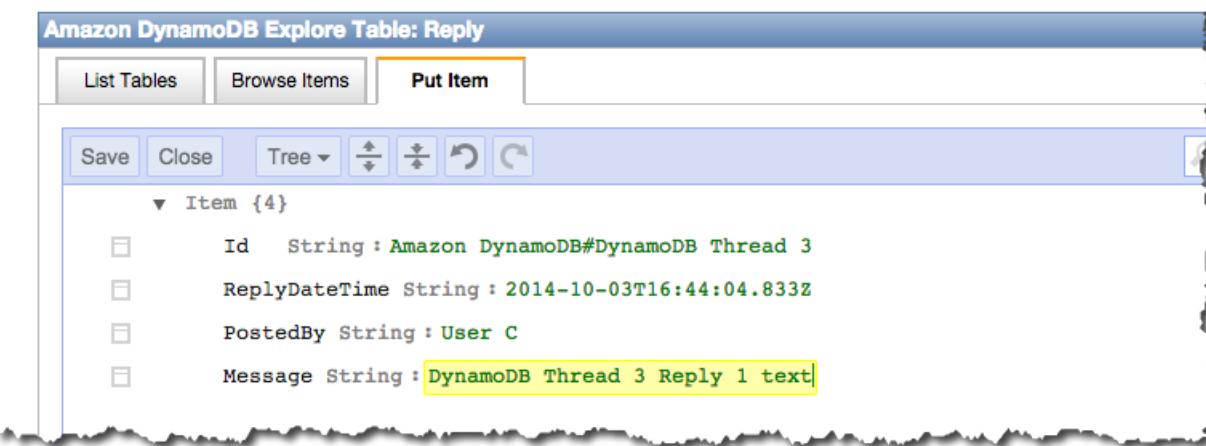
The following screen shot shows the primary key attributes of the **Reply** table (**Id** and **ReplyDateTime**) and **PostedByIndex** (**PostedBy**).

Amazon DynamoDB Explore Table: Reply			
List Tables		Browse Items	
Put Item			
Save	Close	Tree ▾	▼
▼ Item {3}			
<input type="checkbox"/>	Id	String :	<input type="text" value="VALUE"/>
<input type="checkbox"/>	ReplyDateTime	String :	<input type="text" value="VALUE"/>
<input type="checkbox"/>	PostedBy	String :	<input type="text" value="VALUE"/>

5. If you want to add more attributes, click the action menu to the left of **PostedBy**. In the action menu, click **Append**, and then click the data type you want.



Type the new attribute name and value in the fields provided.



- Repeat this step as often as needed if you want to add more attributes.
6. When the item is as you want it, click **Save** to add the new item to the table.

Deleting an Item

You can delete one item at a time using the console.

To delete an item

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. In the **Tables** pane, select a table and click **Explore Table**.

The screenshot shows the 'Amazon DynamoDB Tables' interface. At the top, there's a navigation bar with tabs: 'Explore Table' (which is highlighted with a red box), 'Create Table', 'Modify Throughput', and 'Delete Table'. Below the navigation bar is a table with four columns: 'Name', 'Status', 'Hash Key', and 'Range Key'. There are three rows in the table: 'Forum' (Status: ACTIVE, Hash Key: Name, Range Key: -), 'ProductCatalog' (Status: ACTIVE, Hash Key: Id, Range Key: -), and 'Reply' (Status: ACTIVE, Hash Key: Id, Range Key: ReplyDateTime). The 'Reply' row is also highlighted with a red box.

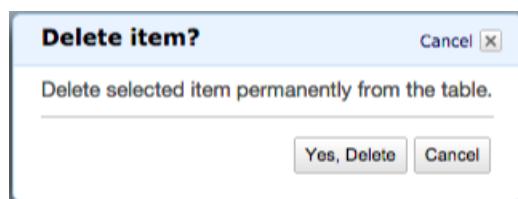
Name	Status	Hash Key	Range Key
Forum	ACTIVE	Name	-
ProductCatalog	ACTIVE	Id	-
Reply	ACTIVE	Id	ReplyDateTime

3. In the **Browse Items** tab, click on the item in the table that you want to remove, and click the **Delete Item** button.

The screenshot shows the 'Amazon DynamoDB Explore Table: Reply' interface. At the top, there are tabs: 'List Tables' and 'Browse Items' (which is highlighted with a blue background). Below the tabs are buttons: 'Scan' (radioed), 'Query', 'Go', 'New', 'Edit', 'Copy to New', 'Details', 'Delete' (which is highlighted with a red box), and 'Export to .csv'. The main area is a table with columns: 'Id', 'ReplyDateTime', and 'Message'. The first row, which contains the value 'Amazon DynamoDB#DynamoDB' in the 'Id' column, has a checked checkbox in the first column and is highlighted with a red box. The 'Delete' button is also highlighted with a red box.

	Id	ReplyDateTime	Message
<input checked="" type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-17T16:45:04.833Z	DynamoDB
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-24T16:45:04.833Z	DynamoDB
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-01T16:45:04.833Z	DynamoDB
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-07T16:45:04.833Z	DynamoDB

4. In the **Delete Item?** dialog box, click **Yes, Delete**.

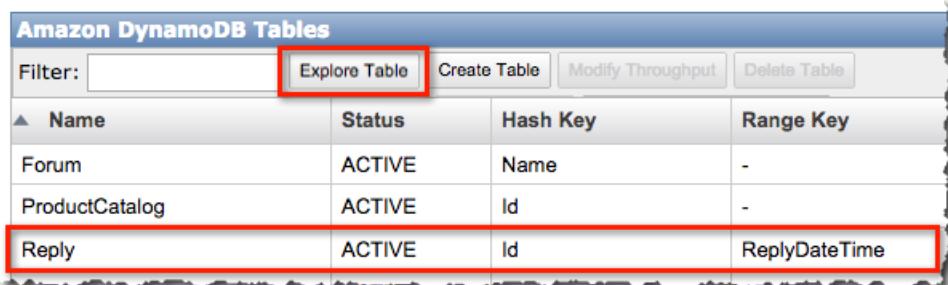


Updating an Item

You can update an item through the DynamoDB console using the **Browse Items** tab.

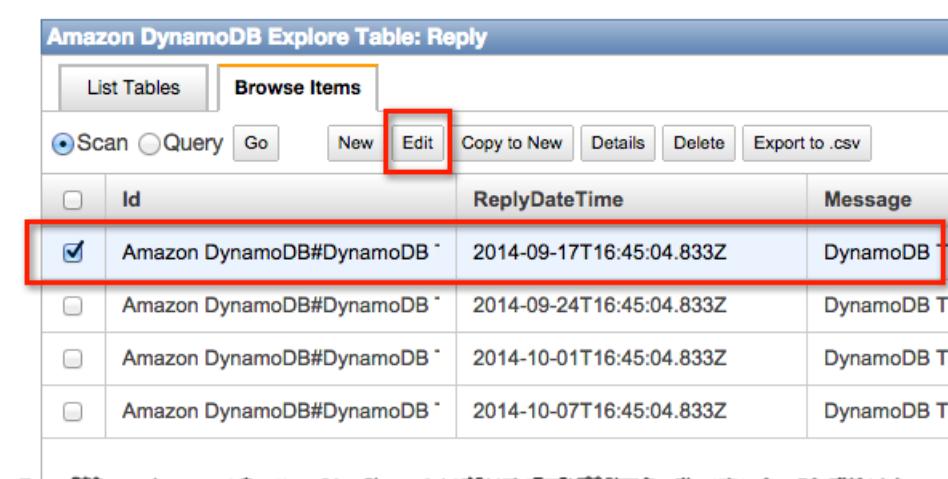
To update an item

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. In the **Tables** pane, select a table and click **Explore Table**.



Name	Status	Hash Key	Range Key
Forum	ACTIVE	Name	-
ProductCatalog	ACTIVE	Id	-
Reply	ACTIVE	Id	ReplyDateTime

3. In the **Browse Items** tab, click on the item in the table that you want to update, and click the **Edit Item** button.



	Id	ReplyDateTime	Message
<input checked="" type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-17T16:45:04.833Z	DynamoDB Th
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-09-24T16:45:04.833Z	DynamoDB Th
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-01T16:45:04.833Z	DynamoDB Th
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB	2014-10-07T16:45:04.833Z	DynamoDB Th

4. Change the desired attributes or values, and click **Save**.

You will be returned to the **Browse Items** tab. To view the data that you updated, you will need to refresh the display; to do this, click the **Scan** radio button and then click **Go**. Be aware that the operation will consume the same number of throughput capacity units that any other full-table scan consumes.

Copying an Item

You can use an existing item to create a new item through the DynamoDB console.

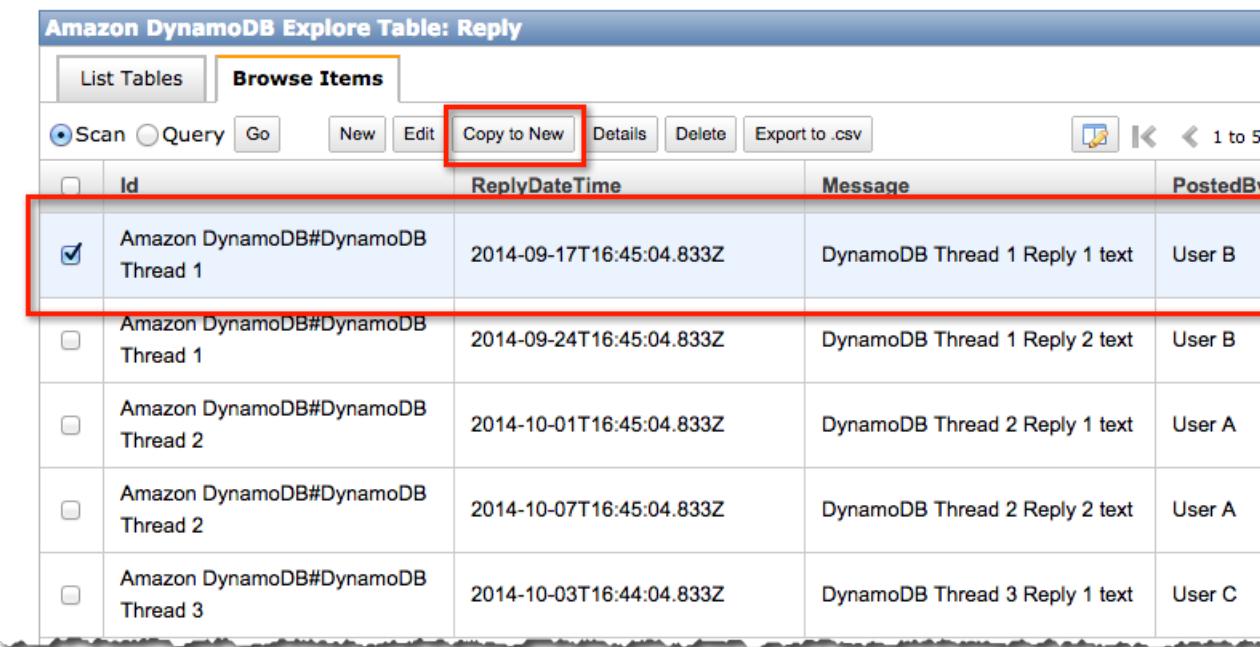
To copy and save a new item

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. In the **Tables** pane, select a table and click **Explore Table**.



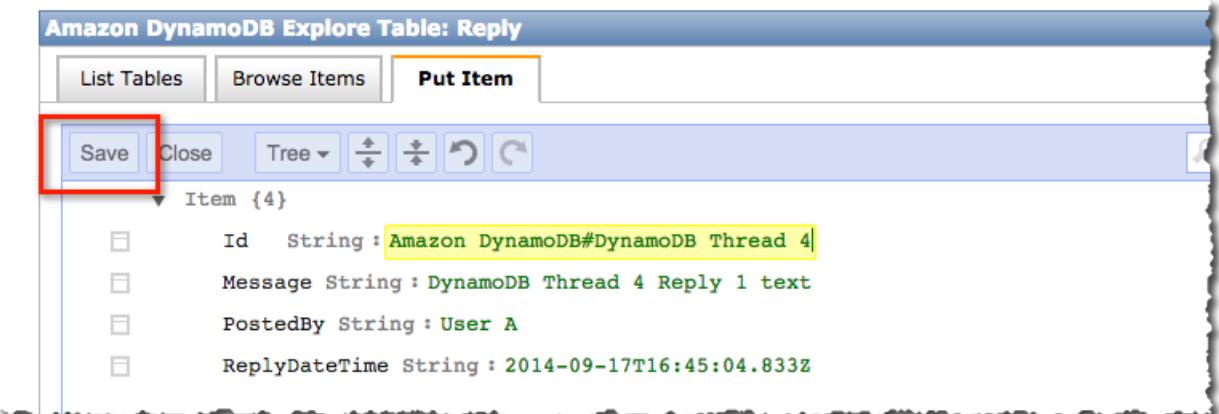
Name	Status	Hash Key	Range Key
Forum	ACTIVE	Name	-
ProductCatalog	ACTIVE	Id	-
Reply	ACTIVE	Id	ReplyDateTime

3. In the **Browse Items** tab, click on the item in the table that you want to copy, and click the **Copy to New** button.



	Id	ReplyDateTime	Message	PostedBy
<input checked="" type="checkbox"/>	Amazon DynamoDB#DynamoDB Thread 1	2014-09-17T16:45:04.833Z	DynamoDB Thread 1 Reply 1 text	User B
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB Thread 1	2014-09-24T16:45:04.833Z	DynamoDB Thread 1 Reply 2 text	User B
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB Thread 2	2014-10-01T16:45:04.833Z	DynamoDB Thread 2 Reply 1 text	User A
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB Thread 2	2014-10-07T16:45:04.833Z	DynamoDB Thread 2 Reply 2 text	User A
<input type="checkbox"/>	Amazon DynamoDB#DynamoDB Thread 3	2014-10-03T16:44:04.833Z	DynamoDB Thread 3 Reply 1 text	User C

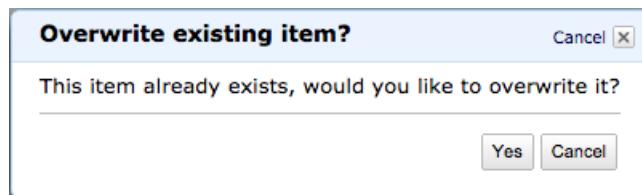
4. Change the hash and/or range key to avoid overwriting the original item, and change other attributes or values as desired. When you are ready to add the new item to the table, click **Save**.



Item {4}

- Id String : Amazon DynamoDB#DynamoDB Thread 4
- Message String : DynamoDB Thread 4 Reply 1 text
- PostedBy String : User A
- ReplyDateTime String : 2014-09-17T16:45:04.833Z

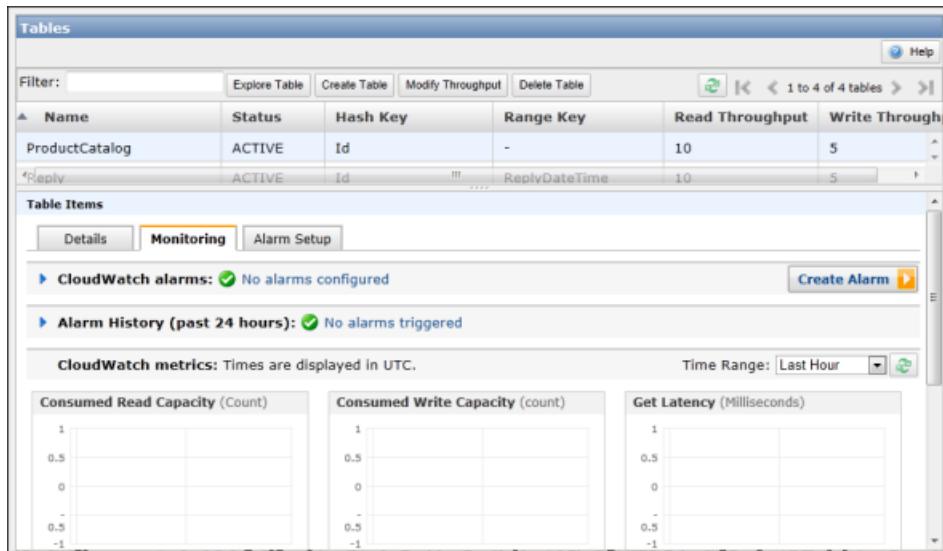
If you forget to update the hash and/or range keys, you'll see a warning that you're about to overwrite the original item. Click **Cancel**, update the keys, and click **Save** to save the changes.



Monitoring Tables

The console for DynamoDB displays some metrics for your table in the lower pane. If you need to see other metrics, you can use the console for DynamoDB to set parameters for CloudWatch to display information about your table.

To see the CloudWatch metrics for your table, with your table selected, click the **Monitoring** tab. You can expand the **CloudWatch alarms** and **Alarm History** sections to see more details about alarms that have been triggered. Also, you can view your CloudWatch metrics on this tab in convenient, real-time graphs.



Note

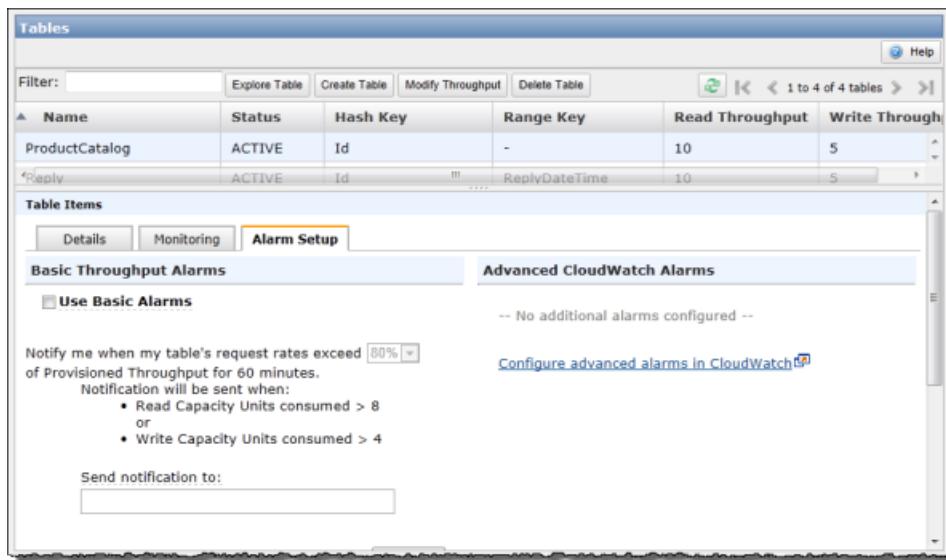
CloudWatch metrics do appear in real-time in the console. However, DynamoDB updates the Storage Size value approximately only every six hours. Recent changes might not be reflected in this value.

For more information about CloudWatch metrics for DynamoDB, see [Monitoring DynamoDB with CloudWatch \(p. 519\)](#).

Setting Up CloudWatch Alarms

When you create a table in the console, you have the option to set up a provisioned throughput alarm while the table is being created. Once a table is created, you can add more CloudWatch alarms using the lower pane of the console.

To add more alarms to your table, click the **Alarm Setup** tab.



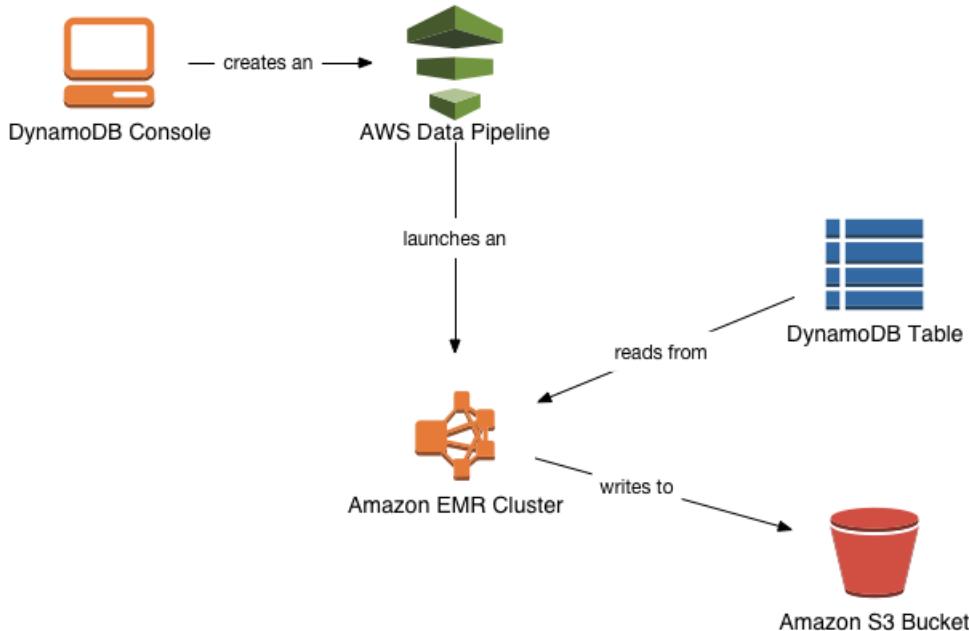
When you make a selection in the "Advanced CloudWatch Alarms" section of the **Alarm Setup** tab, you are redirected to the CloudWatch console. For more information about setting up alarms, see the CloudWatch Help in the CloudWatch console, or go to the [CloudWatch Documentation](#).

Exporting and Importing Data

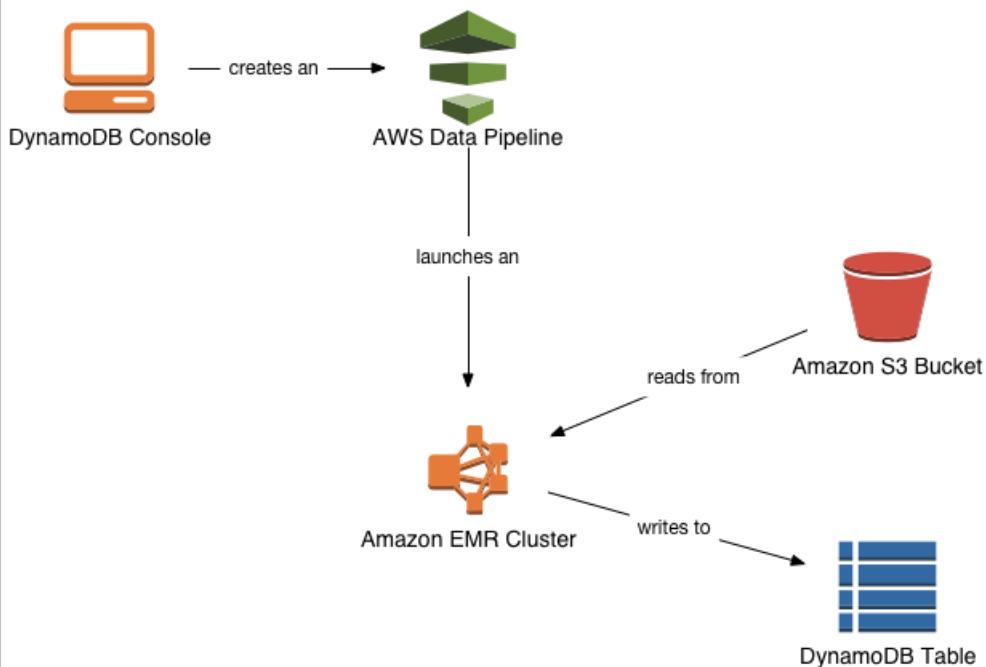
You can use the AWS Management Console to export data from DynamoDB into Amazon S3, and to import data from Amazon S3 into DynamoDB. The work of performing the exports and imports are offloaded to AWS Data Pipeline and Amazon Elastic MapReduce, while the AWS Management Console gives you an intuitive interface for moving data among tables.

For more information, see [Using the AWS Management Console to Export and Import Data \(p. 550\)](#).

Exporting Data from DynamoDB to Amazon S3



Importing Data from Amazon S3 to DynamoDB



Using the AWS SDKs with DynamoDB

Topics

- [Using the AWS SDK for Java \(p. 365\)](#)
- [Using the AWS SDK for .NET \(p. 368\)](#)
- [Using the AWS SDK for PHP \(p. 371\)](#)

AWS provides SDKs for you to develop applications for Amazon DynamoDB. The AWS SDKs for Java, PHP, and .NET wrap the underlying DynamoDB API and request format, simplifying your programming tasks. This section provides an overview of the AWS SDKs. This section also describes how you can test AWS SDK code samples provided in this guide.

In addition to .NET, Java, and PHP, the other AWS SDKs also support DynamoDB, including JavaScript, Python, Android, iOS, and Ruby. For links to the complete set of AWS SDKs, see [Start Developing with Amazon Web Services](#).

The AWS SDKs require an active AWS account. You should follow the steps in [Getting Started with DynamoDB \(p. 13\)](#) to get set up and familiar with using the AWS SDKs for DynamoDB.

Using the AWS SDK for Java

Topics

- [Running Java Examples for DynamoDB \(p. 367\)](#)

The AWS SDK for Java provides APIs for the DynamoDB item and table operations. The SDK allows you to choose from among three different APIs:

API	Comment
Document API	<p>The AWS SDK for Java Document API provides an intuitive interface for DynamoDB operations. With this API, you work with Tables, Items, Attributes, and other objects that closely resemble their counterparts in the database. The Document API also provides utilities for working with document data types, such as Lists and Maps. Finally, the document API makes it easy to store and retrieve JSON documents in DynamoDB.</p> <p>The following sections describe the document API and also provide working samples:</p> <ul style="list-style-type: none"> • Working with Tables Using the AWS SDK for Java Document API (p. 66) • Working with Items Using the AWS SDK for Java Document API (p. 110) • Querying Tables Using the AWS SDK for Java Document API (p. 192) • Scanning Tables Using the AWS SDK for Java Document API (p. 215) <p>The Document API resides in the com.amazonaws.services.dynamodbv2.document namespace.</p>
High-level API	<p>The high-level API uses object persistence programming techniques to map Java objects to DynamoDB tables and attributes. In this API, the <code>DynamoDBMapper</code> class provides an object-oriented view of your data: You define getter and setter methods for attributes in a table, and <code>DynamoDBMapper</code> insulates your application from low-level DynamoDB operations. You cannot create tables using the high-level API, but you can create, read, update, and delete table items. The high-level API is especially advantageous if you have an existing code-base that you want to leverage by mapping to DynamoDB tables.</p> <p>The high-level API resides in the com.amazonaws.services.dynamodbv2.datamodeling namespace.</p>
Low-level API	<p>The methods in the low-level API correspond closely to the underlying DynamoDB API. The low-level API allows you to perform the same operations that you can perform using DynamoDB operations such as create, update, and delete tables, and create, read, update, and delete items.</p> <p>The low-level API resides in the com.amazonaws.services.dynamodbv2.model namespace.</p>

Note

These APIs provide thread-safe clients for accessing DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information about the AWS SDK for Java API, go to the [AWS SDK for Java API Reference](#).

Choosing a JVM

For the best performance of your server-based applications with the AWS SDK for Java, we recommend that you use the 64-bit version of the Java Virtual Machine (JVM). This JVM runs only in Server mode, even if you specify the `-Client` option at run time. Using the 32-bit version of the JVM with the `-Server` option at run time should provide comparable performance to the 64-bit JVM.

Running Java Examples for DynamoDB

General Process of Creating Java Code Examples (Using Eclipse)

1	Download and install the AWS Toolkit for Eclipse . This toolkit includes the AWS SDK for Java, along with preconfigured templates for building applications.
2	From the Eclipse menu, click File, New, Other... . In the Select a wizard box, click File, AWS Java Project , Click Next . In the Project name field, type a name for your project. Click Finish to create the project. Note that the project is pre-configured, and includes the AWS SDK for Java <code>.jar</code> files.
3	You will now need to create a default credential profiles file. This file enhances security by storing your credentials separately from your project directories, so that they cannot be unintentionally committed to a public repository. For more information, see Using the Default Credential Provider Chain in the AWS SDK for Java Developer Guide. The credential properties file should be saved as <code>~/.aws/credentials</code> , where the tilde character represents your home directory. In this file, you can store multiple sets of credentials from any number of accounts. Each set is referred to as a profile. The following is an example of a credential properties file with a profile named <code>default</code> : <pre>[default] aws_access_key_id = <Your Access Key ID> aws_secret_access_key = <Your Secret Key></pre> The code examples in this document use the default client constructors that read your AWS credentials stored in the credential properties file.
4	Copy the code from the section that you are reading to your project.
5	Run the code.

Setting the Region

You can set the DynamoDB region explicitly, as shown in the following Java code snippet.

```
client = new AmazonDynamoDBClient(credentials);
client.setRegion(Region.getRegion(Regions.US_WEST_2));
```

For a current list of supported regions and endpoints, see [Regions and Endpoints](#).

Using the AWS SDK for .NET

Topics

- [Running .NET Examples for DynamoDB \(p. 369\)](#)

The AWS SDK for .NET provides the following APIs to work with DynamoDB. All the APIs are available in the AWSSDK.dll. For information about downloading the AWS SDK for .NET, go to [Sample Code Libraries](#).

Note

The low-level API and high-level API provide thread-safe clients for accessing DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

API	Comment
Low-level API	<p>This is the protocol level API that maps closely to the DynamoDB API. You can use the low-level API for all table and item operations such as create, update, delete table and items. You can also query and scan your tables.</p> <p>This API is available in the <code>Amazon.DynamoDB.DataModel</code> namespace.</p> <p>The following sections describe the low-level API in various AWS SDKs and also provide working samples:</p> <ul style="list-style-type: none">• Working with Tables in DynamoDB (p. 54)• Working with Items in DynamoDB (p. 85)• Query and Scan Operations in DynamoDB (p. 183)• Improving Data Access with Secondary Indexes in DynamoDB (p. 241)
Document Model API	<p>This API provides wrapper classes around the low-level API to further simplify your programming task. The <code>Table</code> and <code>Document</code> are the key wrapper classes. You can use the document model for the data operations such as create, retrieve, update and delete items. To create, update and delete tables, you must use the low-level API.</p> <p>The API is available in the <code>Amazon.DynamoDB.DocumentModel</code> namespace.</p> <p>For more information about the document model API and working samples, see .NET: Document Model (p. 410).</p>

API	Comment
Object Persistence API	<p>The object persistence API enables you to map your client-side classes to the DynamoDB tables. Each object instance then maps to an item in the corresponding tables. The <code>DynamoDBContext</code> class in this API provides methods for you to save client-side objects to a table, retrieve items as objects and perform query and scan.</p> <p>You can use the object persistence model for the data operations such as create, retrieve, update and delete items. You must first create your tables using the low-level API and then use the object persistence model to map your classes to the tables.</p> <p>The API is available in the <code>Amazon.DynamoDB.DataModel</code> namespace.</p> <p>For more information about the object persistence API and working samples, see .NET: Object Persistence Model (p. 441).</p>

To configure the .NET CLR

To ensure the best performance of your server-based applications on systems with multiple processors or processor cores, we recommend that you enable server mode garbage collection (GC). Note that without multiple processors or processor cores, server mode GC has no effect.

To enable server mode GC, add the following to your `app.config` file:

```
<runtime>
    <gcServer enabled="true"/>
    <gcConcurrent enabled="true"/>
</runtime>
```

Running .NET Examples for DynamoDB

General Process of Creating .NET Code Examples (Using Visual Studio)

1	Download and install the AWS SDK for .NET . This toolkit includes the AWS .NET library and the AWS Toolkit for Visual Studio, along with preconfigured templates for building applications.
2	Create a new Visual Studio project using the <i>AWS Empty Project</i> template. You get this template if you installed the Toolkit for Visual Studio. For more information, see Step 1: Before You Begin (p. 13) . The AWS Access Credentials dialog box opens.

3	<p>In the AWS Access Credentials dialog box, select either an account that you previously added to the toolkit, or add a new account. For each account that you add to the toolkit, you must provide your AWS access keys credentials.</p> <p>Visual Studio saves your credentials in the SDK Store. The SDK Store enhances security by storing your credentials separately from your project directories, so that they cannot be unintentionally committed to a public repository. For more information, see Setting Up the AWS Toolkit for Visual Studio in the AWS Toolkit for Visual Studio User Guide.</p> <p>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:</p> <pre style="border: 1px solid black; padding: 5px; font-family: monospace;"> <?xml version="1.0" encoding="utf-8" ?> <configuration> <appSettings> <add key="AWSProfileName" value="default" /> <add key="AWSRegion" value="us-west-2" /> </appSettings> </configuration></pre> <p>The code examples in this document use the default client constructors that read your AWS credentials stored in the SDK Store.</p>
4	Note that the <i>AWS Empty Project</i> template includes the required AWSSDK reference.
5	Replace the code in the project file, Program.cs, with the code in the section you are reading.
6	Run the code.

Setting the Endpoint

You can set the DynamoDB endpoint explicitly, as shown in the following C# code snippet.

```

private static void CreateClient()
{
    AmazonDynamoDBConfig config = new AmazonDynamoDBConfig();
    config.ServiceURL = "http://dynamodb.us-west-2.amazonaws.com";
    var client = new AmazonDynamoDBClient(config);
}
```

You can also set the DynamoDB endpoint explicitly from the app.config file, as shown in the following C# code snippet.

```

private static void CreateClient()
{
    AmazonDynamoDBConfig config = new AmazonDynamoDBConfig();
    config.ServiceURL = System.Configuration.ConfigurationManager.AppSettings["ServiceURL"];
    client = new AmazonDynamoDBClient(config);
}
```

For a current list of supported regions and endpoints, see [Regions and Endpoints](#).

Using the AWS SDK for PHP

The AWS SDK for PHP currently has one level of API (called the "low-level" API) for DynamoDB that maps directly to the service's native API.

The SDK is available at [AWS SDK for PHP](#), which also has instructions for installing and getting started with the SDK.

Note

The setup for using the AWS SDK for PHP depends on your environment and how you want to run your application. To set up your environment to run the examples in this documentation, see the [AWS SDK for PHP Getting Started Guide](#).

Running PHP Examples

The general process for running a PHP code example is as follows. The individual steps are explained in greater detail in the following sections.

General Process of Creating PHP Code Examples

1	Download and install the AWS SDK for PHP , and then verify that your environment meets the minimum requirements as described in the AWS SDK for PHP Getting Started Guide .
2	Install the AWS SDK for PHP according to the instructions in the AWS SDK for PHP Getting Started Guide . Depending on the installation method that you use, you might have to modify your code to resolve dependencies among the PHP extensions. The PHP code samples in this document use the Composer dependency manager that is described in the AWS SDK for PHP Getting Started Guide .
3	Copy the example code from the document to your project.
4	Configure your credentials provider for the AWS SDK for PHP. For more information, go to Providing Credentials to the SDK in the AWS SDK for PHP Getting Started Guide .
5	Test the example according to your setup.

Setting Your AWS Access Keys

For security reasons, we recommend that you create a credential profile that contains your AWS access key ID and secret key. This approach will let you avoid hardcoding your access keys in the PHP code itself. At run time, when you create a new `DynamoDb` object, the client can obtain the keys from the configuration file.

The following is an example of using an AWS credentials file and a credential profile. The credentials file is named `~/.aws/credentials`, where `~` represents your HOME directory.

```
[default]
aws_access_key_id = AWS access key ID goes here
aws_secret_access_key = Secret key goes here
```

Here is how to use the credential profile to instantiate a new client:

```
use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
));
```

For more information, go to [Providing Credentials to the SDK](#) in the [AWS SDK for PHP Getting Started Guide](#)

Setting the Region

If you are using a credential profile, you can set the default region for your client, as in the following example:

```
use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' #replace with your desired region
));
```

For a current list of supported regions and endpoints, see [Regions and Endpoints](#).

Higher-Level Programming Interfaces for DynamoDB

The AWS SDKs provide applications with low-level interfaces for working with Amazon DynamoDB. These client-side API classes and methods correspond directly to the DynamoDB server-side 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 API, 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 APIs with higher levels of abstraction, in addition to the low-level APIs. 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 APIs 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: Object Persistence Model \(p. 373\)](#)

.NET

- [.NET: Document Model \(p. 410\)](#)
- [.NET: Object Persistence Model \(p. 441\)](#)

Java: Object Persistence Model

Topics

- [Supported Data Types \(p. 376\)](#)
- [Java Annotations for DynamoDB \(p. 377\)](#)
- [The DynamoDBMapper Class \(p. 381\)](#)

- Optimistic Locking With Version Number (p. 390)
- Mapping Arbitrary Data (p. 392)
- Example: CRUD Operations (p. 395)
- Example: Batch Write Operations (p. 397)
- Example: Query and Scan (p. 402)

The AWS SDK for Java provides a high-level *object persistence model*, enabling you to map your client-side classes to DynamoDB tables. The individual object instances then map to items in a table. The object persistence model provides a `DynamoDBMapper` class, which provides an entry point to DynamoDB. The `DynamoDBMapper` class provides you with a connection to the DynamoDB database and enables you to access your tables, perform various create, read, update and delete (CRUD) operations, and execute queries.

Note

The object persistence model lets you query database tables and perform item operations such as saving, updating, or deleting. However, the object persistence model does not provide APIs to create, update, or delete tables. To perform those tasks, use AWS SDK for Java Document API instead. For more information, see [Working with Tables Using the AWS SDK for Java Document API \(p. 66\)](#).

The AWS 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 hash primary 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 annotation types to map object fields to DynamoDB attribute names:

```
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 type 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 type 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. If you define a class property name that does not match a corresponding item attribute name, then you must explicitly add the `@DynamoDBAttribute` annotation type to specify the mapping. In the preceding example, the `@DynamoDBAttribute` annotation type is added to each property to ensure that the property names match exactly with the tables created in the *Getting Started* section, 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 type. In the preceding example, the `SomeProp` property is marked with the `@DynamoDBIgnore` annotation type. 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:

```

AmazonDynamoDBClient client = new AmazonDynamoDBClient(new ProfileCredentialsProvider());
DynamoDBMapper mapper = new DynamoDBMapper(client);

CatalogItem item = new CatalogItem();
item.setId(102);
item.setTitle("Book 102 Title");
item.setISBN("222-2222222222");
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 hashKeyValue = new CatalogItem();

hashKeyValue.setId(102);
DynamoDBQueryExpression<CatalogItem> queryExpression = new DynamoDBQueryExpres

```

```
sion<CatalogItem>()
    .withHashKeyValues(hashKeyValues);

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());
}
```

The object persistence model 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 hash and range keys, 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	N (number type)
Strings	S (string type)
boolean	N (number type), 0 or 1.
ByteBuffer	B (binary type)
Date	S (string type). The Date values are stored as ISO-8601 formatted strings.
Set collection types	SS (string set) type, NS (number set) type, or BS (binary set) type.

In addition, DynamoDB supports arbitrary data types. For example, you can define your own complex types on the client. You use the `DynamoDBMarshaller` interface and the `@DynamoDBMarshalling` annotation type for the complex type to describe the mapping ([Mapping Arbitrary Data \(p. 392\)](#)).

Java Annotations for DynamoDB

The following table 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 table, only the `DynamoDBTable` and the `DynamoDBHashKey` are the required tags.

Declarative Tag (Annotation)	Description
<code>@DynamoDBTable</code>	<p>Identifies the target table in DynamoDB. For example, the following Java code snippet defines a class <code>Developer</code> and maps it to the <code>People</code> table in DynamoDB.</p> <pre style="border: 1px solid black; padding: 5px;">@DynamoDBTable(tableName="People") public class Developer { ... }</pre> <p>This annotation can be inherited or overridden.</p> <ul style="list-style-type: none"> The <code>@DynamoDBTable</code> annotation can be inherited. Any new class that inherits from the <code>Developer</code> class also maps to the <code>People</code> table. For example, assume that you create a <code>Lead</code> class that inherits from the <code>Developer</code> class. Because you mapped the <code>Developer</code> class to the <code>People</code> table, the <code>Lead</code> class objects are also stored in the same table. The <code>@DynamoDBTable</code> can also be overridden. Any new class that inherits from the <code>Developer</code> class by default maps to the same <code>People</code> table. However, you can override this default mapping. For example, if you create a class that inherits from the <code>Developer</code> class, you can explicitly map it to another table by adding the <code>@DynamoDBTable</code> annotation as shown in the following Java code snippet. <pre style="border: 1px solid black; padding: 5px;">@DynamoDBTable(tableName="Managers") public class Manager : Developer { ... }</pre>
<code>@DynamoDBIgnore</code>	<p>Indicates to the <code>DynamoDBMapper</code> instance that the associated property should be ignored. When saving data to the table, the <code>DynamoDBMapper</code> does not save this property to the table.</p>

Declarative Tag (Annotation)	Description
@DynamoDBAttribute	<p>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, using this tag you can map a property to the attribute. In the following Java snippet, the <code>DynamoDBAttribute</code> maps the <code>BookAuthors</code> property to the <code>Authors</code> attribute name in the table.</p> <pre style="border: 1px solid black; padding: 5px;">@DynamoDBAttribute(attributeName = "Authors") public List<String> getBookAuthors() { return BookAuthors; } public void setBookAuthors(List<String> BookAuthors) { this.BookAuthors = BookAuthors; }</pre> <p>The <code>DynamoDBMapper</code> uses <code>Authors</code> as the attribute name when saving the object to the table.</p>
@DynamoDBHashKey	<p>Maps a class property to the hash attribute of the table. The property must be one of the scalar string, number or binary types; it cannot be a collection type.</p> <p>Assume that you have a table, <code>ProductCatalog</code>, that has <code>Id</code> as the primary key. The following Java code snippet defines a <code>CatalogItem</code> class and maps its <code>Id</code> property to the primary key of the <code>ProductCatalog</code> table using the <code>@DynamoDBHashKey</code> tag.</p> <pre style="border: 1px solid black; padding: 5px;">@DynamoDBTable(tableName="ProductCatalog") public class CatalogItem { private String Id; @DynamoDBHashKey(attributeName="Id") public String getId() { return Id; } public void setId(String Id) { this.Id = Id; } // Additional properties go here. }</pre>

Declarative Tag (Annotation)	Description
@DynamoDBRangeKey	<p>Maps a class property to the range key attribute of the table. The property must be one of the scalar string, number or binary types; it cannot be a collection type.</p> <p>If the primary key is made of both the hash and range key attributes, you can use this tag to map your class field to the range attribute. 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 hash attribute and ReplyDateTime is the range attribute. 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.</p> <pre style="font-family: monospace; padding-left: 20px;"> @DynamoDBTable(tableName="Reply") public class Reply { private String id; private String replyDateTime; @DynamoDBHashKey(attributeName="Id") public String getId() { return id; } public void setId(String id) { this.id = id; } @DynamoDBRangeKey(attributeName="ReplyDate Time") public String getReplyDateTime() { return replyDateTime; } public void setReplyDateTime(String replyDat eTime) { this.replyDateTime = replyDateTime; } // Additional properties go here. } </pre>

Declarative Tag (Annotation)	Description
<code>@DynamoDBAutoGeneratedKey</code>	<p>Marks a hash key or range key property as being auto-generated. The object persistence model will generate a random UUID when saving these attributes. Only String properties can be marked as auto-generated keys.</p> <p>The following snippet demonstrates using auto-generated keys.</p> <pre style="background-color: #f0f0f0; padding: 10px;">@DynamoDBTable(tableName="AutoGeneratedKeyExample") 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 String 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()); } }</pre>
<code>@DynamoDBVersionAttribute</code>	<p>Identifies a class property for storing an optimistic locking version number. <code>DynamoDBMapper</code> 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 DynamoDB Data Types (p. 6). For more information about versioning, see Optimistic Locking With Version Number (p. 390).</p>

Declarative Tag (Annotation)	Description
@DynamoDBIndexHashKey	<p>Maps a class property to the hash attribute of a global secondary index. The property must be one of the scalar string, number or binary types; it cannot be a collection type.</p> <p>Use this annotation type if you need to Query a global secondary index. You must specify the index name (<code>globalSecondaryIndexName</code>). If the name of the class property is different from the index hash key attribute, you must also specify the name of that index attribute (<code>attributeName</code>).</p>
@DynamoDBIndexRangeKey	<p>Maps a class property to the range attribute 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.</p> <p>Use this annotation type if you need to Query a local secondary index or a global secondary index and want to refine your results using the index range key. You must specify the index name (either <code>globalSecondaryIndexName</code> or <code>localSecondaryIndexName</code>). If the name of the class property is different from the index range key attribute, you must also specify the name of that index attribute (<code>attributeName</code>).</p>
@DynamoDBMarshalling	<p>Identifies a class property that uses a custom marshaller. When used with the <code>DynamoDBMarshaller</code> class, this annotation 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. 392).</p>

The DynamoDBMapper Class

The `DynamoDBMapper` class is the entry point to DynamoDB. It provides a connection to DynamoDB 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 key operations for you to work with DynamoDB.

For the corresponding Javadoc documentation, see [DynamoDBMapper](#) in the [AWS SDK for Java API Reference](#).

Method	Description
save	<p>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 <code>DynamoDBMapperConfig</code> object.</p> <p>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. String hash and range keys annotated with <code>@DynamoDBAutoGeneratedKey</code> are given a random universally unique identifier (UUID) if left uninitialized. Version fields annotated with <code>@DynamoDBVersionAttribute</code> 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.</p> <p>By default, only attributes corresponding to mapped class properties are updated; any additional existing attributes on an item are unaffected. However, if you specify <code>SaveBehavior.CLOBBER</code>, you can force the item to be completely overwritten.</p> <pre>mapper.save(obj, new DynamoDBMapperConfig(DynamoDBMapperConfig.SaveBehavior.CLOBBER));</pre> <p>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 <code>SaveBehavior.CLOBBER</code> option is used. For more information about versioning, see Optimistic Locking With Version Number (p. 390).</p>
load	<p>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 <code>DynamoDBMapperConfig</code> 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.</p> <pre>CatalogItem item = mapper.load(CatalogItem.class, item.getId(), new DynamoDBMapperConfig(DynamoDBMapperConfig.ConsistentReads.CONSTANT));</pre> <p>By default, DynamoDB returns the item that has values that are eventually consistent. For information about the eventual consistency model of DynamoDB, see Data Read and Consistency Considerations (p. 9).</p>
delete	<p>Deletes an item from the table. You must pass in an object instance of the mapped class.</p> <p>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 <code>SaveBehavior.CLOBBER</code> option is used. For more information about versioning, see Optimistic Locking With Version Number (p. 390).</p>

Method	Description
query	

Method	Description
	<p>Queries a table. You can query a table only if its primary key is made of both a hash and a range attribute. This method requires you to provide a hash attribute value and a query filter that is applied on the range attribute. A filter expression includes a condition and a value.</p> <p>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 hash attribute and ReplyDateTime is the range attribute of the primary key.</p> <pre style="border: 1px solid black; padding: 5px;">Reply (<u>Id</u>, <u>ReplyDateTime</u>, ...)</pre> <p>Now, assume that you created an object persistence model that includes a Reply class that maps to the table.</p> <p>The following Java code snippet uses the <code>DynamoDBMapper</code> instance to query the table to find all replies in the past two weeks for a specific thread subject.</p> <pre style="border: 1px solid black; padding: 5px;">String forumName = "DynamoDB"; String forumSubject = "DynamoDB Thread 1"; String hashKey = 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); Condition rangeKeyCondition = new Condition() .withComparisonOperator(ComparisonOperator.GT.toString()) .withAttributeValueList(new AttributeValue().withS(twoWeeksAgoStr.toString())); Reply replyKey = new Reply(); replyKey.setId(hashKey); DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryExpression<Reply>() .withHashKeyValues(replyKey) .withRangeKeyCondition("ReplyDateTime", rangeKeyCondition); List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);</pre> <p>The query returns a collection of <code>Reply</code> objects.</p> <p>Note If your table's primary key is made of only a hash attribute, then you cannot use the <code>query</code> method. Instead, you can use the <code>load</code> method and provide the hash attribute to retrieve the item.</p> <p>By default, the</p>

Method	Description
	<p>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 <code>latestReplies</code> collection.</p>
queryPage	<p>Queries a table and returns a single page of matching results. As with the <code>query</code> method, you must specify a hash attribute value and a query filter that is applied on the range attribute. However, <code>queryPage</code> will only return the first "page" of data - that is, the amount of data that will fit within 1 MB</p>
scan	<p>Scans an entire table. You can optionally specify one or more <code>Condition</code> instances to filter the result set, and you can specify a filter expression for any item attributes.</p> <p>Assume that you have a table, <code>Thread</code>, that stores forum thread information including <code>Subject</code> (part of the composite primary key) and if the thread is answered.</p> <pre style="border: 1px solid black; padding: 5px;">Thread (ForumName, Subject, ..., Answered)</pre> <p>If you have an object persistence model for this table, then you can use the <code>DynamoDBMapper</code> to scan the table. For example, the following Java code snippet filters the <code>Thread</code> table to retrieve all the unanswered threads. The scan condition identifies the attribute and a condition.</p> <pre style="border: 1px solid black; padding: 5px;">DynamoDBScanExpression scanExpression = new DynamoDBScanExpression(); Map<String, Condition> scanFilter = new HashMap<String, Condition>(); Condition scanCondition = new Condition() .withComparisonOperator(ComparisonOperator.EQ.toString()) .withAttributeValueList(new AttributeValue().withN("0")); scanFilter.put("Answered", scanCondition); scanExpression.setScanFilter(scanFilter); List<Thread> unansweredThreads = mapper.scan(Thread.class, scanExpression);</pre> <p>By default, the <code>scan</code> 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 <code>unansweredThreads</code> collection.</p>
scanPage	<p>Scans a table and returns a single page of matching results. As with the <code>scan</code> method, you can optionally specify one or more <code>Condition</code> instances to filter the result set, and you can specify a filter expression for any item attributes. However, <code>scanPage</code> will only return the first "page" of data - that is, the amount of data that will fit within 1 MB</p>

Method	Description
parallelScan	<p>Performs a parallel scan of an entire table. You specify a number of logical segments for the table, along with a scan expression to filter the results. The <code>parallelScan</code> divides the scan task among multiple workers, one for each logical segment; the workers process the data in parallel and return the results.</p> <p>The following Java code snippet performs a parallel scan on the <code>Product</code> table.</p> <pre>int numberOfThreads = 4; DynamoDBScanExpression scanExpression = new DynamoDBScanExpression(); scanExpression.addFilterCondition("Price", new Condition() .withComparisonOperator(ComparisonOperator.GT) .withAttributeValueList(new AttributeValue() .withN("100"))); List<Product> scanResult = mapper.parallelScan(Product.class, scanExpression, numberOfThreads);</pre> <p>For a Java code sample illustrating usage of <code>parallelScan</code>, see Example: Query and Scan (p. 402).</p>
batchSave	<p>Saves objects to one or more tables using one or more calls to the <code>AmazonDynamoDB.batchWriteItem</code> method. This method does not provide transaction guarantees.</p> <p>The following Java code snippet saves two items (books) to the <code>ProductCatalog</code> table.</p> <pre>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));</pre>

Method	Description
batchLoad	<p>Retrieves multiple items from one or more tables using their primary keys.</p> <p>The following Java code snippet retrieves two items from two different tables.</p> <pre>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);</pre>
batchDelete	<p>Deletes objects from one or more tables using one or more calls to the <code>AmazonDynamoDB.batchWriteItem</code> method. This method does not provide transaction guarantees.</p> <p>The following Java code snippet deletes two items (books) from the <code>ProductCatalog</code> table.</p> <pre>Book book1 = mapper.load(Book.class, 901); Book book2 = mapper.load(Book.class, 902); mapper.batchDelete(Arrays.asList(book1, book2));</pre>
batchWrite	<p>Saves objects to and deletes objects from one or more tables using one or more calls to the <code>AmazonDynamoDB.batchWriteItem</code> method. This method does not provide transaction guarantees or support versioning (conditional puts or deletes).</p> <p>The following Java code snippet writes a new item to the <code>Forum</code> table, writes a new item to the <code>Thread</code> table, and deletes an item from the <code>ProductCatalog</code> table.</p> <pre>// 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);</pre>

Method	Description
count	Evaluates the specified scan expression and returns the count of matching items. No item data is returned.
generate-CreateTableRequest	Parses a POJO class that represents a DynamoDB table, and returns a CreateTableRequest for that table.
createS3Link	<p>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.</p> <p>To use <code>createS3Link</code>, 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 <code>CatalogItem</code> class:</p> <pre style="border: 1px solid black; padding: 10px;"> @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; } ... }</pre>
	<p>The following Java code defines a new item to be written to the <code>Product</code> table. The item includes a link to a product image; the image data is uploaded to Amazon S3.</p> <pre style="border: 1px solid black; padding: 10px;"> 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);</pre>
	<p>The <code>S3Link</code> class provides many other methods for manipulating objects in Amazon S3. For more information, see the Javadocs for <code>S3Link</code>.</p>

Method	Description
getS3ClientCache	Returns the underlying <code>S3ClientCache</code> for accessing Amazon S3. An <code>S3ClientCache</code> is a smart Map for <code>AmazonS3Client</code> objects. If you have multiple clients, then an <code>S3ClientCache</code> can help you keep the clients organized by region, and can create new Amazon S3 clients on demand.

DynamoDBMapperConfig: Optional Configuration Settings for DynamoDBMapper

The object persistence model provides the `DynamoDBMapper` for you to communicate with DynamoDB. When you create a mapper instance, 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:

```
ClasspathPropertiesFileCredentialsProvider cp =
    new ClasspathPropertiesFileCredentialsProvider();

AmazonDynamoDBClient client = new AmazonDynamoDBClient(cp);

DynamoDBMapperConfig mapperConfig = new DynamoDBMapperConfig(
    DynamoDBMapperConfig.SaveBehavior.CLOBBER,
    DynamoDBMapperConfig.ConsistentReads.CONSTANT,
    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.

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 type. In the mapping class for your table, you designate one property to store the version number, and mark it using this annotation type. 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.

For example, the following Java code snippet defines a `CatalogItem` class that has several properties. The `Version` property is tagged with the `@DynamoDBVersionAttribute` annotation type.

```
@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 `@DynamoDBVersion` annotation to nullable types provided by the primitive wrappers classes such as `Long` and `Integer`, or you can use the primitive types `int` and `long`. We recommend that you use `Integer` and `Long` whenever possible.

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.

Note that the internal implementation of optimistic locking in the object persistence code uses the conditional update and the conditional delete API support in 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 [DynamoDBMapperConfig: Optional Configuration Settings for DynamoDBMapper \(p. 389\)](#).

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.

```

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 \(p. 376\)](#)), 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 an instance of String and vice-versa, and annotate the complex type accessor method using the `@DynamoDBMarshalling` annotation type. The converter code transforms data when objects are saved or loaded. It is also used for all operations that consume complex types. Note that when comparing data during query and scan operations, the comparisons are made against the data stored in DynamoDB.

For example, consider the following `CatalogItem` class that defines a property, `Dimension`, that is of `DimensionType`. This property stores the item dimensions, as height, width, and thickness. Assume that you decide to store these item dimensions as a string (such as `8.5x11x.05`) in DynamoDB. The following example provides converter code that converts the `DimensionType` object to a string and a string to the `DimensionType`.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;  
  
import java.io.IOException;  
import java.util.Arrays;  
import java.util.HashSet;  
import java.util.Set;  
  
import com.amazonaws.auth.profile.ProfileCredentialsProvider;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;  
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.DynamoDBMarshaller;  
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMarshalling;  
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;  
  
public class ObjectPersistenceMappingExample {  
  
    static AmazonDynamoDBClient client;  
  
    public static void main(String[] args) throws IOException {  
  
        AmazonDynamoDBClient client = new AmazonDynamoDBClient(new ProfileCredentialsProvider());  
  
        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);

System.out.println(book);

DynamoDBMapper mapper = new DynamoDBMapper(client);
mapper.save(book);

Book bookRetrieved = mapper.load(Book.class, 502);

System.out.println(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(bookRetrieved);

}

@DynamoDBTable(tableName="ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private DimensionType dimensionType;

    @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; }

    @DynamoDBMarshalling(marshallerClass = DimensionTypeConverter.class)
```

```

        public DimensionType getDimensions() { return dimensionType; }
        public void setDimensions(DimensionType dimensionType) { this.dimension
Type = dimensionType; }

        @Override
        public String toString() {
            return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors
+ ", dimensionType=" + dimensionType + ", 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 DynamoDBMarshaller<Di
mensionType> {

    @Override
    public String marshall(DimensionType value) {
        DimensionType itemDimensions = (DimensionType)value;
        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 unmarshall(Class<DimensionType> dimensionType,
String value) {

        DimensionType itemDimension = new DimensionType();
        try {
            if (value != null && value.length() !=0 ) {
                String[] data = value.split("x");
                itemDimension.setLength(data[0].trim());

```

```
        itemDimension.setHeight(data[1].trim());
        itemDimension.setThickness(data[2].trim());
    }
} catch (Exception e) {
    e.printStackTrace();
}

return itemDimension;
}
}
```

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 example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 ObjectPersistenceCRUDExample {

    static AmazonDynamoDBClient client = new AmazonDynamoDBClient(new ProfileCredentialsProvider());

    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;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN; }

    @DynamoDBAttribute(attributeName = "Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors
= bookAuthors; }
    @Override
    public String toString() {
        return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors
+ ", id=" + id + ", title=" + title + "]";
    }
}

private static void testCRUDOperations() {

    CatalogItem item = new CatalogItem();
    item.setId(601);
    item.setTitle("Book 601");
    item.setISBN("611-1111111111");
    item.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Au
thor2")));

    // 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("622-2222222222");
    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(DynamoDBMapper
Config.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 object persistence model attributes.

The code example then uses the `DynamoDBMapper` to 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 [Example Tables and Data \(p. 609\)](#). For step-by-step instructions to test the following sample, see [Using the AWS SDK for Java \(p. 365\)](#).

```
package com.amazonaws.codesamples;

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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 ObjectPersistenceBatchWriteExample {

    static AmazonDynamoDBClient client = new AmazonDynamoDBClient(new ProfileCredentialsProvider());
}
```

```
    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 ObjectPersistenceBatchWrit
eExample: " + 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;  
  
    @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.product
Category = 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;

    @DynamoDBHashKey(attributeName="Id")
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    @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;

@DynamoDBHashKey(attributeName="ForumName")
public String getForumName() { return forumName; }
public void setForumName(String forumName) { this.forumName = forumName;
}

@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; }

}
```

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 [Example Tables and Data \(p. 609\)](#).

- 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 range attribute 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 range attribute. 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 example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Getting Started with DynamoDB \(p. 13\)](#) section. Alternatively, you can load the data programmatically using the instructions in the [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#) topic.

For step-by-step instructions to run the following example, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
```

```
import java.util.Date;
import java.util.List;
import java.util.Set;
import java.util.TimeZone;

import com.amazonaws.auth.AWS Credentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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;
import com.amazonaws.services.dynamodbv2.model.ComparisonOperator;
import com.amazonaws.services.dynamodbv2.model.Condition;

public class ObjectPersistenceQueryScanExample {

    static AmazonDynamoDBClient client = new AmazonDynamoDBClient(new ProfileCre-
dentialsProvider());
    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);

            // 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, numberOf
            Threads, "Road");

            System.out.println("Example complete!");

        } catch (Throwable t) {
            System.err.println("Error running the ObjectPersistenceQuery
ScanExample: " + 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 range key attribute, so you Get (but no 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 hashKey = forumName + "#" + threadSubject;
        long twoWeeksAgoMilli = (new Date()).getTime() - (15L*24L*60L*60L*1000L);

        Date twoWeeksAgo = new Date();
        twoWeeksAgo.setTime(twoWeeksAgoMilli);
        dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
        String twoWeeksAgoStr = dateFormatter.format(twoWeeksAgo);

        Condition rangeKeyCondition = new Condition()
            .withComparisonOperator(ComparisonOperator.GT.toString())
            .withAttributeValueList(new AttributeValue().withS(twoWeeksAgoStr.to
String()));

        Reply replyKey = new Reply();
        replyKey.setId(hashKey);

        DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryEx
pression<Reply>()
            .withHashKeyValues(replyKey)
            .withRangeKeyCondition("ReplyDateTime", rangeKeyCondition);

        List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);

        for (Reply reply : latestReplies) {
            System.out.format("Id=%s, Message=%s, PostedBy=%s %n, ReplyDate
Time=%s %n",
                reply.getId(), reply.getMessage(), reply.getPostedBy(),
                reply.getReplyDateTime() );
        }
    }

    private static void FindRepliesPostedWithinTimePeriod(
        DynamoDBMapper mapper,
        String forumName,
```

```
        String threadSubject) throws Exception {
    String hashKey = forumName + "#" + threadSubject;

    System.out.println("FindRepliesPostedWithinTimePeriod: Find replies for
thread Message = 'DynamoDB Thread 2' posted within a period.");
    long startDateMilli = (new Date()).getTime() - (14L*24L*60L*60L*1000L);
// Two weeks ago.
    long endDateMilli = (new Date()).getTime() - (7L*24L*60L*60L*1000L);
// One week ago.
    dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
    String startDate = dateFormatter.format(startDateMilli);
    String endDate = dateFormatter.format(endDateMilli);

    Condition rangeKeyCondition = new Condition()
        .withComparisonOperator(ComparisonOperator.BETWEEN.toString())
        .withAttributeValueList(new AttributeValue().withS(startDate),
            new AttributeValue().withS(endDate));

    Reply replyKey = new Reply();
    replyKey.setId(hashKey);

    DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryEx
pression<Reply>()
        .withHashKeyValues(replyKey)
        .withRangeKeyCondition("ReplyDateTime", rangeKeyCondition);

    List<Reply> betweenReplies = mapper.query(Reply.class, queryExpression);

    for (Reply reply : betweenReplies) {
        System.out.format("Id=%s, Message=%s, PostedBy=%s %n, PostedDate
Time=%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 Product
Catalog.");

    DynamoDBScanExpression scanExpression = new DynamoDBScanExpression();
    scanExpression.addFilterCondition("Price",
        new Condition()
            .withComparisonOperator(ComparisonOperator.LT)
            .withAttributeValueList(new AttributeValue().withN(value)));

    scanExpression.addFilterCondition("ProductCategory",
        new Condition()
            .withComparisonOperator(ComparisonOperator.EQ)
            .withAttributeValueList(new AttributeValue().withS("Book")));

    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.");

    DynamoDBScanExpression scanExpression = new DynamoDBScanExpression();
    scanExpression.addFilterCondition("ProductCategory",
        new Condition()
            .withComparisonOperator(ComparisonOperator.EQ)
            .withAttributeValueList(new AttributeValue().withS("Bi
cycle")));
    scanExpression.addFilterCondition("BicycleType",
        new Condition()
            .withComparisonOperator(ComparisonOperator.EQ)
            .withAttributeValueList(new AttributeValue().withS(bicycle
Type)));
    List<Bicycle> scanResult = mapper.parallelScan(Bicycle.class, scanEx
pression, 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.product
Category = 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 String gender;
    private Set<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="Gender")
public String getGender() { return gender; }
public void setGender(String gender) { this.gender = gender; }

@DynamoDBAttribute(attributeName="Color")
public Set<String> getColor() { return color; }
public void setColor(Set<String> color) { this.color = color; }

@DynamoDBAttribute(attributeName="ProductCategory")
public String getProductCategory() { return productCategory; }
public void setProductCategory(String productCategory) { this.product
Category = 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;

    @DynamoDBHashKey(attributeName="Id")
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    @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;

@DynamoDBHashKey(attributeName="ForumName")
public String getForumName() { return forumName; }
public void setForumName(String forumName) { this.forumName = forumName;
}

@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; }

}
```

.NET: Document Model

Topics

- Operations Not Supported by the Document Model (p. 410)
- Working with Items in DynamoDB Using the AWS SDK for .NET Document Model (p. 410)
- Getting an Item - Table.GetItem (p. 414)
- Deleting an Item - Table.DeleteItem (p. 415)
- Updating an Item - Table.UpdateItem (p. 416)
- Batch Write - Putting and Deleting Multiple Items (p. 417)
- Example: CRUD Operations Using the AWS SDK for .NET Document Model (p. 419)
- Example: Batch Operations Using AWS SDK for .NET Document Model API (p. 425)
- Querying Tables in DynamoDB Using the AWS SDK for .NET Document Model (p. 428)

The AWS SDK for .NET provides document model classes that wrap some of the low-level API (see [Working with Items Using the AWS SDK for .NET Low-Level API \(p. 132\)](#)) functionality 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. For information about tables and items, see [DynamoDB Data Model \(p. 3\)](#).

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. 411)
- Specifying Optional Parameters (p. 413)

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.

```
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 remote `DescribeTable` API 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:

```
// Get the item from a table that has a primary key that is composed of only a hash attribute.  
Table.GetItem(Primitive hashAttribute);  
// Get the item from a table whose primary key is composed of both a hash and range attribute.  
Table.GetItem(Primitive hashAttribute, Primitive rangeAttribute);
```

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:

```
// 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 hash attribute as primary key.  
Table.GetItem(Primitive hashAttribute, GetItemOperationConfig config);  
// 2. Table that has both a hash and range attribute as a primary key.  
Table.GetItem(Primitive hashAttribute, Primitive rangeAttribute, 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.

```
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:

```
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.

```
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/205_front.jpg" );
pictures.Add("RearView", "http://example.com/products/205_rear.jpg" );
pictures.Add("SideView", "http://example.com/products/205_left_side.jpg" );

item.Add("Pictures", pictures);

table.PutItem(book);
```

These examples are based on the item shown in [Case Study: A ProductCatalog Item \(p. 91\)](#). 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.

```
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:

```
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 [Data Read and Consistency Considerations \(p. 9\)](#)) 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 [Data Read and Consistency Considerations \(p. 9\)](#).

```
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:

```
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:

```
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.

```
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 hashKey = 222;  
table.DeleteItem(hashKey)
```

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.

```
Table table = Table.LoadTable(client, "ProductCatalog");  
int hashKey = 111; // Primary key.  
  
Expression expr = new Expression();  
expr.ExpressionStatement = "ISBN = :val";  
expr.ExpressionAttributeValues[":val"] = "11-11-11-11";  
  
// Specify optional parameters for Delete operation.  
DeleteItemOperationConfig config = new DeleteItemOperationConfig  
{
```

```
ConditionalExpression = expr,
ReturnValues = ReturnValues.AllOldAttributes // This is the only supported
value when using the document model.
};

// Delete the book.
Document d = table.DeleteItem(hashKey, 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` API uses the following guidelines:

- If the item does not exist, the `UpdateItem` API adds a new item using the primary key that is specified in the input.
- If the item exists, the `UpdateItem` API 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 API.

Note

The `PutItem` operation ([Putting an Item - Table.PutItem Method \(p. 411\)](#)) can also perform an update. If you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. 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, the `UpdateItem` API 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 multivalued attribute, deletes the PageCount attribute, and adds a new attribute XYZ. The `Document` instance includes the primary key of the book to update.

```
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.

```
Table table = Table.LoadTable(client, "ProductCatalog");
string hashKey = "111";

var book = new Document();
book["Id"] = hashKey;
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 API 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 the API allows, 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 \(p. 425\)](#).

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 [Using the AWS SDK for .NET \(p. 368\)](#).

```
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 hashKey = sampleBookId;

    var book = new Document();
    book["Id"] = hashKey;
    // 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 hashKey = sampleBookId;

    var book = new Document();

    book["Id"] = hashKey;

    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.AsPrimitiveL

                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. 425)

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 Getting Started section, 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 \(p. 623\)](#). For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
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 SingleTableBatch
Write()");

        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 MultiTableBatch

Write()");

superBatch.Execute();

}

}

}
```

Querying Tables in DynamoDB Using the AWS SDK for .NET Document Model

Topics

- [Table.Query Method in the AWS SDK for .NET \(p. 429\)](#)
- [Table.Scan Method in the AWS SDK for .NET \(p. 436\)](#)

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 primary key that is composed of both a hash and range attribute. If your table's primary key is made of only a hash attribute, then the `Query` operation is not supported. By default, the API internally performs queries that are eventually consistent. To learn about the consistency model, see [Data Read and Consistency Considerations \(p. 9\)](#).

The `Query` method provides two overloads. The minimum required parameters to the `Query` method are a hash key value and a range filter. You can use the following overload to provide these minimum required parameters.

```
Query(Primitive hashKey, RangeFilter Filter);
```

For example, the following C# code snippet queries for all forum replies that were posted in the last 15 days.

```
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.

```
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.

```
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.

```

Table table = Table.LoadTable(client, "Reply");
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
QueryOperationConfig config = new QueryOperationConfig()
{
    HashKey = "DynamoDB Thread 2",
    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 hash attribute, you can only get items; you cannot query the table. The example retrieves a specific product item using the item Id.

```

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, "Product Catalog");

```

```
        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 forum
Name, string threadSubject)

{

    string hashAttribute = forumName + "#" + threadSubject;

    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
```

```
        QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal,
hashAttribute);

        filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

        // Use Query overloads that takes the minimum required query para-
meters.

        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 QueryOperationCon
fig.

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()
                                              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.

```
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, a multivalued attribute for keywords, and other information. Assume that the subject is the primary key.

```
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 "rangekey". 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.

```
string tableName = "Thread";
Table ThreadTable = Table.LoadTable(client, tableName);

ScanFilter scanFilter = new ScanFilter();
scanFilter.AddCondition("ForumId", ScanOperator.Equal, 101);
scanFilter.AddCondition("Tags", ScanOperator.Contains, "rangekey");

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.

```
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 "rangekey" 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.

```
string tableName = "Thread";
Table ThreadTable = Table.LoadTable(client, tableName);

ScanFilter scanFilter = new ScanFilter();
scanFilter.AddCondition("ForumId", ScanOperator.Equal, forumId);
scanFilter.AddCondition("Tags", ScanOperator.Contains, "rangekey");

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.

```
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 pro  
ductCatalogTable)  
  
{  
  
    // 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.AsPrimitiveL

ist().Entries
```

```
ray());
    select primitive.Value).ToArray();
    Console.WriteLine("{0} - {1}", attribute, stringValue);
}
}
}
}
```

.NET: Object Persistence Model

Topics

- [DynamoDB Attributes \(p. 443\)](#)
- [DynamoDBContext Class \(p. 445\)](#)
- [Supported Data Types \(p. 450\)](#)
- [Optimistic Locking Using Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 451\)](#)
- [Mapping Arbitrary Data with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 453\)](#)
- [Batch Operations Using AWS SDK for .NET Object Persistence Model \(p. 457\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Object Persistence Model \(p. 461\)](#)
- [Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model \(p. 464\)](#)
- [Example: Query and Scan in DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 470\)](#)

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 Using the AWS SDK for .NET Low-Level API \(p. 71\)](#).

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.

```
[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 \(p. 453\)](#).

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 \(p. 451\)](#).

DynamoDB Attributes

The following table lists the attributes the object persistence model offers so you can map your classes and properties to DynamoDB tables and attributes.

Note

In the following table, only `DynamoDBTable` and `DynamoDBHashKey` are required tags.

Declarative Tag (attribute)	Description
<code>DynamoDBGlobalSecondaryIndex-HashKey</code>	Maps a class property to the hash key attribute of a global secondary index. Use this attribute if you need to <code>Query</code> a global secondary index.
<code>DynamoDBGlobalSecondaryIndexIn-dexRangeKey</code>	Maps a class property to the range key attribute of a global secondary index. Use this attribute if you need to <code>Query</code> a global secondary index and want to refine your results using the index range key.
<code>DynamoDBHashKey</code>	Maps a class property to the hash attribute of the table's primary key. The primary key attributes cannot be a collection type. The following C# code examples maps the <code>Book</code> class to the <code>ProductCatalog</code> table, and the <code>Id</code> property to the table's primary key hash attribute.
	<pre>[DynamoDBTable("ProductCatalog")] public class Book { [DynamoDBHashKey] public int Id { get; set; } // Additional properties go here. }</pre>
<code>DynamoDBIgnore</code>	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 <code>DynamoDBContext</code> not to include this property when saving objects to the table.
<code>DynamoDBLocalSecondaryIndexIn-dexRangeKey</code>	Maps a class property to the range key attribute of a local secondary index. Use this attribute if you need to <code>Query</code> a local secondary index and want to refine your results using the index range key.
<code>DynamoDBProperty</code>	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 <code>DynamoDBProperty</code> maps the <code>BookAuthors</code> property to the <code>Authors</code> attribute in the table.
	<pre>[DynamoDBProperty("Authors")] public List<string> BookAuthors { get; set; }</pre>
	DynamoDBContext uses this mapping information to create the <code>Authors</code> attribute when saving object data to the corresponding table.
<code>DynamoDBRenamable</code>	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.

Declarative Tag (attribute)	Description
DynamoDBRangeKey	<p>Maps a class property to the range attribute of the table's primary key. If the table's primary key is made of both the hash and range attributes, you must specify both the DynamoDBHashKey and DynamoDBRangeKey attributes in your class mapping.</p> <p>For example, the sample table Reply has a primary key made of the Id hash attribute and Replenishment range attribute. 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.</p> <p>For more information about sample tables, see Example Tables and Data (p. 609).</p> <div style="border: 1px solid black; padding: 10px;"><pre>[DynamoDBTable("Reply")] public class Reply { [DynamoDBHashKey] public int ThreadId { get; set; } [DynamoDBRangeKey] public string Replenishment { get; set; } // Additional properties go here. }</pre></div>

Declarative Tag (attribute)	Description
DynamoDBTable	<p>Identifies the target table in DynamoDB to which the class maps. For example, the following C# code example maps the <code>Developer</code> class to the <code>People</code> table in DynamoDB.</p> <pre>[DynamoDBTable("People")] public class Developer { ...}</pre> <p>This attribute can be inherited or overridden.</p> <ul style="list-style-type: none"> The <code>DynamoDBTable</code> attribute can be inherited. In the preceding example, if you add a new class, <code>Lead</code>, that inherits from the <code>Developer</code> class, it also maps to the <code>People</code> table. Both the <code>Developer</code> and <code>Lead</code> objects are stored in the <code>People</code> table. The <code>DynamoDBTable</code> attribute can also be overridden. In the following C# code example, the <code>Manager</code> class inherits from the <code>Developer</code> class, however the explicit addition of the <code>DynamoDBTable</code> attribute maps the class to another table (<code>Managers</code>). <pre>[DynamoDBTable("Managers")] public class Manager : Developer { ...}</pre>
	<p>You can add the optional parameter, <code>LowerCamelCaseProperties</code>, 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.</p> <pre>[DynamoDBTable("People", LowerCamelCaseProperties=true)] public class Developer { string DeveloperName; ... }</pre> <p>When saving instances of the <code>Developer</code> class, <code>DynamoDBContext</code> saves the <code>DeveloperName</code> property as the <code>developerName</code>.</p>
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. 451) .

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:

Method	Description
CreateMultiTableBatchGet	<p>Creates a <code>MultiTableBatchGet</code> object, composed of multiple individual <code>BatchGet</code> objects. Each of these <code>BatchGet</code> objects can be used for retrieving items from a single DynamoDB table.</p> <p>To retrieve the items from the table(s), use the <code>ExecuteBatchGet</code> method, passing the <code>MultiTableBatchGet</code> object as a parameter.</p>
CreateMultiTableBatchWrite	<p>Creates a <code>MultiTableBatchWrite</code> object, composed of multiple individual <code>BatchWrite</code> objects. Each of these <code>BatchWrite</code> objects can be used for writing or deleting items in a single DynamoDB table.</p> <p>To write to the table(s), use the <code>ExecuteBatchWrite</code> method, passing the <code>MultiTableBatchWrite</code> object as a parameter.</p>
CreateBatchGet	Creates a <code>BatchGet</code> object that you can use to retrieve multiple items from a table. For more information, see Batch Get: Getting Multiple Items (p. 459) .
CreateBatchWrite	Creates a <code>BatchWrite</code> 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. 457) .
Delete	<p>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.</p> <ul style="list-style-type: none"> 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. <p>Note To perform this operation in the background, use the <code>DeleteAsync</code> method instead.</p>
Dispose	Disposes of all managed and unmanaged resources.
ExecuteBatchGet	<p>Reads data from one or more tables, processing all of the <code>BatchGet</code> objects in a <code>MultiTableBatchGet</code>.</p> <p>Note To perform this operation in the background, use the <code>ExecuteBatchGetAsync</code> method instead.</p>
ExecuteBatchWrite	<p>Writes or deletes data in one or more tables, processing all of the <code>BatchWrite</code> objects in a <code>MultiTableBatchWrite</code>.</p> <p>Note To perform this operation in the background, use the <code>ExecuteBatchWriteAsync</code> method instead.</p>

Method	Description
FromDocument	<p>Given an instance of a <code>Document</code>, the <code>FromDocument</code> method returns an instance of a client-side class.</p> <p>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. 410).</p> <p>Suppose you have a <code>Document</code> object named <code>doc</code>, containing a representation of a Forum item. (To see how to construct this object, see the description for the <code>ToDocument</code> method below.) You can use <code>FromDocument</code> to retrieve the Forum item from the <code>Document</code> as shown in the following C# code snippet.</p> <pre style="border: 1px solid black; padding: 5px;">forum101 = context.FromDocument<Forum>(101);</pre> <p>Note If your <code>Document</code> object implements the <code>IEnumerable</code> interface, you can use the <code>FromDocuments</code> method instead. This will allow you to iterate over all of the class instances in the <code>Document</code>.</p>
FromQuery	<p>Executes a <code>Query</code> operation, with the query parameters defined in a <code>QueryOperationConfig</code> object.</p> <p>Note To perform this operation in the background, use the <code>FromQueryAsync</code> method instead.</p>
FromScan	<p>Executes a <code>Scan</code> operation, with the scan parameters defined in a <code>ScanOperationConfig</code> object.</p> <p>Note To perform this operation in the background, use the <code>FromScanAsync</code> method instead.</p>
GetTargetTable	<p>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.</p>
Load	<p>Retrieves an item from a table. The method requires only the primary key of the item you want to retrieve.</p> <p>By default, DynamoDB returns the item with values that are eventually consistent. For information on the eventual consistency model, see Data Read and Consistency Considerations (p. 9).</p> <p>Note To perform this operation in the background, use the <code>LoadAsync</code> method instead.</p>

Method	Description
Query	<p>Queries a table based on query parameters you provide.</p> <p>You can query a table only if its primary key is composed of both the hash and the range attributes. When querying, you must specify a hash attribute and a condition that applies to the range attribute.</p> <p>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 hash attribute and the ReplyDateTime range attribute. For more information about the Reply table, see Example Tables and Data (p. 609).</p> <pre style="border: 1px solid black; padding: 5px;">DynamoDBContext context = new DynamoDBContext(client); string replyId = "DynamoDB#DynamoDB Thread 1"; // Hash value. DateTime twoWeeksAgoDate = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0)); // Date to compare. IEnumerable<Reply> latestReplies = context.Query<Reply>(replyId, QueryOperator.GreaterThan, twoWeeksAgoDate);</pre> <p>This returns a collection of Reply objects.</p> <p>The <code>Query</code> method returns a "lazy-loaded" <code>IEnumerable</code> 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 <code>IEnumerable</code>.</p> <p>If your table's primary key consists of only a hash attribute, then you cannot use the <code>Query</code> method. Instead, you can use the <code>Load</code> method and provide the hash attribute to retrieve the item.</p> <p>Note To perform this operation in the background, use the <code>QueryAsync</code> method instead.</p>
Save	<p>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.</p> <p>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. 451).</p> <p>Note To perform this operation in the background, use the <code>SaveAsync</code> method instead.</p>

Method	Description
Scan	<p>Performs an entire table scan.</p> <p>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 <code>Book</code> mapped to the <code>ProductCatalog</code> table in DynamoDB. The following C# snippet scans the table and returns only the book items priced less than 0.</p> <pre>IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(new ScanCondition("Price", ScanOperator.LessThan, price), new ScanCondition("ProductCategory", ScanOperator.Equal, "Book"));</pre> <p>The <code>Scan</code> method returns a "lazy-loaded" <code>IEnumerable</code> 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 <code>IEnumerable</code>.</p> <p>For performance reasons you should query your tables and avoid a table scan.</p> <p>Note To perform this operation in the background, use the <code>ScanAsync</code> method instead.</p>
ToDocument	<p>Returns an instance of the <code>Document</code> document model class from your class instance.</p> <p>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. 410).</p> <p>Suppose you have a client-side class mapped to the sample <code>Forum</code> table. You can then use a <code>DynamoDBContext</code> to get an item, as a <code>Document</code> object, from the <code>Forum</code> table as shown in the following C# code snippet.</p> <pre>DynamoDBContext context = new DynamoDBContext(client); Forum forum101 = context.Load<Forum>(101); // Retrieve a forum by primary key. Document doc = context.ToDocument<Forum>(forum101);</pre>

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. For more information about read consistency, see [DynamoDB Data Model \(p. 3\)](#).
- **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 \(p. 451\)](#).
- **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.

```
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.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context = new DynamoDBContext(client);
Book bookItem = context.Load<Book>(productId,new DynamoDBContextConfig{ Consist
entRead = 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 with the following limitations:

- Collection type must implement `ICollection` interface.
- Collection type must be composed of the supported primitive types. For example, `ICollection<string>`, `ICollection<bool>`.
- Collection type must provide a parameter-less constructor.

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

.NET primitive type	DynamoDB type
All number types	N (number type)
All string types	S (string type)
MemoryStream, byte[]	B (binary type)
bool	N (number type), 0 represents false and 1 represents true.
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.

```
[DynamoDBTable("ProductCatalog")]
public class Book
```

```
{
    [DynamoDBHashKey]    // Hash 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:

- **Save**—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.
- **Delete**—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.

```
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 \(p. 449\)](#).

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.

```
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 \(p. 450\)](#)), 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, go to [Using the AWS SDK for .NET \(p. 368\)](#) in the *Amazon DynamoDB Developer Guide*.

```
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] // hash 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.

```
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.

```
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 \(p. 464\)](#).

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.

```
DynamoDBContext context = new DynamoDBContext(client);
var bookBatch = context.CreateBatchGet<ProductCatalog>();
bookBatch.AddKey(101);
bookBatch.AddKey(102);
bookBatch.AddKey(103);
bookBatch.Execute();
// Process result.
Console.WriteLine(bookBatch.Results.Count);
Book book1 = bookBatch.Results[0];
Book book2 = bookBatch.Results[1];
Book book3 = bookBatch.Results[2];
```

To retrieve objects from multiple tables, do the following:

- For each type, create an instance of the `CreateBatchGet` type and provide the primary key values you want to retrieve from each table.
- Create an instance of the `MultiTableBatchGet` class using one of the following methods:
 - 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.

```
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 [Example Tables and Data \(p. 609\)](#). For step-by-step instructions to test the following sample, go to [Using the AWS SDK for .NET \(p. 368\)](#) in the *Amazon DynamoDB Developer Guide*.

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class HighLevelItemCRUD
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                DynamoDBContext context = new DynamoDBContext(client);
                TestCRUDOperations(context);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
        }
    }
}
```

```
        catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }

        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }

        catch (Exception e) { Console.WriteLine(e.Message); }

    }

private static void TestCRUDOperations(DynamoDBContext context)
{
    int bookID = 1001; // Some unique value.

    Book myBook = new Book
    {
        Id = bookID,
        Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
        ISBN = "111-1111111001",
        BookAuthors = new List<string> { "Author 1", "Author 2" },
    };

    // Save the book.

    context.Save(myBook);

    // Retrieve the book.

    Book bookRetrieved = context.Load<Book>(bookID);

    // Update few properties.

    bookRetrieved.ISBN = "222-2222221001";

    bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author x" }; // Replace existing authors list with this.

    context.Save(bookRetrieved);

    // Retrieve the updated book. This time add the optional ConsistentRead parameter using DynamoDBContextConfig object.
}
```

```
        Book updatedBook = context.Load<Book>(bookID, new DynamoDBContextCon
fig { 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 DynamoDBContextCon
fig { ConsistentRead = true });

        if (deletedBook == null)

            Console.WriteLine("Book is deleted");

    }

}

[DynamoDBTable("ProductCatalog")]

public class Book

{

    [DynamoDBHashKey]      // Hash key.

    public int Id { get; set; }

    [DynamoDBProperty]

    public string Title { get; set; }

    [DynamoDBProperty]

    public string ISBN { get; set; }

    [DynamoDBProperty("Authors")]      // Multi-valued (set type) attribute.

    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 [Example Tables and Data \(p. 609\)](#). For step-by-step instructions to test the following sample, see [Using the AWS SDK for .NET \(p. 368\)](#).

```
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 SingleTableBatch
Write().");

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 hash attr", "some range attr");

var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);

Console.WriteLine("Performing batch write in MultiTableBatch
Write().");
superBatch.Execute();

}

}

[DynamoDBTable("Reply")]
public class Reply
{
    [DynamoDBHashKey]      // Hash key.

    public string Id { get; set; }

    [DynamoDBRangeKey]    // Range 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]  
  
    public string ForumName { get; set; }  
  
    [DynamoDBRangeKey]  
  
    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]
    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]      // Hash 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 sample tables, see [Example Tables and Data \(p. 609\)](#).

- 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 range attribute 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 range attribute. 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 [Using the AWS SDK for .NET \(p. 368\)](#).

```
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, threadSub  
ject);  
  
  
                // 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 threadSub  
ject)  
{  
    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,  
  
        QueryOp  
erator.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 res  
ult.....");  
  
    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]      // Hash key.

    public string Id { get; set; }

    [DynamoDBRangeKey]    // Range 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]
```

```
public string ForumName { get; set; }

[DynamoDBRangeKey]

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]      // Hash 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; }

}

}
```

Using the DynamoDB API

Topics

- [Using JSON Data Format with DynamoDB \(p. 477\)](#)
- [Making HTTP Requests to DynamoDB \(p. 479\)](#)
- [Handling Errors in DynamoDB Operations \(p. 482\)](#)
- [Operations in DynamoDB \(p. 488\)](#)

Using JSON Data Format with DynamoDB

Amazon DynamoDB uses JavaScript Object Notation format (JSON) to send and receive formatted data. 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.

For example, the following shows a table named "Users" with a composite primary key based on the attributes *user* and *time*.

```
{  
    "Table": {  
        "AttributeDefinitions": [  
            {  
                "AttributeName": "User",  
                "AttributeType": "S"  
            },  
            {  
                "AttributeName": "Time",  
                "AttributeType": "N"  
            }  
        ],  
        "TableName": "Users",  
        "KeySchema": [  
            {  
                "AttributeName": "User",  
                "KeyType": "HASH"  
            },  
            {  
                "AttributeName": "Time",  
                "KeyType": "RANGE"  
            }  
        ]  
    }  
}
```

```
        "AttributeName": "Time",
        "KeyType": "RANGE"
    }
],
"TableStatus": "ACTIVE",
"CreationDateTime": Mon Mar 25 09:46:00 PDT 2013,
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "ReadCapacityUnits": 10,
    "WriteCapacityUnits": 10
},
"TableSizeBytes": 949,
"ItemCount": 23
}
```

In the low-level JSON wire protocol used by DynamoDB, the following abbreviations are used to denote data types:

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

For more information about data types, see [DynamoDB Data Types \(p. 6\)](#).

JSON Is Used as a Transport Protocol Only

DynamoDB uses JSON only as a transport protocol. You use JSON notation to send data, and DynamoDB responds with JSON notation, but the data is not being stored "on-disk" in the JSON data format.

Applications that use DynamoDB must either implement their own JSON parsing or use a library like one of the AWS SDKs to do this parsing for them.

Many libraries support the JSON Number type by using the data types *int*, *long* and *double*. However, because DynamoDB provides a numeric type that does not map exactly to these other data types, these type distinctions can cause conflicts.

Unfortunately, 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, it 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 *"0.1"*, *"2"*, and *"0.3"* in the proper sequence.

If number precision is important to your application, convert number values to strings before you pass them to DynamoDB.

Note

DynamoDB limits numbers to 38 digits. More than 38 digits will cause an error.

Transferring Binary Data Type Values in JSON

DynamoDB supports binary attributes. However, JSON does not natively support encoding binary data. To send binary data over the wire, you will need to encode it as base64-encoded text. Upon receiving the payload DynamoDB decodes the payload back to binary.

For more information about the base64 encoding, go to <http://tools.ietf.org/html/rfc4648>. However, note the following DynamoDB specific restrictions:

- The base64 encoding may not include characters that are outside of the base64 character set, whitespaces, or line separators.
- The encoded data must include the correct number of padding characters as required by the base64 encoding guidelines.
- The DynamoDB base64 encoding uses the characters '/' and '+', as illustrated in table 1 in the preceding RFC.

Making HTTP Requests to DynamoDB

If you don't use one of the AWS SDKs, you can perform DynamoDB operations over HTTP using the POST request method. The POST method requires you to specify the operation in the header of the request and provide the data for the operation in JSON format in the body of the request.

HTTP Header Contents

DynamoDB requires the following information in the header of an HTTP request:

- *host* The DynamoDB endpoint. For more information about endpoints, see [Accessing DynamoDB \(p. 12\)](#).
- *x-amz-date* You must provide the time stamp in either the HTTP *Date* header or the AWS *x-amz-date* header. (Some HTTP client libraries don't let you set the *Date* header.) When an *x-amz-date* header is present, the system ignores any *Date* header during the request authentication.

The *x-amz-date* header must be specified in ISO 8601 basic format. For example:

- 20130315T092054Z
- *Authorization* The set of authorization parameters that AWS uses to ensure the validity and authenticity of the request. For more information about constructing this header, go to [Signature Version 4 Signing Process](#).
- *x-amz-target* The destination service of the request and the operation for the data, in the format

<<serviceName>>_<<API version>>. <<operationName>>

For example, *DynamoDB_20120810.CreateTable*

- *content-type* Specifies JSON and the version. For example, *Content-Type*: *application/x-amz-json-1.0*

The following is an example header for an HTTP request to create a table.

```
POST / HTTP/1.1
host: dynamodb.us-west-2.amazonaws.com
x-amz-date: 20130112T092034Z
x-amz-target: DynamoDB_20120810.CreateTable
Authorization: AWS4-HMAC-SHA256 Credential=AccessKeyID/20130112/us-west-2/dynamodb/aws4_request,SignedHeaders=host;x-amz-date;x-amz-target,Signature=145b1567ab3c50d929412f28f52c45dbf1e63ec5c66023d232a539a4afdf11fd9
content-type: application/x-amz-json-1.0
content-length: 23
connection: Keep-Alive
```

HTTP Body Content

The body of an HTTP request contains the data for the operation specified in the header of the HTTP request. The data must be formatted according to the JSON data schema for each DynamoDB API. The DynamoDB JSON data schema defines the types of data and parameters (such as comparison operators and enumeration constants) available for each operation.

Note

DynamoDB uses JSON as a transport protocol, and it parses the data for storage. However, data is not stored natively in JSON format. For more information, see [Using JSON Data Format with DynamoDB \(p. 477\)](#).

DynamoDB does not serialize null values. If you are using a JSON parser set to serialize null values for requests, DynamoDB ignores them.

Formatting the Body of HTTP requests

Use the JSON data format to convey data values and data structure, simultaneously. Elements can be nested within other elements by using bracket notation. The following example shows a request for several items from a table named "highscores".

```
{"RequestItems": {
    "highscores": {
        "Keys": [
            {"name": {"S": "Dave"}},
            {"name": {"S": "John"}},
            {"name": {"S": "Jane"}}
        ],
        "ProjectionExpression": "score"
    }
}}
```

Handling HTTP Responses

Here are some important headers in the HTTP response, and how you should handle them in your application:

- **HTTP/1.1**—This header is followed by a status code. A code value of *200* indicates a successful operation. For information on error codes, see [API Error Codes \(p. 482\)](#).
- **x-amzn-RequestId**—This header contains a request ID that you can use if you need to troubleshoot a request with DynamoDB. An example of a request ID is K2QH8DNOU907N97FNA2GDLL8OBVV4KQNSO5AEMVJF66Q9ASUAAJG.

- **x-amz-crc32**—DynamoDB calculates a CRC32 checksum of the UTF-8 encoded bytes in the HTTP response payload, and returns this checksum in the `x-amz-crc32` header. We recommend that you compute your own CRC32 checksum on the client side and compare it with the `x-amz-crc32` header; if the checksums do not match, it might indicate that the data was corrupted in transit. If this happens, you should retry your request.

AWS SDK users do not need to manually perform this verification, because the SDKs compute the checksum of each reply from DynamoDB and automatically retry if a mismatch is detected.

Sample DynamoDB JSON Request and Response

The following examples show a request for the item in a table where the hash key (Name) is `"Back To The Future"` and the range key (Year) is `1985`. Then it shows the DynamoDB response, including all the attributes of that item.

HTTP POST Request:

```
POST / HTTP/1.1
Host: dynamodb.us-west-2.amazonaws.com
x-amz-target: DynamoDB_20120810.GetItem
x-amz-date: 20130116T175052Z
Authorization: AWS4-HMAC-SHA256 Credential=AccessKeyID/20130116/us-west-2/dynamodb/aws4_request,SignedHeaders=host;x-amz-date;x-amz-target,Signature=ccb4ee48bcb506aaa7e412a7f2f5dceef338666e2478b34acf6631623d377d51
Date: Wed, 16 Jan 2013 17:50:52 GMT
Content-Type: application/x-amz-json-1.0
Content-Length: 135
Connection: Keep-Alive

{
    "TableName": "my-table",
    "Key": {
        "Name": {"S": "Back To The Future"},
        "Year": {"S": "1985"}
    }
}
```

DynamoDB Response:

```
HTTP/1.1 200
x-amzn-RequestId: K2QH8DNOU907N97FNA2GDL8OBVV4KQNSO5AEMVJF66Q9ASUAAJG
x-amz-crc32: 2215946753
Content-Type: application/x-amz-json-1.0
Content-Length: 144
Date: Mon, 16 Jan 2012 17:50:53 GMT

{
    "Item": {
        "Year": {"S": "1985"},
        "Fans": {"SS": ["Fox", "Lloyd"]},
        "Name": {"S": "Back To The Future"},
        "Rating": {"S": "*****"}
    }
}
```

For more request and response examples using various API operations, go to the [Amazon DynamoDB API Reference](#).

Handling Errors in DynamoDB Operations

Topics

- [Error Types \(p. 482\)](#)
- [API Error Codes \(p. 482\)](#)
- [Catching Errors \(p. 486\)](#)
- [Error Retries and Exponential Backoff \(p. 487\)](#)
- [Batch Operations and Error Handling \(p. 488\)](#)

This section describes how to handle client and server errors. For information on specific error messages, see [API Error Codes \(p. 482\)](#).

Error Types

While interacting with DynamoDB programmatically, you might encounter errors of two types: client errors and server errors. Each error has a status code (such as 400), an error code (such as ValidationException), and an error message (such as Supplied AttributeValue is empty, must contain exactly one of the supported data types).

Client Errors

Client errors are indicated by a 4xx HTTP response code.

Client errors indicate that DynamoDB found a problem with the client request, such as an authentication failure, missing required parameters, or exceeding the table's provisioned throughput. Fix the issue in the client application before submitting the request again.

Server Errors

Server errors are indicated by a 5xx HTTP response code, and need to be resolved by Amazon. You can resubmit/retry the request until it succeeds.

API Error Codes

HTTP status codes indicate whether an operation is successful or not. There are two types of error codes, client (4xx) and server (5xx).

A response code of `200` indicates the operation was successful.

The following table lists the errors returned by DynamoDB. Some errors are resolved if you simply retry the same request. The table indicates which errors are likely to be resolved with successive retries. If the Retry column contains a "Y", submit the same request again. If the Retry column contains an "N", fix the problem on the client side before submitting a new request. For more information about retrying requests, see [Error Retries and Exponential Backoff \(p. 487\)](#).

HTTP Status Code	Error code	Message	Cause	Retry
400	AccessDeniedException	Access denied.	General authentication failure. The client did not correctly sign the request. Consult the signing documentation.	N
400	ConditionalCheckFailedException	The conditional request failed.	Example: The expected value did not match what was stored in the system.	N
400	IncompleteSignatureException	The request signature does not conform to AWS standards.	The signature in the request did not include all of the required components. See HTTP Header Contents (p. 479) .	N
400	ItemCollectionSizeLimitExceededException	Collection size exceeded.	For a table with a local secondary index, a group of items with the same hash key has exceeded the maximum size limit of 10 GB. For more information on item collections, see Item Collections (p. 311) .	Y
400	LimitExceededException	Too many operations for a given subscriber.	Example: The number of concurrent table requests (cumulative number of tables in the <i>CREATING</i> , <i>DELETING</i> or <i>UPDATING</i> state) exceeds the maximum allowed of 10. The total limit of tables (currently in the <i>ACTIVE</i> state) is 250.	N

HTTP Status Code	Error code	Message	Cause	Retry
400	MissingAuthenticationTokenException	Request must contain a valid (registered) AWS Access Key ID.	The request did not include the required <code>x-amz-security-token</code> . See Making HTTP Requests to DynamoDB (p. 479) .	N
400	ProvisionedThroughputExceededException	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, go to 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 Back-off (p. 487) . Or, see Specifying Read and Write Requirements for Tables (p. 55) for other strategies.	Y
400	ResourceInUseException	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 <code>CREATING</code> state.	N
400	ResourceNotFoundException	Requested resource not found.	Example: Table which is being requested does not exist, or is too early in the <code>CREATING</code> state.	N

HTTP Status Code	Error code	Message	Cause	Retry
400	ThrottlingException	Rate of requests exceeds the allowed throughput.	This exception might be returned if the following API operations are requested too rapidly: CreateTable; UpdateTable; DeleteTable	Y
400	UnrecognizedClientException	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.	Y
400	ValidationException	The request could not be processed.	This error can occur for several reasons, such as a required parameter that is missing, a value that is out of range, or mismatched data types. The error message contains details about the specific part of the request that caused the error.	N
413	(n/a)	Request Entity Too Large.	Maximum request size of 1 MB exceeded.	N
500	InternalFailure	The server encountered an internal error trying to fulfill the request.	The server encountered an error while processing your request.	Y
500	InternalServerError	The server encountered an internal error trying to fulfill the request.	The server encountered an error while processing your request.	Y
503	ServiceUnavailableException	The service is currently unavailable or busy.	There was an unexpected error on the server while processing your request.	Y

Sample Error Response

The following is an HTTP response indicating the request exceeded the provisioned throughput limit for the table. The Error codes listed in the previous table appear after the pound sign (#) in the body of the response. When handling errors in an HTTP response, you only need to parse the content after the pound sign (#).

```
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.v20111205#ProvisionedThroughputExceededException",
  "message": "The level of configured provisioned throughput for the table was exceeded.
Consider increasing your provisioning level with the UpdateTable API"}
```

Catching Errors

For your application to run smoothly, you need to build logic into the application to catch and respond to errors. One typical approach is to implement your request within a `try` block or `if-then` statement.

The AWS SDKs perform their own retries and error checking. If you encounter an error while using one of the AWS SDKs, you should see the error code and description. You should also see a `Request ID` value. The `Request ID` value can help troubleshoot problems with DynamoDB support.

The following example uses the AWS SDK for Java to delete an item within a `try` block and uses a `catch` block to respond to the error (in this case, it warns the user that the request failed). The example uses the `AmazonServiceException` class to retrieve information about any operation errors, including the `Request ID`. The example also uses the `AmazonClientException` class in case the request is not successful for other reasons.

```
try {
    DeleteItemRequest request = new DeleteItemRequest(tableName, key);
    DeleteItemResult result = dynamoDB.deleteItem(request);
    System.out.println("Result: " + result);
    // Get error information from the service while trying to run the operation

} catch (AmazonServiceException ase) {
    System.err.println("Failed to delete item in " + tableName);
    // Get specific error information
    System.out.println("Error Message: " + ase.getMessage());
    System.out.println("HTTP Status Code: " + ase.getStatusCode());
    System.out.println("AWS Error Code: " + ase.getErrorCode());
    System.out.println("Error Type: " + ase.getErrorType());
    System.out.println("Request ID: " + ase.getRequestId());
    // Get information in case the operation is not successful for other reasons

} catch (AmazonClientException ace) {
    System.out.println("Caught an AmazonClientException, which means"+
        " the client encountered " +
        "an internal error while trying to " +
        "communicate with DynamoDB, " +
```

```
    "such as not being able to access the network.");
    System.out.println("Error Message: " + ace.getMessage());
}
```

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 supporting DynamoDB implements retry logic, automatically. The AWS SDK for Java automatically retries requests, and you can configure the retry settings using the `ClientConfiguration` class. For example, in some cases, such as a web page making a request with minimal latency and no retries, you might want to turn off the retry logic. Use the `ClientConfiguration` class and provide a `maxErrorRetry` value of 0 to turn off the retries. For more information, see [Using the AWS SDKs with DynamoDB \(p. 365\)](#).

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, we recommend using an exponential backoff algorithm for better flow control. The concept behind exponential backoff is to use progressively longer waits between retries for consecutive error responses. For example, up to 50 milliseconds before the first retry, up to 100 milliseconds before the second, up to 200 milliseconds before third, and so on. However, after a minute, if the request has not succeeded, the problem might be the request size exceeding your provisioned throughput, and not the request rate. Set the maximum number of retries to stop around one minute. If the request is not successful, investigate your provisioned throughput options. For more information, see [Guidelines for Working with Tables \(p. 59\)](#).

Following is a workflow showing retry logic. The workflow logic first determines if the error is a server error (5xx). Then, if the error is a server error, the code retries the original request.

```
currentRetry = 0
DO
    set retry to false
    execute DynamoDB request
    IF Exception.errorCode = ProvisionedThroughputExceededException
        set retry to true
    ELSE IF Exception.getStatusCode = 500
        set retry to true
    ELSE IF Exception.getStatusCode = 400
        set retry to false
        fix client error (4xx)

    IF retry = true
        wait for (2^currentRetry * 50) milliseconds
        currentRetry = currentRetry + 1

WHILE (retry = true AND currentRetry < MaxNumberOfRetries) // limit retries
```

Batch Operations and Error Handling

DynamoDB supports batch operations for reads and writes. The `BatchGetItem` API reads items from one or more tables, and the `BatchWriteItem` API puts or deletes items in one or more tables. These batch APIs are implemented as wrappers around other non-batch DynamoDB operations. `BatchGetItem` invokes the `GetItem` for each item in the batch. `BatchWriteItem` calls either `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 should fail, this will not cause the `BatchGetItem` to fail. On the other hand, if all five of the reads should 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.

Operations in DynamoDB

The DynamoDB API supports the following operations:

- [BatchGetItem](#)
- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [.GetItem](#)
- [ListTables](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)
- [UpdateTable](#)

For more information, go to the [Amazon DynamoDB API Reference](#).

Note

For backward compatibility with existing applications, DynamoDB also supports the previous API version (2011-12-05). For more information, see [Previous API Version \(2011-12-05\) \(p. 666\)](#).

New applications should use the current API version (2012-08-10). For more information on the current API, go to the [Amazon DynamoDB API Reference](#).

DynamoDB Example Application Using AWS SDK for Python (Boto): Tic-Tac-Toe

Topics

- [Step 1: Deploy and Test Locally Using DynamoDB Local \(p. 491\)](#)
- [Step 2: Examine the Data Model and Implementation Details \(p. 494\)](#)
- [Step 3: Deploy in Production Using the DynamoDB Service \(p. 501\)](#)
- [Step 4: Clean Up Resources \(p. 507\)](#)

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 API 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 composite value attributes.

You play the Tic-Tac-Toe application on the web as follows:

1. You log in to the application home page (for example, as user1).
2. User1 then invites another user (for example, user2) to play the game.

Until the opponent accepts the invitation, the game status remains as `PENDING`. After the opponent accepts the invite, the game status changes to `IN_PROGRESS`.

3. The game begins after user2 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 Using DynamoDB Local \(p. 491\)**](#) – In this section, you download, deploy, and test the application on your local computer. You will use DynamoDB Local to create the required tables.

- **Step 2: Examine the Data Model and Implementation Details (p. 494)** – This section first describes in detail the data model, including the indexes and the use of the composite value attribute. Then the section explains how the application works.
- **Step 3: Deploy in Production Using the DynamoDB Service (p. 501)** – 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. 507)** – 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 Using DynamoDB Local

Topics

- [1.1: Download and Install Required Packages \(p. 491\)](#)
- [1.2: Test the Game Application \(p. 492\)](#)

In this step you download, deploy, and test the Tic-Tac-Toe game application on your local computer. Instead of using the Amazon DynamoDB service, you will use DynamoDB Local to create the required table locally on your computer.

1.1: Download and Install Required Packages

You will need the following to test this application locally using DynamoDB Local:

- Python
- Flask (a microframework for Python)
- AWS SDK for Python (Boto)
- DynamoDB Local
- 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, click 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
```

3. Download DynamoDB Local. For instructions on how to run it, see [DynamoDB Local](#).
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 DynamoDB Local running locally on your computer.

To run the Tic-Tac-Toe application

1. Start DynamoDB Local.
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:

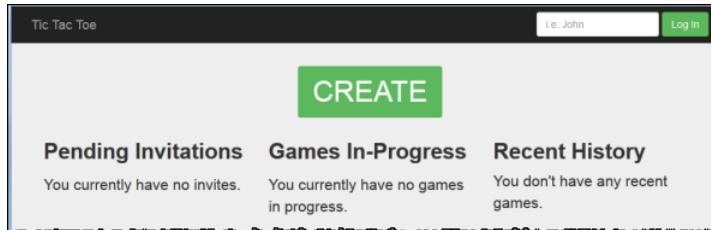
```
python.exe application.py --mode local --serverPort 5000 --port 8000
```

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:

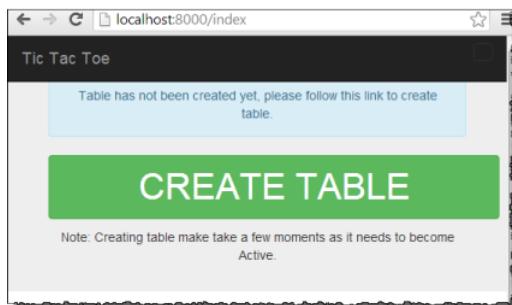


4. Type **user1** in the **Log in** box to log in as user1.

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 Local. Click **CREATE TABLE**.



6. Click **CREATE** to create the first tic-tac-toe game.
7. Type **user2** in the **Choose an Opponent** box, and click **Create Game!**



Doing this creates the game by adding an item in the Games table. It sets the game status to PENDING.

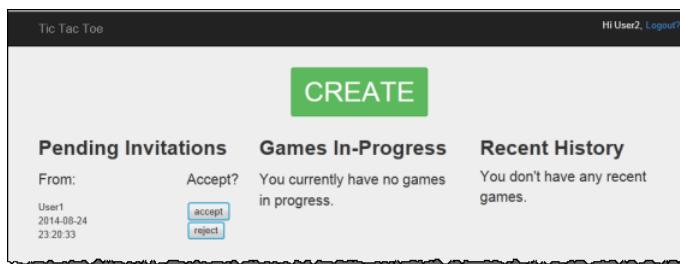
8. Open another browser window, and type the following.

```
http://localhost:5000/
```

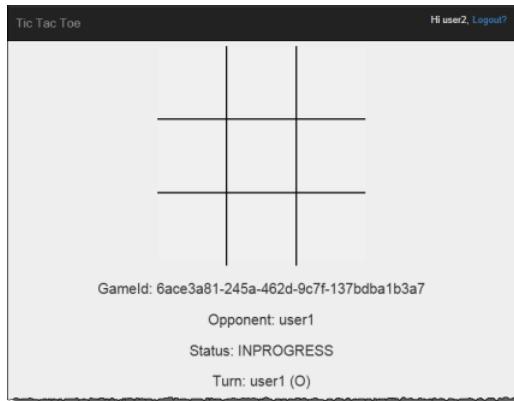
The browser passes information through cookies, so you should use incognito mode or private browsing so that your cookies don't carry over.

9. Log in as user2.

A page appears that shows a pending invitation from user1.



10. Click **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 clicked 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 click 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 \(p. 494\)](#)
- [2.2: Application in Action \(Code Walkthrough\) \(p. 496\)](#)

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. For more information on the DynamoDB data model, see [DynamoDB Data Model \(p. 3\)](#).

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 hash-type primary key made of the 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 \(p. 5\)](#).

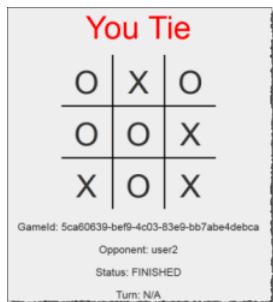
When a user initiates a tic-tac-toe game by inviting another user to play, the application creates a new item in the Games table with attributes storing game metadata, such as the following:

- 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 composite attribute, marking the initial game state as PENDING. The following screenshot shows an example item as it appears in the DynamoDB console:

Attribute	Type	Value
GameId (Hash Key)	String	"6ffff7f5-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:



- **Composite value attributes** – The StatusDate attribute illustrates a composite 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 range attribute of the index key. The benefit of using the StatusDate composite value attribute is further illustrated in the indexes discussed next.

- **Global secondary indexes** – You can use the table's primary key, GameId, to efficiently query the table to find a game item. To query the table on attributes other than the primary key attributes, DynamoDB supports the creation of secondary indexes. In this example application, you build the following two secondary indexes:

Global Secondary Indexes										
Index Name	Hash Key	Range Key	Projected Attributes	Status	Read Capacity Units	Write Capacity Units	Last Decrease Time	Last Increase Time	Index Size (Bytes)*	Item Count*
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

- **hostStatusDate**. This index has HostId as a hash key and StatusDate as a range key. You can use this index to query on HostId, for example to find games hosted by a particular user.
- **opponentStatusDate**. This index has Opponent as a hash key and StatusDate as a range 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 hash key attribute in these indexes is not the same the hash key attribute GameId, used in the primary key of the table.

Note that both the indexes specify StatusDate, a composite range attribute. 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 keeps items in the index sorted by range attribute. 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 `hostStatusDate` 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 `oppStatusDate` index.

For more information about secondary indexes, see [Improving Data Access with Secondary Indexes in DynamoDB \(p. 241\)](#).

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.

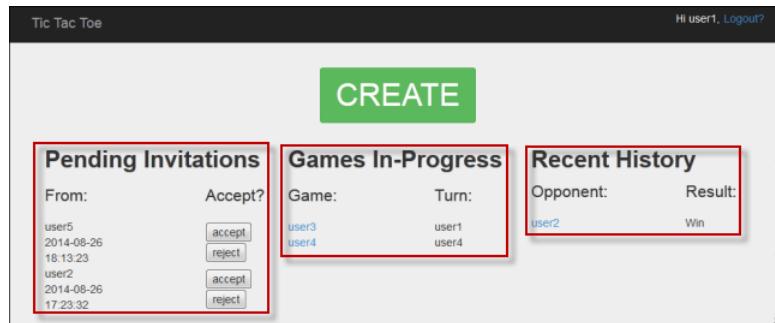
- **Game page** – This page shows the tic-tac-toe grid where users play.

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 removes the corresponding item from the table.

The function specifies the query as follows:

- It specifies the `opponentStatusDate` index to use with the following index key values and comparison operators:
 - The hash attribute is `OpponentId` and takes the index key `user ID`.
 - The range attribute is `StatusDate` and takes the comparison operator and index key value `beginswith="PENDING_"`.

You use the `opponentStatusDate` 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="opponentStatusDate",  
    limit=10)
```

In the index, for each `OpponentId` (the hash attribute) DynamoDB keeps items sorted by `StatusDate` (the range attribute). 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 `hostStatusDate` index. For the index, the query specifies primary key values—both the hash attribute (`HostId`) and range attribute (`StatusDate`) values, along with comparison operators.

```
hostGamesInProgress = self.cm.getGamesTable().query(HostId__eq=user,
                                                    StatusDate__beginswith=status,
                                                    index="hostStatusDate",
                                                    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 `opponentStatusDate` index.

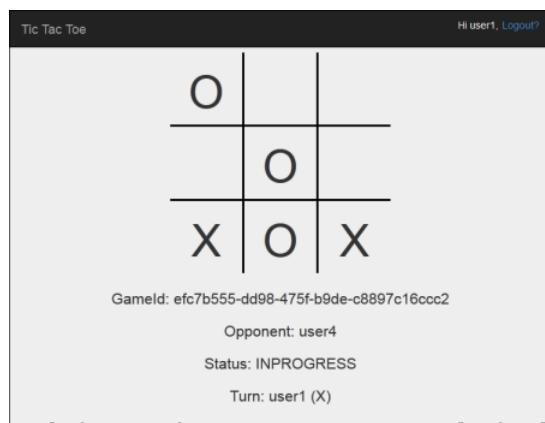
```
oppGamesInProgress = self.cm.getGamesTable().query(Opponent__eq=user,
                                                    StatusDate__beginswith=status,
                                                    index="opponentStatusDate",
                                                    limit=10)
```

- 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 click 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 AWS SDK for Python API function calls are to low-level functions and not to mid-level functions as in other parts of the code. Use of the low-level API lets the application have full control over the request it sends to perform the update.

The mid-level API requires you to get an item first in order to update it using the UpdateItem API operation. For information about the mid-level Python API for DynamoDB, go to [DynamoDB API](#). Note that these mid-level API functions are referred to as high-level in the SDK.

- 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 DynamoDB API operation. 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 clicked 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 \(p. 502\)](#)
- [3.2: Create the Games Table in Amazon DynamoDB \(p. 503\)](#)
- [3.3: Bundle and Deploy Tic-Tac-Toe Application Code \(p. 503\)](#)
- [3.4: Set Up the AWS Elastic Beanstalk Environment \(p. 504\)](#)

In the preceding sections, you deployed and tested the Tic-Tac-Toe application locally on your computer using DynamoDB Local. Now, you deploy the application in production as follows:

- Deploy the application using AWS Elastic Beanstalk, an easy-to-use service for deploying and scaling web applications and services. For more information, 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 hash-type primary key made of the `GameId` hash attribute 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": "*"
},
{
    "Action": [
        "dynamodb:*"
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dynamodb:us-west-2:922852403271:table/Games"
    ]
}
]
```

For further instructions, go to [Creating a Role for an AWS Service \(AWS Management Console\)](#) in *Using IAM*.

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 Using DynamoDB Local \(p. 491\)](#).

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.

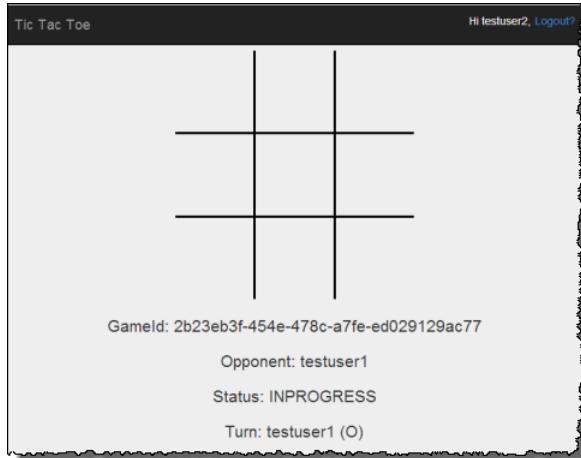
3. Zip the content of the root folder of the application (which includes the `beanstalk.config` file)—for example, `TicTacToe.zip`.
4. Upload the `.zip` file to an Amazon Simple Storage Service (Amazon S3) bucket. In the next section, you provide this `.zip` file to Elastic Beanstalk to upload on the 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.

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.

1. Type the following custom URL to set up an Elastic Beanstalk console to set up the environment:

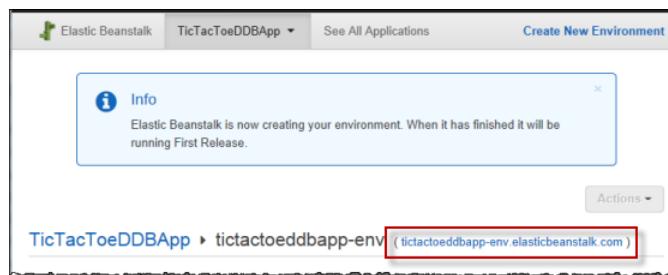
```
https://console.aws.amazon.com/elasticbeanstalk/?region=<AWS-Region>#/newApplication  
?applicationName=TicTacToe<your-name>  
&solutionStackName=Python  
&sourceBundleUrl=https://s3.amazonaws.com/<bucket-name>/TicTacToe.zip  
&environmentType=SingleInstance  
&instanceType=t1.micro
```

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.

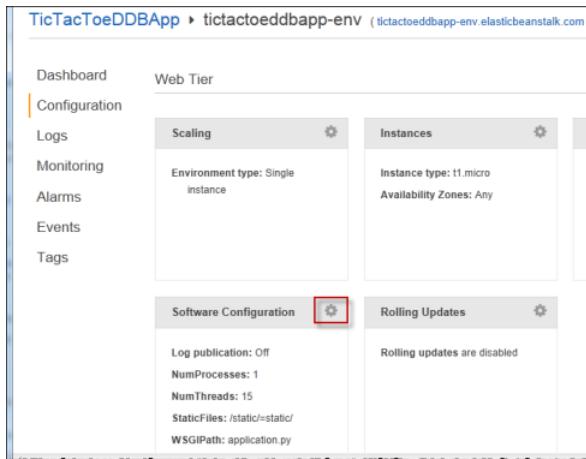
2. In the Elastic Beanstalk console, click **Review and Launch**, and then click **Launch**.
3. Note the URL for future reference. This URL opens your Tic-Tac-Toe application home page.



4. Configure the Tic-Tac-Toe application so it knows the location of the configuration file.

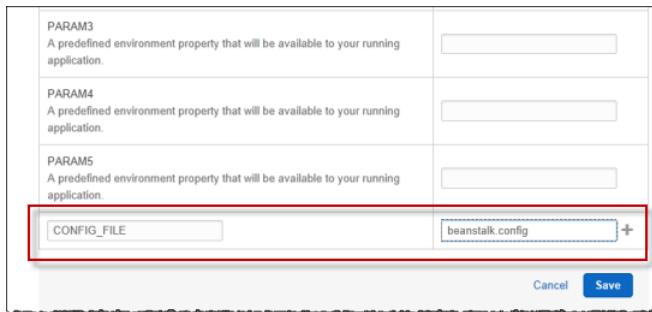
After Elastic Beanstalk creates the application, click **Configuration**.

- Click the gear box next to **Software Configuration**, as shown in the following screenshot.



- At the end of the **Environment Properties** section, type **CONFIG_FILE** and its value `beanstalk.config`, and then click **Save**.

It might take a few minutes for this environment update to complete.

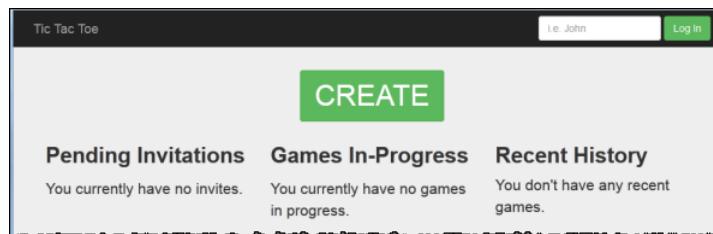


After the update completes, you can play the game.

- In the browser, type the URL you copied in the previous step, as shown in the following example.

```
http://<open-name>.elasticbeanstalk.com
```

Doing this will open the application home page.



- Log in as testuser1, and click **CREATE** to start a new tic-tac-toe game.
- 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, click **accept**.



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.

Additional Tools and Resources For DynamoDB

Topics

- [DynamoDB Local \(p. 508\)](#)
- [JavaScript Shell for DynamoDB Local \(p. 511\)](#)
- [AWS Command Line Interface for DynamoDB \(p. 516\)](#)

This section describes some additional tools and resources for application development with Amazon DynamoDB.

Tip

For more best practices, how-to guides and tools, be sure to check the DynamoDB Developer Resources page:

- <http://www.amazonaws.cn/dynamodb/developer-resources/>

DynamoDB Local

Topics

- [Downloading and Running DynamoDB Local \(p. 509\)](#)
- [Using DynamoDB Local \(p. 510\)](#)
- [Usage Notes \(p. 510\)](#)
- [Differences Between DynamoDB Local and DynamoDB \(p. 511\)](#)

DynamoDB Local is a small client-side database and server that mimics the DynamoDB service. DynamoDB Local enables you to write applications that use the DynamoDB API, without actually manipulating any tables or data in DynamoDB. Instead, all of the API actions are rerouted to DynamoDB Local. When your application creates a table or modifies data, those changes are written to a local database. This lets you save on provisioned throughput, data storage, and data transfer fees.

DynamoDB Local is compatible with the DynamoDB API. When you are ready to deploy your application, you simply redirect it to DynamoDB, without having to modify your application code. In addition, you do

not need to have an Internet connection to use DynamoDB Local. You can develop applications without having to be connected to the network.

Note

DynamoDB Local also includes the JavaScript Shell, an interactive browser-based interface for DynamoDB Local. To learn more about the JavaScript Shell, see [JavaScript Shell for DynamoDB Local \(p. 511\)](#).

Downloading and Running DynamoDB Local

DynamoDB Local is available as an executable Java archive (.jar file), and will run on Windows, Mac, or Linux computers.

Important

DynamoDB Local supports the Java Runtime Engine (JRE) version 6.x or newer; it will not run on older JRE versions.

You can download DynamoDB Local for free using one of these links:

- .tar.gz format: http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest.tar.gz
- .zip format: http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest.zip

Once you have downloaded the archive to your computer, extract the contents and copy the extracted directory to a location of your choice. To start DynamoDB Local, open a command prompt window, navigate to the downloaded directory where you will find `DynamoDBLocal.jar`, and enter the following command:

```
java -Djava.library.path= ./DynamoDBLocal_lib -jar DynamoDBLocal.jar [options]
```

The DynamoDB Local command line accepts the following options:

- `-cors value` — Enable CORS support (cross-origin resource sharing) for JavaScript. You must provide a comma-separated "allow" list of specific domains. You can also use an asterisk (*) to allow public access.
- `-dbPath value` — The directory where DynamoDB Local will write its database file. If you do not specify this option, the file will be written to the current directory. Note that you cannot specify both `--dbPath` and `--inMemory` at once.
- `-delayTransientStatuses` — Causes DynamoDB Local to introduce delays for certain operations. DynamoDB Local can perform some tasks almost instantaneously, such as create/update/delete operations on tables and indexes; however, the actual DynamoDB service requires more time for these tasks. Setting this parameter helps DynamoDB Local simulate the behavior of DynamoDB 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 for DynamoDB Local .
- `-inMemory` — DynamoDB Local will run in memory, instead of using a database file. When you stop DynamoDB Local, none of the data will be saved. Note that you cannot specify both `--dbPath` and `--inMemory` at once.
- `-optimizeDbBeforeStartup` — Optimizes the underlying database tables before starting up the DynamoDB Local server. You must also specify `-dbPath` when you use this parameter.
- `-port value` — The port number that DynamoDB Local will use to communicate with your application. If you do not specify this option, the default port is 8000.
- `-sharedDb` — DynamoDB Local will use a single database file, instead of using separate files for each credential and region. If you specify `-sharedDb`, all DynamoDB Local clients will interact with the same set of tables regardless of their region and credential configuration.

DynamoDB Local will process incoming requests until you stop it. To stop DynamoDB Local, type Ctrl+C in the command prompt window.

Note

If you are using a version of DynamoDB Local that was released prior to December 12, 2013, use this command line instead:

```
java -Djava.library.path=. -jar DynamoDBLocal.jar
```

We recommend that you use the download link (see above) to obtain the latest version of DynamoDB Local.

Using DynamoDB Local

To use DynamoDB Local with an application program, you need to configure your client so that it can communicate with the DynamoDB Local endpoint. The way that you do this depends on what programming language and AWS software development kit (SDK) you are using. The following code snippets show examples of how you can do this.

Java

```
client = new AmazonDynamoDBClient(credentials);
client.setEndpoint("http://localhost:8000");
```

.NET

```
var config = new AmazonDynamoDBConfig();
config.ServiceURL = "http://localhost:8000";
client = new AmazonDynamoDBClient(config);
```

PHP

```
$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => Region::US_EAST_1, #replace with your desired region
    'base_url' => 'http://localhost:8000'
));
```

To verify your setup, modify your client object so that it uses the DynamoDB Local endpoint. When you run your program, you should see diagnostic messages in the window where DynamoDB Local is running, indicating that DynamoDB Local is processing requests from your code.

Usage Notes

In general, application code that uses DynamoDB should run unmodified when used with DynamoDB Local. However, you should be aware of some usage notes:

- Unless you use the `-inMemory` option, DynamoDB Local writes a database file to disk. By default, this file is written to the same directory from where you launched DynamoDB Local. (You can specify a different directory using the `-dbPath` parameter.) 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 this database file, you will lose any data you have stored in DynamoDB Local.
- If you use the `-optimizeDbBeforeStartup` option, you must also specify the `-dbPath` parameter so that DynamoDB Local will be able to find its database file.

- DynamoDB Local only uses the values for AWS access key ID and region to name the database file, so you can set them to any value you like.

Note

Each database file represents a separate set of data, so if you change the access key or region it creates a new database without the data stored in previous data files.

- DynamoDB Local ignores your AWS secret access key, even though you must still specify this parameter. We recommend that you set it to a dummy string of characters, so that no one will be able to see your secret access key.

Differences Between DynamoDB Local and DynamoDB

DynamoDB Local attempts to emulate the actual DynamoDB service as closely as possible; however, there are several differences:

- Regions and distinct AWS accounts are not supported at the client level.
- DynamoDB Local ignores provisioned throughput settings, even though the API requires them. For `CreateTable`, you can specify any numbers you want for provisioned read and write throughput, even though these numbers will not be used. You can call `UpdateTable` as many times as you like per day; however, any changes to provisioned throughput values are ignored.
- DynamoDB Local does not throttle read or write activity. `CreateTable`, `UpdateTable` and `DeleteTable` operations occur immediately, and table state is always ACTIVE. The speed of read and write operations on table data are limited only by the speed of your computer.
- DynamoDB Local does not throttle read or write activity. The speed of read and write operations on table data are limited only by the speed of your computer. `CreateTable` and `DeleteTable` operations will occur immediately, and table state is always ACTIVE. `UpdateTable` operations that only change the provisioned throughput settings on tables and/or global secondary indexes will 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 ACTIVE state. The table remains ACTIVE during this time.
- Read operations in DynamoDB Local are eventually consistent. However, due to the speed of DynamoDB Local, most reads will actually appear to be strongly consistent.
- DynamoDB Local does not keep track of consumed capacity. In API responses, nulls are returned instead of capacity units.
- DynamoDB Local does not keep track of item collection metrics; nor does it support item collection sizes. In API responses, nulls are returned instead of item collection metrics.
- In the DynamoDB API, there is a 1 MB limit on data returned per result set. The DynamoDB service enforces this limit, and so does DynamoDB Local. However, when querying an index, DynamoDB only calculates the size of the projected key and attributes. By contrast, DynamoDB Local calculates the size of the entire item.

JavaScript Shell for DynamoDB Local

The JavaScript Shell for DynamoDB Local can help jump-start your usage of DynamoDB, all within an interactive, hands-on environment. The JavaScript Shell is bundled with DynamoDB Local, and provides an easy-to-use environment for prototyping and application development.



To get started with the JavaScript Shell, do the following:

- Download the latest version of DynamoDB Local, and then run it on your computer. For details on how to do this, see [Downloading and Running DynamoDB Local \(p. 509\)](#).
- Open a web browser on your computer and go to the following URL: `http://localhost:8000/shell`

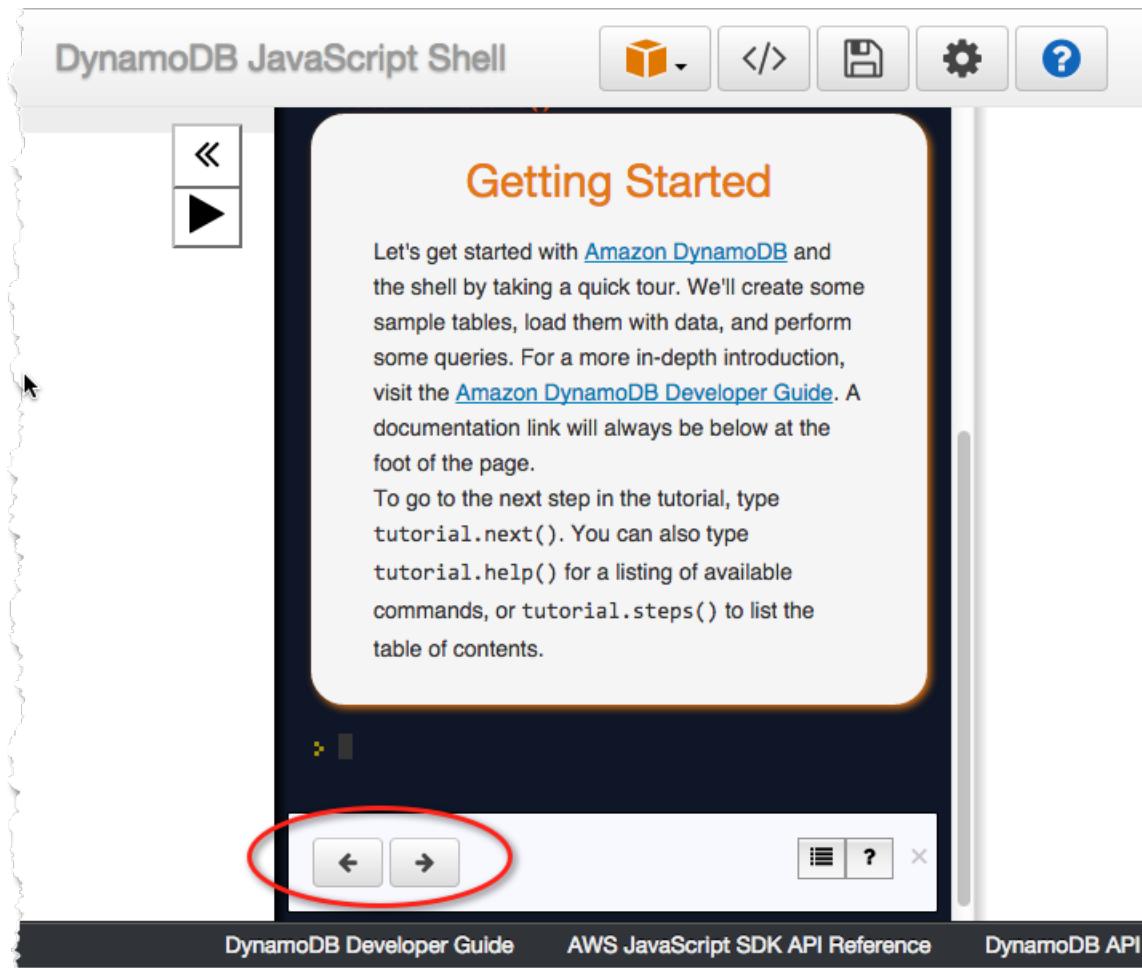
Tutorial

If this is your first time using the JavaScript Shell, we recommend that you take advantage of the built-in tutorial. The tutorial gives you a guided tour of the JavaScript Shell, and shows you how to build a working JavaScript application that runs on top of DynamoDB Local.

To launch the tutorial, go to the right-hand side of the screen and type `tutorial.start()` as shown below:



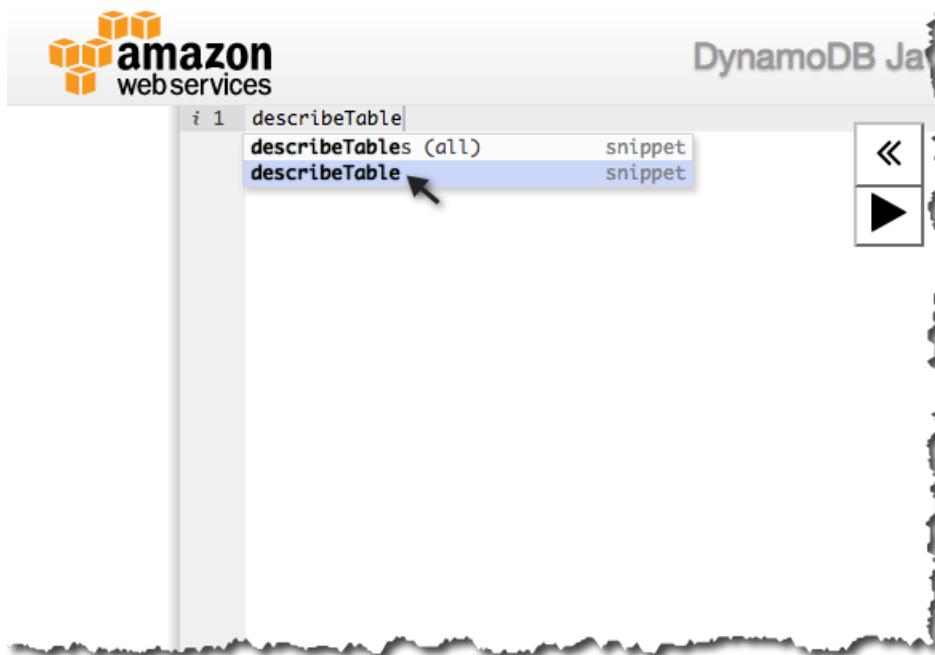
Use the arrows at the bottom of the window to navigate to different sections within the tutorial.



Code Editor

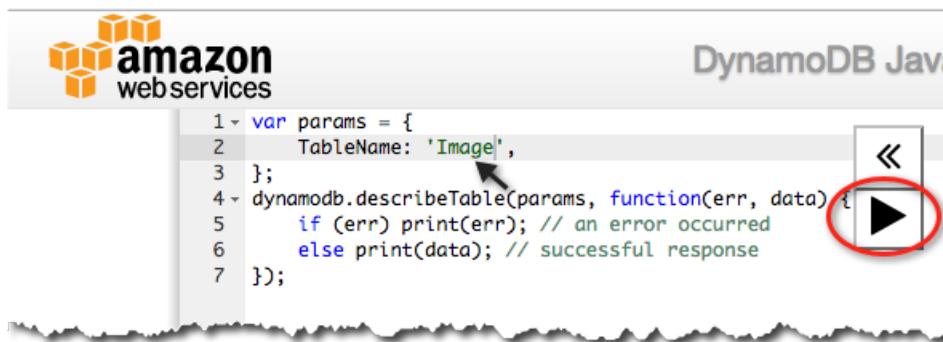
After you complete the tutorial, you can interact with DynamoDB Local using the code editor on the left-hand side of the screen. The JavaScript Shell uses the JavaScript API for DynamoDB, so that you can issue API calls and see the results immediately. In addition, the code editor is preloaded with JavaScript snippets and macros for listing tables, issuing queries, manipulating data, and much more. Type **Ctrl+Space** in the editor for a complete list.

The following screen shots show the editor in action. We begin by typing `describeTable` and then entering **Ctrl+Space**, as shown below.



The autocomplete feature displays the matching snippets. For this exercise, we select the `describeTable` snippet. The editor expands the snippet in the editor window, where it is ready for editing.

To describe a table, we replace the placeholder `TableName:` with an actual table name, and then click the Run button as shown below.



The results of `describeTable` are displayed in the right-hand side of the window.



```
=>
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "Id",
        "AttributeType": "S"
      }
    ],
    "TableName": "Image",
    "KeySchema": [
      {
        "AttributeName": "Id",
        "KeyType": "HASH"
      }
    ],
    "TableStatus": "ACTIVE",
    "CreationDateTime": null,
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 1,
      "WriteCapacityUnits": 1
    },
    "TableSizeBytes": 0,
    "ItemCount": 0
  }
}
>
```

Tip

Use the toolbar in the upper right-hand corner of the window to access other features of the JavaScript Shell. If you need help, click the question mark button (?).

AWS Command Line Interface for DynamoDB

Topics

- [Downloading and Configuring the AWS CLI \(p. 516\)](#)
- [Using the AWS CLI with DynamoDB \(p. 517\)](#)
- [Using the AWS CLI with DynamoDB Local \(p. 518\)](#)

The AWS Command Line Interface (AWS CLI) provides support for DynamoDB. 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.

Downloading and Configuring the AWS CLI

The AWS CLI is available at <http://www.amazonaws.cn/cli>, and will run on Windows, Mac, or Linux computers. After you download the AWS CLI, go to [AWS Command Line Interface User Guide](#) and follow the setup instructions there.

Using the AWS CLI with DynamoDB

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

For example, the following command will create a table named *MusicCollection*. (For easier readability, long commands in this section are broken into separate lines.)

```
aws dynamodb create-table
  --table-name MusicCollection
  --attribute-definitions
    AttributeName=Artist,AttributeType=S AttributeName=SongTitle,Attribute
    Type=S
    --key-schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,Key
    Type=RANGE
    --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1
```

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

```
aws dynamodb put-item
  --table-name MusicCollection
  --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 MusicCollection
  --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 MusicCollection --key-conditions file://key-conditions.json
```

For more documentation on using the AWS CLI with DynamoDB, go to <http://docs.amazonaws.cn/cli/latest/reference/dynamodb/index.html>.

Using the AWS CLI with DynamoDB Local

The AWS CLI can interact with [DynamoDB Local \(p. 508\)](#), in addition to DynamoDB. To do this, add the `--endpoint-url` parameter to each command:

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

Here is an example, using the AWS CLI to list the tables in a DynamoDB Local database:

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

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

Note

At this time, the AWS CLI cannot use DynamoDB Local as a default endpoint; therefore, you will need to specify `--endpoint-url` with each CLI command.

DynamoDB Integration with Other Services

Topics

- [Monitoring DynamoDB with CloudWatch \(p. 519\)](#)
- [Using IAM to Control Access to DynamoDB Resources \(p. 527\)](#)
- [Exporting, Importing and Transforming Data Using AWS Data Pipeline \(p. 550\)](#)
- [Querying and Joining Tables Using Amazon Elastic MapReduce \(p. 564\)](#)
- [Loading Data From DynamoDB Into Amazon Redshift \(p. 595\)](#)

Amazon DynamoDB is integrated with other AWS services, letting you automate repeating tasks or build applications that span multiple services. Here are some ways you can leverage this integration:

- Gather and analyze DynamoDB metrics using Amazon CloudWatch
- Use AWS Identity and Access Management to control access to DynamoDB tables, indexes and other resources.
- Periodically export DynamoDB data to Amazon S3, within the same region or across regions.
- Use Amazon Elastic MapReduce to perform complex queries on DynamoDB data.
- Load data from DynamoDB into Amazon Redshift for a complete data warehousing solution.

Monitoring DynamoDB with CloudWatch

Topics

- [AWS Management Console \(p. 520\)](#)
- [Command Line Interface \(CLI\) \(p. 520\)](#)
- [API \(p. 520\)](#)
- [DynamoDB Metrics \(p. 521\)](#)
- [Dimensions for DynamoDB Metrics \(p. 526\)](#)

Amazon DynamoDB and Amazon CloudWatch are integrated, so you can gather and analyze performance metrics. You can monitor these metrics using the CloudWatch console, CloudWatch's own command-line

interface, or programmatically using the CloudWatch API. CloudWatch also allows you to set alarms when you reach a specified threshold for a metric.

For more information about using CloudWatch and alarms, see the [CloudWatch Documentation](#).

AWS Management Console

To view CloudWatch data for a table in DynamoDB

1. Sign in to the AWS Management Console and open the CloudWatch console at <https://console.amazonaws.cn/cloudwatch/>.
2. In the navigation pane, click **Metrics**.
3. In the **CloudWatch Metrics by Category** pane, under **DynamoDB Metrics**, select **Table Metrics**, and then in the upper pane, scroll down to view the full list of metrics for your table.

The available DynamoDB metric options appear in the **Viewing** list.

To select or deselect an individual metric, in the results pane, select the check box next to the resource name and metric. Graphs showing the metrics for the selected items are displayed at the bottom of the console. To learn more about CloudWatch graphs, see the [Amazon CloudWatch Developer Guide](#).

Command Line Interface (CLI)

To view CloudWatch data for a table in DynamoDB

1. Install the CloudWatch command line tool. For instructions and links about the tool, see the [Amazon CloudWatch Developer Guide](#).
2. Use the [CloudWatch command line client commands](#) to fetch information. The parameters for each command are listed in [DynamoDB Metrics \(p. 521\)](#).

The following example uses the command **mon-get-stats** with the following parameters to determine how many `GetItem` requests exceeded your provisioned throughput during a specific time period.

```
PROMPT>mon-get-stats ThrottledRequests --aws-credential-file ./credential-file-path.template --namespace "AWS/DynamoDB"
--statistics "Sum" --start-time 2013-11-14T00:00:00Z --end-time 2013-11-16T00:00:00Z --period 300
--dimensions "Operation=GetItem"
```

API

CloudWatch also supports a Query API so you can request information programmatically. For more information, see the [CloudWatch Query API documentation](#) and [Amazon CloudWatch API Reference](#).

When a CloudWatch action requires a parameter that is specific to DynamoDB monitoring, such as `MetricName`, use the values listed in [DynamoDB Metrics \(p. 521\)](#).

The following example shows a CloudWatch API request, using the following parameters:

- `Statistics.member.1 = Average`
- `Dimensions.member.1 = Operation=PutItem,TableName=TestTable`
- `Namespace = AWS/DynamoDB`

- *StartTime* = 2013-11-14T00:00:00Z
- *EndTime* = 2013-11-16T00:00:00Z
- *Period* = 300
- *MetricName* = SuccessfulRequestLatency

Here is what the CloudWatch request looks like. However, note that this is just to show the form of the request; you will need to construct your own request based on your metrics and time frame.

```
http://monitoring.amazonaws.com/
    ?SignatureVersion=2
    &Action=SuccessfulRequestLatency
    &Version=2010-08-01
    &StartTime=2013-11-14T00:00:00
    &EndTime=2013-11-16T00:00:00
    &Period=300
    &Statistics.member.1=Average
    &Dimensions.member.1=Operation=PutItem,TableName=TestTable
    &Namespace=AWS/DynamoDB
    &MetricName=SuccessfulRequestLatency

    &Timestamp=2013-10-15T17%3A48%3A21.746Z
    &AWSAccessKeyId=<AWS Access Key ID>
    &Signature=<Signature>
```

DynamoDB Metrics

The following metrics are available from 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 error code; if no requests have resulted in a 400 code during a particular time period, then no metrics for `UserErrors` are shown.

Note

Not all statistics, such as `Average` or `Sum`, are applicable for every metric. However, all of these values are available through the console, API, and command line client for all services. In the following table, each metric has a list of Valid Statistics that is applicable to that metric.

Metric	Description
SuccessfulRequestLatency	<p>The number of successful requests in the specified time period. By default, <code>SuccessfulRequestLatency</code> provides the elapsed time for successful calls. You can see statistics for the Minimum, Maximum, or Average, over time.</p> <p>Note CloudWatch also provides a Data Samples statistic: the total number of successful calls for a sample time period.</p> <p>Units: Milliseconds (or a count for Data Samples)</p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics: Minimum, Maximum, Average, Data Samples</p>

Metric	Description
UserErrors	<p>The number of requests generating an HTTP 400 status code (likely indicating a client error) response in the specified time period.</p> <p>Units: Count</p> <p>This is an account level metric. It represents HTTP 400 errors for DynamoDB requests in this AWS account.</p> <p>Valid Statistics: Sum, Data Samples</p>
SystemErrors	<p>The number of requests generating a 500 status code (likely indicating a server error) response in the specified time period.</p> <p>Units: Count</p> <p>Dimensions: All dimensions</p> <p>Valid Statistics: Sum, Data Samples</p>
ThrottledRequests	<p>The number of user requests that exceeded the preset provisioned throughput limits in the specified time period.</p> <p><code>ThrottledRequests</code> is incremented by 1 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 1.</p> <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>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics: Sum, Data Samples</p>

Metric	Description
ReadThrottleEvents	<p>The number of read events that exceeded the preset provisioned throughput limits in the specified time period.</p> <p>A single API 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 1 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>GlobalSecondaryIndexName</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics: Sum, Data Samples</p>
WriteThrottleEvents	<p>The number of write events that exceeded the preset provisioned throughput limits in the specified time period.</p> <p>A single API 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, <code>WriteThrottleEvents</code> metric is incremented by 1 if that event is throttled. If any of the events are throttled, then <code>ThrottledRequests</code> is also incremented by 1.</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: Sum, Data Samples</p>

Metric	Description
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: Minimum, Maximum, Average, Sum</p>
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: Minimum, Maximum, Average, Sum</p>
ConsumedReadCapacityUnits	<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> <p>Note Use the <code>Sum</code> value to calculate the consumed throughput. For example, get the <code>Sum</code> value over a span of 5 minutes. Divide the <code>Sum</code> value by the number of seconds in 5 minutes (300) to get an average for the <code>ConsumedReadCapacityUnits</code> per second. (Note that such an average does not highlight any large spikes in read activity that take place during this period.) 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: Minimum, Maximum, Average, Sum</p>

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 Sum value to calculate the consumed throughput. For example, get the Sum value over a span of 5 minutes. Divide the Sum value by the number of seconds in 5 minutes (300) to get an average for the ConsumedWriteCapacityUnits per second. (Note that such an average does not highlight any large spikes in write activity that take place during this period.) You can compare the calculated value to the provisioned throughput value you provide DynamoDB.</p> <p>Units: Count</p> <p>Dimensions: TableName, GlobalSecondaryIndexName</p> <p>Valid Statistics: Minimum, Maximum, Average, Sum</p>
ReturnedItemCount	<p>The number of items returned by a Scan or Query operation.</p> <p>Units: Count</p> <p>Dimensions: TableName</p> <p>Valid Statistics: Minimum, Maximum, Average, Data Samples, Sum</p>
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: Count</p> <p>Dimensions: TableName, GlobalSecondaryIndexName</p> <p>Valid Statistics: Minimum, Maximum, Average, Sum</p>

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, then 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: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics: Minimum, Maximum, Average, Sum</p>
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: Minimum, Maximum, Average, Sum</p>

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
<code>TableName</code>	This dimension limits the data you request to a specific table. This value can be any table name for the current account.
<code>GlobalSecondaryIndexName</code>	This dimension limits the data you request to a global secondary index on a table. If you specify <code>GlobalSecondaryIndexName</code> , you must also specify <code>TableName</code> .

Dimension	Description
Operation	<p>The operation corresponds to the DynamoDB service API, and can be one of the following:</p> <ul style="list-style-type: none">• PutItem• DeleteItem• UpdateItem• GetItem• BatchGetItem• Scan• Query <p>For all of the operations in the current DynamoDB service API, see Operations in Amazon DynamoDB.</p>

Using IAM to Control Access to DynamoDB Resources

Topics

- [Amazon Resource Names \(ARNs\) for DynamoDB \(p. 528\)](#)
- [DynamoDB Actions \(p. 528\)](#)
- [Condition Types and Operators \(p. 529\)](#)
- [IAM Policy Keys \(p. 529\)](#)
- [Example Policies for API Actions and Resource Access \(p. 531\)](#)
- [Fine-Grained Access Control for DynamoDB \(p. 536\)](#)
- [Example Policies for Fine-Grained Access Control \(p. 538\)](#)
- [Using Web Identity Federation \(p. 544\)](#)

Amazon DynamoDB integrates with AWS Identity and Access Management (IAM), a service that enables you to do the following:

- Create users and groups under your AWS account
- Easily share your AWS resources between the users in your AWS account
- Assign unique security credentials to each user
- Control each user's access to services and resources
- Get a single bill for all users in your AWS account

For more information about IAM, see the following:

- [Identity and Access Management \(IAM\)](#)
- [IAM Getting Started Guide](#)
- [Using IAM](#)

You can use IAM to grant access to DynamoDB resources and API actions. To do this, you first write an IAM policy, which is a document that explicitly lists the permissions you want to grant. You then attach that policy to an IAM user or role.

For example, an IAM user named Joe could create a DynamoDB table, and then write an IAM policy to allow read-only access to this table. Joe could then apply that policy to selected IAM users, groups or roles in his AWS account. These recipients would then have read-only access to Joe's table.

To create and manage IAM policies, go to the IAM console at <https://console.aws.amazon.com/iam/>.

For examples of IAM policies that cover DynamoDB actions and resources, see:

- [Example Policies for API Actions and Resource Access \(p. 531\)](#)
- [Example Policies for Fine-Grained Access Control \(p. 538\)](#)

Amazon Resource Names (ARNs) for DynamoDB

When writing IAM policies for DynamoDB, you use Amazon Resource Names (ARNs) to refer to individual tables and indexes. If you want to write an IAM policy for a particular table, you specify an ARN with the table name, the region in which the table is located, and the owners' AWS account number.

Here is an example ARN for a table named *Books*, which is located in *us-west-2* and is owned by AWS account number 12345678012:

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
```

Here is another example ARN for an index named *TitleIndex* on the *Books* table:

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/TitleIndex"
```

Note

To find your AWS account number, go to the AWS Management Console and click **My Account**. Your AWS account number is shown in the upper right portion of the **Manage Your Account** page. The account number is formatted using dashes (for example, 1234-5678-9012); however, if you use it in an ARN, be sure to remove the dashes (for example, 123456789012).

You can use resource-level ARNs in IAM policies for all DynamoDB actions, with the exception of *ListTables*. The *ListTables* action returns the table names owned by the current account making the request for the current region; it is the only DynamoDB action that does not support resource-level ARN policies.

DynamoDB Actions

In an IAM policy, you can specify any of the actions in the [DynamoDB API](#). You must prefix each action name with the lowercase string `dynamodb:`. Here are some examples:

- `dynamodb:CreateTable`
- `dynamodb:PutItem`
- `dynamodb:Query`
- `dynamodb:UpdateItem`

For a list of API actions, go to the [Amazon DynamoDB API Reference](#).

DynamoDB allows customers to purchase Reserved Capacity, as described at <http://aws.amazon.com/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. The following actions are available for controlling access to Reserved Capacity management:

- dynamodb:PurchaseReservedCapacityOfferings
- dynamodb:DescribeReservedCapacityOfferings
- dynamodb:DescribeReservedCapacity

To refer to *all* of the DynamoDB actions, use an asterisk:

- dynamodb:*

Condition Types and Operators

In IAM, a condition is composed of a *condition type* and an *operator*. The following condition types are available:

- String
- Numeric
- Date and time
- Boolean
- Binary
- IP address
- Amazon Resource Name (ARN)
- ...IfExists
- Existence of condition keys

The operators that are available depend on the condition type being used. For example, with a String value, you can specify StringEquals, StringNotEquals, StringEqualsIgnoreCase, StringNotEqualsIgnoreCase, StringLike, or StringNotLike.

You can optionally specify a [set operator](#) in a condition. The following IAM set operators are available:

- ForAnyValue — Returns true if any one of the key values matches any one of the condition values.
- ForAllValues — Returns true if there's a match between every one of the specified key values and at least one condition value.

For more information about IAM condition types and operators, see the [Condition](#) section in [Using IAM](#).

IAM Policy Keys

The following IAM policy keys are available for DynamoDB and other AWS services.

Policy Keys Specific to DynamoDB

- `dynamodb:LeadingKeys` – Represents the first key attribute of a table. For a hash type or a hash-and-range type primary key, `LeadingKeys` is just the hash key.
Note that `LeadingKeys` is plural, even if it is used with single-item actions. In addition, note that you must use the `ForAllValues` modifier when using `LeadingKeys` in a condition.
- `dynamodb:Select` – Represents the `Select` parameter of a `Query` or `Scan` request. `Select` can be any of the following values:
 - `ALL_ATTRIBUTES`
 - `ALL_PROJECTED_ATTRIBUTES`
 - `SPECIFIC_ATTRIBUTES`
 - `COUNT`
- `dynamodb:Attributes` – Represents a list of the attribute names in a request, or the attributes that are returned from a request. The value for `Attributes` is expressed as the parameter name of a DynamoDB action.

Parameter Name	API Actions That Use This Parameter
<code>AttributesToGet</code>	<code>BatchGetItem</code> , <code>.GetItem</code> , <code>Query</code> , <code>Scan</code>
<code>AttributeUpdates</code>	<code>UpdateItem</code>
<code>Expected</code>	<code>DeleteItem</code> , <code>PutItem</code> , <code>UpdateItem</code>
<code>Item</code>	<code>PutItem</code>
<code>ScanFilter</code>	<code>Scan</code>

- `dynamodb:ReturnValues` – Represents the `ReturnValues` parameter of a request. `ReturnValues` can be any of the following values:
 - `ALL_OLD`
 - `UPDATED_OLD`
 - `ALL_NEW`
 - `UPDATED_NEW`
 - `NONE`
- `dynamodb:ReturnConsumedCapacity` – Represents the `ReturnConsumedCapacity` parameter of a request. `ReturnConsumedCapacity` can be one of the following values:
 - `TOTAL`
 - `NONE`

In addition to product-specific policy keys, DynamoDB supports the following keys that are common to other AWS services that use AWS Identity and Access Management:

AWS-Wide Policy Keys

- `aws:CurrentTime`–To check for date/time conditions.
- `aws:EpochTime`–To check for date/time conditions using a date in epoch or UNIX time.
- `aws:MultiFactorAuthAge`–To check how long ago (in seconds) the MFA-validated security credentials making the request were issued using Multi-Factor Authentication (MFA). Unlike other keys, if MFA is not used, this key is not present.

- `aws:principalType`—To check the type of principal (user, account, federated user, etc.) for the current request.
- `aws:SecureTransport`—To check whether the request was sent using SSL. For services that use only SSL, such as Amazon RDS and Amazon Route 53, the `aws:SecureTransport` key has no meaning.
- `aws:SourceArn`—To check the source of the request, using the Amazon Resource Name (ARN) of the source. (This value is available for only some services. For more information, see [Amazon Resource Name \(ARN\)](#) under "Element Descriptions" in the *Amazon Simple Queue Service Developer Guide*.)
- `aws:SourceIp`—To check the IP address of the requester. Note that if you use `aws:SourceIp`, and the request comes from an Amazon EC2 instance, the public IP address of the instance is evaluated.
- `aws:UserAgent`—To check the client application that made the request.
- `aws:userid`—To check the user ID of the requester.
- `aws:username`—To check the user name of the requester, if available.

Note

Key names are case sensitive.

For more information about AWS-wide policy keys, see [Condition](#) in [Using IAM](#).

Example Policies for API Actions and Resource Access

Topics

- [Allow any DynamoDB actions on all tables \(p. 531\)](#)
- [Allow read-only access on items in the AWS account's tables \(p. 532\)](#)
- [Allow put, update, and delete operations on one table \(p. 532\)](#)
- [Allow access to a specific table and all of its indexes \(p. 532\)](#)
- [Prevent a partner from using API actions that change data \(p. 533\)](#)
- [Separate test and production environments \(p. 533\)](#)
- [Allow access to the DynamoDB console \(p. 535\)](#)
- [Disallow purchasing of Reserved Capacity offerings \(p. 536\)](#)

This section shows several policies for controlling user access to DynamoDB API actions, and resources such as tables and indexes. For additional policies that address web identity federation and fine-grained access control, see [Example Policies for Fine-Grained Access Control \(p. 538\)](#).

Allow any DynamoDB actions on all tables

In this example, we create a policy that lets the recipient use any DynamoDB API action on any of the AWS account's tables.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "dynamodb:*",  
            "Resource": "*"  
        }  
    ]  
}
```

```
    ]  
}
```

Allow read-only access on items in the AWS account's tables

In this example, we create a policy that lets the recipient use only the `GetItem` and `BatchGetItem` actions with any of the AWS account's tables.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Allow put, update, and delete operations on one table

In this example, we create a policy that lets the recipient use the `PutItem`, `UpdateItem` and `DeleteItem` actions with a table named "Books", which is owned by AWS account number 123456789012.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

Allow access to a specific table and all of its indexes

You may want to limit access of one of your users to a specific table and its indexes.

In this example, we create a policy that gives access to all actions on the table named "Books" and all of its indexes. To test this policy in your own environment, you will need to replace the example account ID "123456789012" with your AWS account ID.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {
```

```
{  
    "Effect": "Allow",  
    "Action": [ "dynamodb:*" ],  
    "Resource": [  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books",  
        "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"  
    ]  
}  
}
```

Prevent a partner from using API actions that change data

IAM Roles provide a way to share a table with another AWS account. For details on creating a role and granting access to another AWS account, see [Roles](#) in the IAM documentation.

In this example, we create an IAM role for the partner, and create a policy for the role that gives access to all the actions except those that edit data; essentially, they have read-only access. Attach the following policy to the IAM role:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb>ListTables",  
                "dynamodb>DescribeTable",  
                "dynamodb>GetItem",  
                "dynamodb>BatchGetItem",  
                "dynamodb>Query",  
                "dynamodb>Scan"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Note

This example uses an `Allow` action and enumerates each of the "read" actions supported by DynamoDB. An alternative approach could use a `Deny` action and enumerate each of the "write" actions. However, the illustrated `Allow` approach is recommended because new "write" actions may be added to DynamoDB in the future, which could result in unintended access with the `Deny` approach.

Separate test and production environments

Suppose you maintain separate test and production environments where each environment maintains its own version of a table named `ProductCatalog`. If you create these tables from the same AWS account, testing work might affect the production environment, because, 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, you might consider creating separate AWS accounts for the production and testing.

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. You might create a copy of the same table for each developer, such as Bob_ProductCatalog and Alice_ProductCatalog . In this case, you can create IAM users Bob and Alice in the AWS account that you created for the test environment. You can then allow these users to perform DynamoDB actions on to the tables that they own. You have the following policy options to grant the user permissions:

- Create a separate policy for each user and attach the policy to the users 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": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:*"],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Alice_ProductCatalog"
        }
    ]
}
```

You would then create a similar policy with a different resource (Bob_ProductCatalog table) for user Bob.

To test these policies in your own environment, you will need to replace the example account ID "123456789012" with your AWS account ID. If you want to test them on the DynamoDB management console, the console requires permission for additional DynamoDB, CloudWatch, and Amazon Simple Notification Service actions, as shown in the following policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:*"],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Alice_ProductCatalog"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb>ListTables",
                "dynamodb>DescribeTable",
                "cloudwatch:*",
                "sns:>"
            ],
            "Resource": "*"
        }
    ]
}
```

- 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 will need to create a group and, for this example, add both users Alice and user Bob to the group. The following example allows 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 item to the `Alice_ProductCatalog` table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:*"],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_ProductCatalog"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb>ListTables",
                "dynamodb>DescribeTable",
                "cloudwatch:/*",
                "sns:/*"
            ],
            "Resource": "*"
        }
    ]
}
```

Note

When using IAM 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.

Note that, instead of identifying a specific table as a resource, you could use a wild card (*) to grant permissions on all tables whose name is prefixed with the name of the IAM user who is logged making the request.

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_*"
```

Allow access to the DynamoDB console

In this example, we create a policy that lets the recipient work with the DynamoDB console.

Even if you grant access to individual DynamoDB tables, members of the group will not be able to list all the tables or view CloudWatch metrics in the DynamoDB console. The following policy provides minimal permissions for using the console, but does not allow access to any data in the DynamoDB tables.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb>DescribeTable",
                "dynamodb>ListTables",
                "cloudwatch>DescribeAlarms",
                "cloudwatch>ListMetrics"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

```
        "Resource": "*"
    ]
}
```

Disallow purchasing of Reserved Capacity offerings

In this example, we create a policy to prevent users from purchasing Reserved Capacity offerings.

Recipients of this policy can still view existing Reserved Capacity purchases using the DynamoDB console; however, they won't be able to purchase any new Reserved Capacity for DynamoDB.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "*"
        }
    ]
}
```

Fine-Grained Access Control for DynamoDB

In addition to controlling access to DynamoDB API actions, you can also control access to individual data items and attributes. *Fine-grained access control* is the ability to determine who can access individual data items and attributes in DynamoDB tables and indexes, and the actions that can be performed on them.

To implement fine-grained access control, you write an AWS Identity and Access Management (IAM) 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.

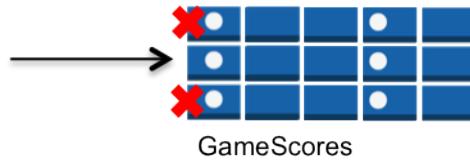
You can optionally use web identity federation to control access by users who are authenticated by Login with Amazon, Facebook, or Google. For more information, see [Using Web Identity Federation \(p. 544\)](#)

Here are some possible use cases for fine-grained access control:

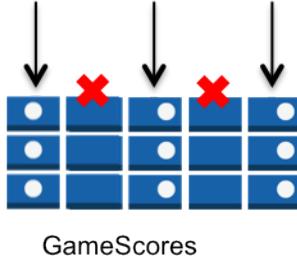
- An app that displays flight data for nearby airports, based on the user's location. Airline names, arrival and departure times, and flight numbers are all displayed; however, attributes such as pilot names or the number of passengers are restricted.
- A social networking app for games, where all users' saved game data is stored in a single table, but no user can access data items that they do not own.
- An app for targeted advertising that stores data about ad impressions and click tracking. The app can only write data items for the current user, and can only retrieve targeted ad recommendations for that user.

An IAM Condition element is the central feature of a fine-grained access control policy. By adding a Condition to a policy, you can allow or deny access to items and attributes in DynamoDB tables and indexes, based upon your particular business requirements.

Using fine-grained access control, you can "hide" information in a DynamoDB table in a horizontal fashion by matching primary key values:



To hide information vertically, you can control which attributes are visible by listing those values in a policy document:



You can also implement horizontal and vertical information hiding in the same policy.

To show how fine-grained access control works, consider a mobile gaming app. This app 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. Users should only have access to game data associated with their user ID.

Here is the primary key for *GameScores*:

Table Name	Primary Key Type	Hash Attribute Name and Type	Range Attribute Name and Type
GameScores (<u>UserId</u> , <u>GameTitle</u> , ...)	Hash and Range	Attribute Name: UserId Type: String	Attribute Name: Game- Title Type: String

A user that wants to play a game must belong to an IAM role named *GameRole*, which has a security policy attached to it. Here is the policy document:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ]
        }
    ]
}
```

```

        ],
        "Resource": [
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
        ],
        "Condition": {
            "ForAllValues:StringEquals": {
                "dynamodb:LeadingKeys": ["${www.amazon.com:user_id}"],
                "dynamodb:Attributes": [
                    "UserId", "GameTitle", "Wins", "Losses",
                    "TopScore", "TopScoreDateTime"
                ]
            },
            "StringEqualsIfExists": { "dynamodb:Select": "SPECIFIC_ATTRIBUTES" }
        }
    }
}
]
}
}

```

The `Condition` clause implements fine-grained access control, hiding information both horizontally and vertically:

- The `dynamodb:LeadingKeys` policy key lets a user access items where the hash key value matches their user identifier. This identifier is provided by the `${www.amazon.com:user_id}` substitution variable. (More information about such substitution variables is presented in [Using Web Identity Federation \(p. 544\)](#).)
- The `dynamodb:Attributes` policy key allows access to only a subset of attributes in any item. No other attributes are returned by any of the listed actions. In addition, the `StringEqualsIfExists` clause ensures that the app must always provide a list of specific attributes to act upon. It is not possible to request all attributes.

When an IAM policy is evaluated, the result will always be true (access is allowed) or false (access is denied). If any part of the `Condition` element is false, then 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 the policy. Otherwise, DynamoDB will not be able to use these key attributes to perform the requested action.

The attribute names and leading key names in an IAM policy cannot have any characters other than TAB, CR, LF, SPACE, and ASCII codes 33 (0x21) through 255 (0xFF). Please refer to the ASCII table at <http://www.asciitable.com>.

Example Policies for Fine-Grained Access Control

This section shows several policies for implementing fine-grained access control on DynamoDB tables.

Many of these policies allow users to access only those items in a table where the hash key value matches the user identifier. 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 \(p. 544\)](#).

Note

In each of these examples, we set the `Effect` clause to `Allow` and specify only the actions, resources and parameters that will be 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.

Limit access to items with a specific hash key value

In this example, we create a policy that allows an authenticated user to perform DynamoDB actions on `GameScores`, but only on the items whose hash key values match the Login with Amazon unique user ID for this app. Note that `Scan` is not included in the list of actions, because `Scan` would provide access to *all* of the leading keys.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "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.

If we wanted to implement read-only access, we could remove any actions that could 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": [
        {
            "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}" ] }
    }
}
}

```

We can also filter the data items that are returned, so that the user only sees specific attributes. In the following policy, only *GameTitle* and *TopScore* will be returned.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "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}" ],
                    "dynamodb:Attributes": [ "GameTitle", "TopScore" ]
                },
                "StringEquals": { "dynamodb>Select": "SPECIFIC_ATTRIBUTES" }
            }
        }
    ]
}
```

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 the policy. Otherwise, DynamoDB will not be able to use these key attributes to perform the requested action.

Limit access to specific attributes in a table

In this example, we create a policy that permits access to only two specific attributes in a table. These attributes can be read, written, or evaluated in a conditional write or scan filter.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "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, allowing access to a named set of attributes. It is possible to write an equivalent policy that disallows access to other attributes instead. This "blacklist" approach is not recommended, because users could determine the names of these blacklisted attributes by repeatedly issuing requests for all possible attribute names, eventually finding an attribute that they are not allowed access to. To avoid this, follow the [principle of least privilege](#) and use a whitelist approach to enumerate all of the allowed values, rather than specifying the disallowed attribute names.

This policy does not permit `PutItem`, `DeleteItem` or `BatchWriteItem`, because those actions always replace the entire previous item. This would allow users to delete the previous values for attributes that they are not allowed to access.

The `StringEqualsIfExists` clause ensures the following:

- If the user specifies the `Select` parameter, then its value must be `SPECIFIC_ATTRIBUTES`. This prevents the API action from returning any non-allowed attributes, 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 because the `UpdateItem` action also performs implicit reads, to check whether an item exists before replacing it, or 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.

Here are some important considerations about the limits of fine-grained access policy controls:

- If you are limiting access to certain attributes, you must allow the primary key attribute names. If you want to allow `Query` actions on a secondary index, you must also allow the index range key attribute name.
- `UpdateItem` is actually an "upsert", meaning that if an item does not exist with the specified primary key, then DynamoDB will put a new item into the table. This implies that users could put new items

without having any of the whitelisted attributes defined. To avoid this, ensure that your applications are designed to expect upsert behavior, and that they will not be vulnerable to any kinds of exploits from it.

The following are some variations on this policy:

- To allow only read actions, we can remove `UpdateItem` from the list of allowed actions. Since none of the remaining actions accept `ReturnValues`, we can remove `ReturnValues` from the condition. We can also change `StringEqualsIfExists` to `StringEquals`, since the `Select` parameter will always have a value (`ALL_ATTRIBUTES`, unless otherwise specified).
- To allow only write actions, we can remove everything except `UpdateItem` from the list of allowed actions. Since `UpdateItem` does not use the `Select` parameter, we can remove `Select` from the condition. We must also change `StringEqualsIfExists` to `StringEquals`, since the `ReturnValues` parameter will always have 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: "`*`".

Hide certain attributes on write actions

In this example, we create a policy that allows `UpdateItem` actions only on a limited number of attributes in the `GameScores` table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "dynamodb:UpdateItem" ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
            "Condition": {
                "ForAllValues:StringNotLike": {
                    "dynamodb:Attributes": [
                        "FreeGamesAvailable",
                        "BossLevelUnlocked"
                    ]
                },
                "StringEquals": {
                    "dynamodb:ReturnValues": [
                        "NONE",
                        "UPDATED_OLD",
                        "UPDATED_NEW"
                    ]
                }
            }
        }
    ]
}
```

In this policy, `PutItem` and `DeleteItem` are not allowed. These actions always replace the entire item, which means there is no way to prevent a user from modifying other attributes. Like other write actions, in DynamoDB, `UpdateItem` also performs a read, in order to return the values as they appeared either before or after they were updated. To ensure that other attributes are not seen, the calling application

must set the `ReturnValues` parameter to `NONE`, `UPDATED_OLD`, or `UPDATED_NEW`. (`ALL_OLD` and `ALL_NEW` are not included in this policy.)

Query only projected attributes in an index

In this example, we create a policy that allows queries on a secondary index named `TopScoreDateTimeIndex`, but only for queries that request specific attributes that have been projected into the index.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "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 policy is similar, but the query must request all of the attributes that have been projected into the index.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "dynamodb:Query" ],
            "Resource": [ "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/TopScoreDateTimeIndex" ],
            "Condition": { "StringEquals": { "dynamodb:Select": "ALL_PROJECTED_ATTRIBUTES" } }
        }
    ]
}
```

Limit access to certain attributes and hash key values

In this example, we create a policy that allows access only to items in the `GameScores` table and `TopScoreDateTimeIndex`, with specific hash key values.

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
{
    "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" : [ "allowed_attribute", "another_allowed_attribute" ]
        },
        "StringEqualsIfExists": {
            "dynamodb:Select": "SPECIFIC_ATTRIBUTES",
            "dynamodb:ReturnValues": [
                "NONE",
                "UPDATED_OLD",
                "UPDATED_NEW"
            ]
        }
    }
}
}
```

In this policy:

- The user can only access items where the hash key values match the unique identifier from Facebook. The identifier is unique for this user and this application.
- Write actions can only modify `allowed_attribute` or `another_allowed_attribute`. To prevent other attributes from being modified, `PutItem`, `DeleteItem` and `BatchWriteItem` are omitted from the list of allowed actions. However, an application can insert new items using `UpdateItem`, and those 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 would cause fetches from the table.
- Applications cannot read any attributes other than those listed in `dynamodb:Attributes` (`allowed_attribute` and `another_allowed_attribute`). An application must now 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 writes based on any other attributes.

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 [Fine-Grained Access Control for DynamoDB \(p. 536\)](#). Here is the primary key for *GameScores*:

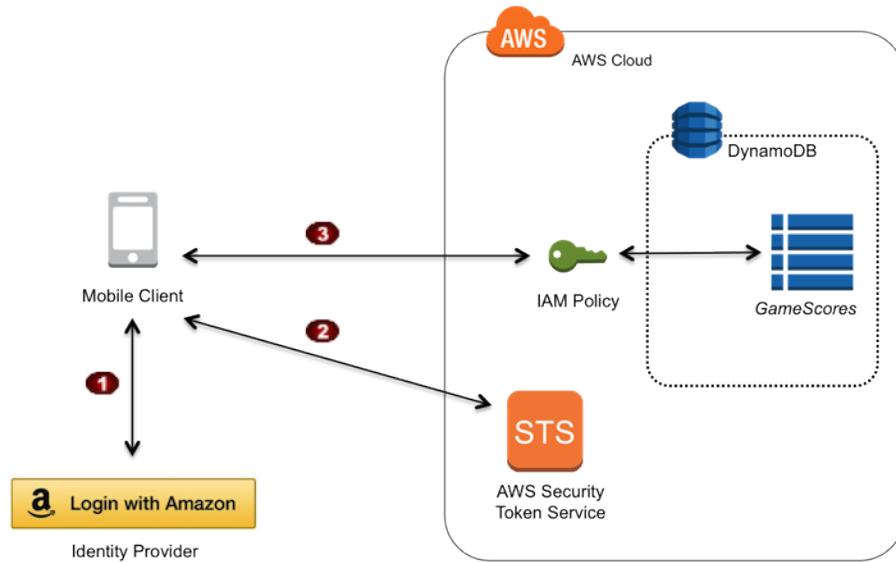
Table Name	Primary Key Type	Hash Attribute Name and Type	Range Attribute Name and Type
GameScores (<u>UserId</u> , <u>GameTitle</u> , ...)	Hash and Range	Attribute Name: UserId Type: String	Attribute Name: GameTitle Type: String

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.



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 [Fine-Grained Access Control for DynamoDB \(p. 536\)](#):

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": ["${www.amazon.com:user_id}"],
                    "dynamodb:Attributes": [
                        "UserId", "GameTitle", "Wins", "Losses",
                    ]
                }
            }
        }
    ]
}
```

```
        "TopScore", "TopScoreDateTime"
    ],
},
"StringEqualsIfExists": {"dynamodb:Select": "SPECIFIC_ATTRIBUTES"}
}
}
]
}
```

The Condition clause determines which items in *GameScores* are visible to the app. It does this by comparing the Login with Amazon ID to the *UserId* hash key values in *GameScores*. Only the items belonging to the current user can be processed using one of DynamoDB actions that are listed in this policy; other items in the table cannot be accessed. Furthermore, only the specific attributes listed in the policy can be accessed.

Preparing to Use Web Identity Federation

If you are an application developer and want to use web identity federation for your app, follow these steps:

1. **Sign up as a developer with a third-party identity provider.** The following external links provide information about signing up with supported identity providers:
 - [Login with Amazon Developer Center](#)
 - [Registration](#) on the Facebook site
 - [Using OAuth 2.0 to Access Google APIs](#) on the Google site
2. **Register your app with the identity provider.** When you do this, the provider gives you an ID that's unique to your app. If you want your app to work with multiple identity providers, you will need to obtain an app ID from each provider.
3. **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, go to [Creating Temporary Security Credentials for Mobile Apps Using Identity Providers](#) in [Using Temporary Security Credentials](#).

Note

As an alternative to AWS Security Token Service, you can use Amazon Cognito. Amazon Cognito is now the preferred service for managing temporary credentials for mobile apps. For more information, go to the following pages:

- [How to Authenticate Users \(AWS SDK for iOS\)](#)
- [How to Authenticate Users \(AWS SDK for Android\)](#)

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.

To generate an IAM policy using the DynamoDB console

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. In the Tables pane, click the table you want to create the policy for, and then click **Access Control**. The **Table Access Permissions** wizard opens.



3. In the **Set Permissions** pane, choose the identity provider, actions, and attributes for the policy.



When the settings are as you want them, click **Continue**.

4. In the **Review** pane, the generated policy appears in the **Policy Document** field.



You can now go to the IAM console to create a new IAM role. For step-by-step instructions, see [Creating a Role for Web Identity Federation](#) section in [Using IAM](#). You will need to specify the third-party identity provider you want to use, along with the app ID that you obtained from that provider. When you are asked to set permissions for the role, choose **Custom Policy** and paste your DynamoDB policy in the **Policy Document** field.

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.ac
    count.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>AROAJU4SA2VW5SZRF2YMG: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 for Fine-Grained Access Control \(p. 538\)](#).

For more information about how AWS Security Token Service generates temporary access credentials, go to [Creating Auth Tokens in Using Temporary Security Credentials](#).

Exporting, Importing and Transforming Data Using AWS Data Pipeline

Topics

- [Using the AWS Management Console to Export and Import Data \(p. 550\)](#)
- [Predefined Templates for AWS Data Pipeline and DynamoDB \(p. 564\)](#)

You can use AWS Data Pipeline to automate data movement and transformation into and out of Amazon DynamoDB. The built-in scheduling capabilities of AWS Data Pipeline let you schedule and execute recurring jobs, without having to write your own complex data transfer or transformation logic. For example, you can set up a recurring job to automatically copy data from DynamoDB into Amazon Redshift. As another example, you can copy data from DynamoDB into Amazon S3, and then analyze the data using a statistics program (such as [R](#)) running on Amazon EC2.

Tip

The DynamoDB console provides a point-and-click interface for exporting data to Amazon S3, and for importing data from Amazon S3 to DynamoDB. Using the console is the easiest way to leverage AWS Data Pipeline for exporting and importing DynamoDB data. For more information, see [Using the AWS Management Console to Export and Import Data \(p. 550\)](#).

Using the AWS Management Console to Export and Import Data

You can use the AWS Management Console to export data from one or more DynamoDB tables 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.

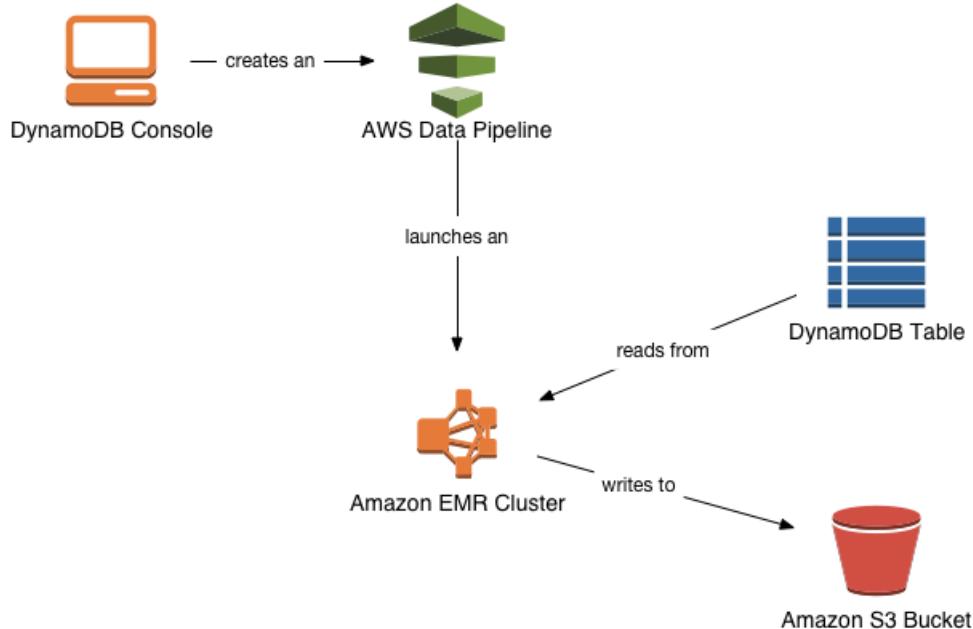
You can even set up an export job to copy data from a DynamoDB table in one AWS region, store the data in Amazon S3, and then import the data from Amazon S3 to an identical DynamoDB table in a second region. Applications in the second region could then connect to their nearest DynamoDB endpoint and work with their own copy of the data, with reduced network latency.

The AWS Management Console lets you easily perform exports and imports, without having to manually create an AWS Data Pipeline or provision and maintain an Amazon EMR cluster. The console automates these steps for you, and lets you monitor the progress of your export and import jobs.

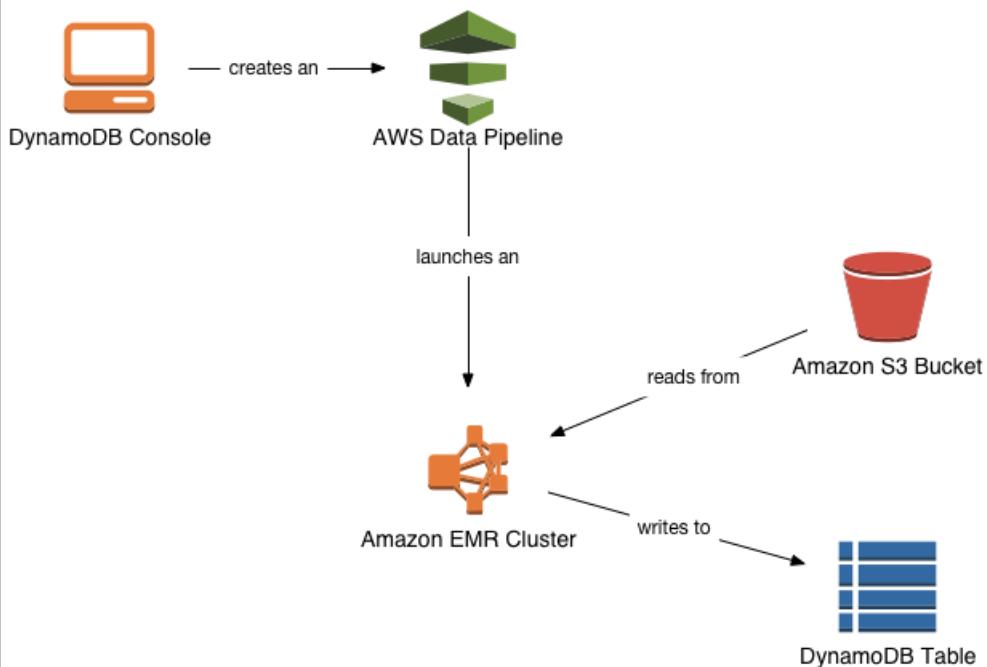
Overview of the Export and Import Process

To perform exports and imports, you use an AWS Data Pipeline and an Amazon Elastic MapReduce cluster (Amazon EMR). The DynamoDB console automates these tasks for you, and lets you monitor the progress of your export and import jobs. The following shows an overview of the process.

Exporting Data from DynamoDB to Amazon S3



Importing Data from Amazon S3 to DynamoDB



To export a table, you use the console to create a new AWS Data Pipeline. The pipeline, in turn, launches an Amazon EMR cluster to perform the actual export. The Amazon EMR cluster reads the data from a table in 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 Elastic MapReduce (Amazon EMR)**— runs a managed Hadoop cluster to performs reads and writes between DynamoDB to Amazon S3. The cluster configuration is one `m1.small` instance master node and one `m1.xlarge` instance task node.

For more information see [AWS Data Pipeline Pricing](#), [Amazon EMR Pricing](#), and [Amazon S3 Pricing](#).

Prerequisites to Export and Import Data

Topics

- [Creating IAM Roles for AWS Data Pipeline \(p. 553\)](#)
- [Granting IAM Users and Groups Permission to Perform Export and Import Tasks \(p. 554\)](#)

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.

If you are an administrator for your DynamoDB tables, you can optionally configure IAM policies and attach them to IAM users or groups in your AWS account. These policies let you specify which users can import and export your DynamoDB table 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, go to [Administering Access Keys for IAM Users](#) in *Using IAM*.

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 \(p. 557\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 558\)](#).

To create *DataPipelineDefaultRole* and *DataPipelineDefaultResourceRole*

1. Sign in to the AWS Management Console and open the IAM console at <https://console.amazonaws.cn/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 **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 **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 \(p. 557\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 558\)](#).

Granting IAM Users and Groups Permission to Perform Export and Import Tasks

If you are a DynamoDB administrator and 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.

To attach the *AmazonDynamoDBFullAccesswithDataPipeline* policy to an IAM user

1. Sign in to the AWS Management Console and open the IAM console at <https://console.amazonaws.cn/iam/>.
2. From the IAM Console Dashboard, click **Users** and select the user you want to modify.
3. In the **Permissions** panel, 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 are a DynamoDB administrator and you want to restrict access, so that a user can only perform export and import tasks on 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.

To create a custom policy

1. Sign in to the AWS Management Console and open the IAM console at <https://console.amazonaws.cn/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 API actions on all of your tables:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "cloudwatch:DeleteAlarms",  
                "cloudwatch:DescribeAlarmHistory",  
                "cloudwatch:DescribeAlarms",  
                "cloudwatch:DescribeAlarmsForMetric",  
                "cloudwatch:GetMetricStatistics",  
                "cloudwatch>ListMetrics",  
                "cloudwatch:PutMetricAlarm",  
                "dynamodb:*",  
                "sns:CreateTopic",  
                "sns>DeleteTopic",  
                "sns>ListSubscriptions",  
                "sns>ListSubscriptionsByTopic",  
                "sns>ListTopics",  
                "sns:Subscribe",  
                "sns:Unsubscribe"  
            ],  
            "Effect": "Allow",  
            "Resource": "*",  
            "Sid": "DDBConsole"  
        },  
    ]  
}
```

...remainder of document omitted...

To restrict the policy, first remove the following line:

```
"dynamodb:*",
```

Next, construct a new Action that allows access to only the *Forum*, *Thread* and *Reply* tables:

```
{
    "Action": [
        "dynamodb:*"
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/Forum",
        "arn:aws:dynamodb:us-west-2:123456789012:table/Thread",
        "arn:aws:dynamodb:us-west-2:123456789012:table/Reply"
    ]
},
```

Note

Replace `us-west-2` with the region in which your DynamoDB tables reside. Replace `123456789012` with your AWS account number. For more information, see [Amazon Resource Names \(ARNs\) for DynamoDB \(p. 528\)](#).

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:ListTopics"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:sns:us-west-2:123456789012:MyTopic"
            ]
        }
    ]
},
```

```
        "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**.

Now that you have created the policy, you can attach it to an IAM user.

To attach the custom policy to an IAM user

1. From the IAM Console Dashboard, click **Users** and select the user you want to modify.
2. In the **Permissions** panel, click **Attach Policy**.
3. In the **Attach Policy** panel, select the name of your policy and click **Attach Policy**.

Note

You can use a similar procedure to attach your policy to a 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 \(p. 553\)](#).

To export data from DynamoDB to Amazon S3

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. On the **Amazon DynamoDB Tables** page, click **Export/Import**.
3. On the **Export/Import** page, select one or more source tables containing data you want to export, and then click **Export from DynamoDB**.
4. On the **Create Export Table Data Pipeline(s)** page, do the following:
 - a. In the **S3 Output 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`.
- b. In the **S3 Log Folder** 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 **S3 Output Folder**. The URI must resolve to a folder; log files cannot be written to the top level of the S3 bucket.
- c. In the **Throughput Rate** text box, choose a percentage from the drop-down list. This percentage will be used during the export process to regulate the amount of provisioned read throughput consumed. For example, suppose you are exporting a source table that has a `ReadCapacityUnits` setting of 20, and you set the throughput rate percentage to 40%. The export job will consume no more than 8 read capacity units per second from your table's provisioned read throughput.
- If you are exporting multiple tables, the **Throughput Rate** will be applied to each of the tables during the export process.
- d. In the **Execution Timeout** text box, enter the number of hours after which the export job will time out. If the job has not completed within this period, it will fail.
- e. In the **Send notifications to** text box, enter your email address. After the pipeline is created, you will receive an email message inviting you to subscribe to a topic in Amazon Simple Notification Service (Amazon SNS); if you accept this invite, you will receive periodic notifications via email as the export progresses.
- f. In the **Schedule** section, choose one of the following:
- **One-time Export** — the export will begin immediately after the pipeline is created.
 - **Daily Export** — the export will begin at the time of day you specify, and will be repeated every day at that time.
- g. In the **Data Pipeline Role field**, select **DataPipelineDefaultRole**.
- h. In the **Resource Role** field, select **DataPipelineDefaultResourceRole**

When the settings are as you want them, click **Create Export Pipeline**.

Your pipeline will now be created; this process can take several minutes to complete. To view the current status, see [Managing Export and Import Pipelines \(p. 560\)](#).

If you chose a one-time export, the export job will begin immediately after the pipeline has been created. If you chose a daily export, the job will begin at the time you have selected, and will repeat at that time each day.

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*.

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 name as the source table.
- 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.

Important

If you have never used AWS Data Pipeline before, you will need to set up two AWS Identity and Access Management roles before following this procedure. For more information, see [Creating IAM Roles for AWS Data Pipeline \(p. 553\)](#).

To import data from Amazon S3 to DynamoDB

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. (Optional) If you want to perform a cross-region import, click **Select a Region** in the upper right-hand corner of the console and click the destination region. The console will display all of the DynamoDB tables in that region. If the destination table does not already exist, you can create it using the console.
3. On the **Amazon DynamoDB Tables** page, click **Export/Import**.
4. On the **Export/Import** page, select the destination table for your data, and then click **Import into DynamoDB**.
5. On the **Create Import Table Data Pipeline** page, do the following:
 - a. In the **S3 Input 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*.

- b. In the **S3 Log Folder** 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 **S3 Input Folder**. The URI must resolve to a folder; log files cannot be written to the top level of the S3 bucket.

- c. In the **Throughput Ratio** text box, choose a percentage from the drop-down list. This percentage will be used during the import process to regulate the amount of provisioned write throughput consumed. For example, suppose you are importing data into a table that has a *WriteCapacityUnits* setting of 10, and you set the throughput ratio percentage to 60%. The import job will consume no more than 6 write capacity units per second from your table's provisioned write throughput.
- d. In the **Execution Timeout** text box, enter the number of hours after which the import job will time out. If the job has not completed within this period, it will fail.
- e. In the **Send notifications to** text box, enter your email address. After the pipeline is created, you will receive an email message inviting you to subscribe to a topic in Amazon Simple Notification Service (Amazon SNS); if you accept this invite, you will receive periodic notifications via email as the import proceeds.
- f. In the **Associate pipeline with a role** section, in **Data Pipeline Role**, select **DataPipelineDefaultRole**.
- g. In the **Associate pipeline compute resources with a role** section, in **Resource Role**, select **DataPipelineDefaultResourceRole**

When the settings are as you want them, click **Create Export Pipeline**.

Your pipeline will now be created; this process can take several minutes to complete. To view the current status, see [Managing Export and Import Pipelines \(p. 560\)](#).

The import job will begin immediately after the pipeline has been created.

Managing Export and Import Pipelines

Topics

- [Canceling an Export or Import Job \(p. 562\)](#)
- [Deleting a Pipeline \(p. 562\)](#)

You can use the AWS Management Console to monitor the progress of pipeline creation, as well as import and export jobs. The **Export/Import** page displays all of your DynamoDB tables, along with any export and import pipelines that may be present.

Note

This page displays only the export and import pipelines that you have created in the DynamoDB console. If you have created other pipelines using the AWS Data Pipeline, those pipelines will not be shown here.

To view all of your pipelines, including those that are not used for DynamoDB export, and imports, go to the [AWS Data Pipeline console](#).

Each pipeline has a **Pipeline Status** field; the status changes in response to events within that pipeline. To get more details about a pipeline, click its name:

Amazon DynamoDB Developer Guide

Using the AWS Management Console to Export and Import Data

Table Name	Export Pipelines	Import Pipelines
Forum		
ProductCatalog	[DDB OneTime Export] ProductCatalog to s3://mybucket/Forum-backup to Forum 2014-03-03T18:19:41 Pipeline State: SCHEDULED Execution Status: EmrActivity: WAITING_O	
Reply		
Thread		[DDB OneTime Import] s3://mybucket/Forum-backup to Forum 2014-03-03T18:19:41 Pipeline State: SCHEDULING Execution Status: Not started
TransactionExamples		

When you do this, a **Data Pipeline Detail** page is shown:

Data Pipeline Details

Pipeline ID: df-03389393VFN5W9E29Y3C
Pipeline Name: [DDB OneTime Import] s3://mybucket/Forum-backup to Forum 2014-03-03T18:19:41
Description: This pipeline will execute a one-time import from S3 directory s3://mybucket/Forum-backup into the DynamoDB table 'Forum'
DynamoDB Table Name: Forum
S3 Directory: s3://mybucket/Forum-backup
Schedule: startDateTime: 03 Mar 2014 10:19:41 GMT-08:00, period: 1 Day, endDateTime: 04 Mar 2014 10:19:41 GMT-08:00
EMR Cluster Configuration:

EmrCluster emrLogUri:	s3://mybucket/Forum-log/import/us-east-1/Forum/# {format(@scheduledStartTime,'YYYY-MM-dd_hh.mm')};
masterInstanceType:	m1.small
coreInstanceType:	m1.xlarge
enableDebugging:	true
installHive:	latest
coreInstanceCount:	1
EmrActivity maximumRetries:	0
myDynamoDBWriteThroughputRatio:	0.25
attemptTimeout:	24 hours
step:	s3://elasticmapreduce/libs/script-runner/script-runner.jar;s3://elasticmapreduce/libs/hive/hive-script,--run-hive-script,--hive-versions,latest,--args,-f,s3://elasticmapreduce/libs/hive/dynamodb/importDynamoDBTableFromS3,-d,DYNAMODB_OUTPUT_TABLE=#{output.tableName},-d,S3_INPUT_BUCKET=#{input.directoryPath},-d,DYNAMODB_WRITE_PERCENT=#{myDynamoDBWriteThroughputRatio},-d,DYNAMODB_ENDPOINT=dynamodb.us-east-1.amazonaws.com

Pipeline Execution History:

Type	Status	Scheduled Start Time	Actual Start Time	Actual End Time
EmrActivity	WAITING_ON_DEPENDENCIES	03 Mar 2014 10:19:41 GMT-08:00	03 Mar 2014 10:20:29 GMT-08:00	

To see all of the configuration details of the pipeline, or to delete or cancel the pipeline, visit the [Data Pipeline Console](#).

[Back](#)

This page provides a summary of the pipeline's events and its current status.

If you need more information, go to the lower right-hand corner and click **Data Pipeline Console**. This will open the AWS Data Pipeline console, where you can review the pipeline and all of its tasks in detail.

To return to the **Export/Import** page, click **Back**.

Canceling an Export or Import Job

At any time, you can cancel a running export or import job. This will terminate the Amazon EMR processing, and prevent any further reads and writes on your DynamoDB table(s).

To cancel an export or import job

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. On the **Amazon DynamoDB Tables** page, click **Export/Import**.
3. On the **Export/Import** page, go to the pipeline and click its name.
4. On the **Data Pipeline Detail** page, go to the lower right-hand corner and click **Data Pipeline Console**.
5. In the AWS Data Pipeline console, look for a task that has a status of **RUNNING**. (This will most likely be a task of type **EmrActivity**; however, you can cancel any task in the pipeline using the console.).
6. In the **Actions** drop-down list, click **Cancel**.

Note

Even if you cancel a task, the pipeline itself will remain intact. If the pipeline is for a daily export, the job will restart the next day at its scheduled time.

Deleting a Pipeline

If you create an export or import pipeline, it remains available until you delete it. You can delete a pipeline at any time using the DynamoDB console. In addition, if you delete a table using the console, you can also delete all of the table's pipelines at that time.

Note that if an import or export job is still running, the job will be cancelled first before its pipeline is deleted.

To delete a single pipeline

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. On the **Amazon DynamoDB Tables** page, click **Export/Import**.
3. On the **Export/Import** page, go to the pipeline and click its name.
4. On the **Data Pipeline Detail** page, go to the lower right-hand corner and click **Delete This Pipeline**.
5. In the confirmation dialog, click **Yes**.

The pipeline and all resources associated with it will be deleted.

Note

Each DynamoDB table can have a maximum of three pipelines associated with it:

1. A one-time export pipeline.
2. A daily export pipeline.
3. A one-time import pipeline.

If you want to perform a one-time export or import, you must first delete any existing pipeline of that type for the table, and then re-create the pipeline. Similarly, if you currently have a daily export pipeline but want to perform the tasks at a different time, you will need to delete and then re-create the existing pipeline.

After you delete a table's pipeline, you can create a new one of the same type. See [Exporting Data From DynamoDB to Amazon S3 \(p. 557\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 558\)](#) for instructions on how to create new pipelines.

If you delete a table using the DynamoDB console, you can also delete all of the pipelines associated with that table.

To delete a table and all of its pipelines

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.amazonaws.cn/dynamodb/>.
2. Select the table that you want to delete.
3. Click **Delete Table** in the Tables wizard.
4. In the **Delete Table** confirmation window, select **Delete all export/import pipelines for this table** and then click **Delete**.

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 DynamoDB console will display as FAILED. If this happens, click the name of the failed pipeline to go to the **Data Pipeline Detail** page. On that page, take note of any errors that you see.

Next, go to the lower right-hand corner and click **Data Pipeline Console**. This will show details about all of the steps in the pipeline, and the status of each one.

- *DynamoDBDataNode* — represents the DynamoDB table to be exported.
- *S3DataNode* — represents the Amazon S3 bucket where the export file is to be written.
- *EmrCluster* — represents the provisioning process for the Amazon EMR cluster that will perform the export.
- *EmrActivity* — represents the Amazon EMR cluster running the export job.

In the AWS Data Pipeline console, click each of these steps and take note of any errors. In particular, examine the execution stack traces and look for errors there.

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 \(p. 553\)](#).
- 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*.

Tip

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](#)

Cross-Region DynamoDB Copy

This AWS Data Pipeline console template lets you configure periodic movement of data between DynamoDB instances across different regions or to a different table within the same region. This feature is useful in the following scenarios:

- Disaster recovery in the case of data loss or region failure
- Moving Amazon DynamoDB data across regions to support applications in those regions
- Performing full or incremental Amazon DynamoDB data backups

For more information about this template, see [Cross-Region DynamoDB Copy](#) in the [AWS Data Pipeline Developer Guide](#).

Querying and Joining Tables Using Amazon Elastic MapReduce

Topics

- [Prerequisites for Integrating Amazon EMR with DynamoDB \(p. 566\)](#)
- [Step 1: Create a Key Pair \(p. 566\)](#)
- [Step 2: Create a Cluster \(p. 567\)](#)
- [Step 3: SSH into the Master Node \(p. 570\)](#)
- [Step 4: Set Up a Hive Table to Run Hive Commands \(p. 572\)](#)
- [Hive Command Examples for Exporting, Importing, and Querying Data in DynamoDB \(p. 576\)](#)

- [Optimizing Performance for Amazon EMR Operations in DynamoDB \(p. 583\)](#)
- [Walkthrough: Using DynamoDB and Amazon Elastic MapReduce \(p. 586\)](#)

In the following sections, you will learn how to use Amazon Elastic MapReduce (Amazon EMR) with a customized version of Hive that includes connectivity to Amazon DynamoDB to perform operations on data stored in DynamoDB, such as:

- Exporting data stored in DynamoDB to Amazon S3.
- Importing data in Amazon S3 to DynamoDB.
- Querying live DynamoDB data using SQL-like statements (HiveQL).
- Joining data stored in DynamoDB and exporting it or querying against the joined data.
- Loading DynamoDB data into the Hadoop Distributed File System (HDFS) and using it as input into an Amazon EMR job flow.

To perform each of the tasks above, you'll launch an Amazon EMR job flow, specify the location of the data in DynamoDB, and issue Hive commands to manipulate the data in DynamoDB.

Amazon EMR runs Apache Hadoop on Amazon EC2 instances. Hadoop is an application that implements the map-reduce algorithm, in which a computational task is mapped to multiple computers that work in parallel to process a task. The output of these computers is reduced together onto a single computer to produce the final result. Using Amazon EMR you can quickly and efficiently process large amounts of data, such as data stored in DynamoDB. For more information about Amazon EMR, go to the [Amazon Elastic MapReduce Developer Guide](#).

Apache Hive is a software layer that you can use to query map reduce job flows using a simplified, SQL-like query language called HiveQL. It runs on top of the Hadoop architecture. For more information about Hive and HiveQL, go to the [HiveQL Language Manual](#).

There are several ways to launch an Amazon EMR job flow: you can use the AWS Management Console Amazon EMR tab, the Amazon EMR command-line interface (CLI), or you can program your job flow using the AWS SDK or the API. You can also choose whether to run a Hive job flow interactively or from a script. In this document, we will show you how to launch an interactive Hive job flow from the console and the CLI.

Using Hive interactively is a great way to test query performance and tune your application. Once you have established a set of Hive commands that will run on a regular basis, consider creating a Hive script that Amazon EMR can run for you. For more information about how to run Hive from a script, go to [How to Create a Job Flow Using Hive](#).

Warning

Amazon EMR read and write operations on a DynamoDB table count against your established provisioned throughput, potentially increasing the frequency of provisioned throughput exceptions. For large requests, Amazon EMR implements retries with exponential backoff to manage the request load on the DynamoDB table. Running Amazon EMR jobs concurrently with other traffic may cause you to exceed the allocated provisioned throughput level. You can monitor this by checking the **ThrottleRequests** metric in CloudWatch. If the request load is too high, you can relaunch the job flow and set [Read Percent Setting \(p. 584\)](#) and [Write Percent Setting \(p. 584\)](#) to lower values to throttle the Amazon EMR read and write operations. For information about DynamoDB throughput settings, see [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

Note

The integration of DynamoDB with Amazon EMR does not currently support Binary and Binary Set type attributes.

Prerequisites for Integrating Amazon EMR with DynamoDB

To use Amazon EMR (Amazon EMR) and Hive to manipulate data in DynamoDB, you need the following:

- An Amazon Web Services account. If you do not have one, you can get an account by going to <http://www.amazonaws.cn>, and clicking **Create an AWS Account**.
- A DynamoDB table that contains data on the same account used with Amazon EMR.
- A customized version of Hive that includes connectivity to DynamoDB. The latest version of Hive provided by Amazon EMR is available by default when you launch an Amazon EMR cluster from the AWS Management Console . For more information about Amazon EMR AMIs and Hive versioning, go to [Specifying the Amazon EMR AMI Version](#) and to [Configuring Hive](#) in the *Amazon EMR Developer Guide*.
- Support for DynamoDB connectivity. This is included in the Amazon EMR AMI version 2.0.2 or later.
- (Optional) An Amazon S3 bucket. For instructions about how to create a bucket, see [Get Started With Amazon Simple Storage Service](#). This bucket is used as a destination when exporting DynamoDB data to Amazon S3 or as a location to store a Hive script.
- (Optional) A Secure Shell (SSH) client application to connect to the master node of the Amazon EMR cluster and run HiveQL queries against the DynamoDB data. SSH is used to run Hive interactively. You can also save Hive commands in a text file and have Amazon EMR run the Hive commands from the script. In this case an SSH client is not necessary, though the ability to SSH into the master node is useful even in non-interactive clusters, for debugging purposes.

An SSH client is available by default on most Linux, Unix, and Mac OS X installations. Windows users can install and use the [PuTTY](#) client, which has SSH support.

- (Optional) An Amazon EC2 key pair. This is only required for interactive clusters. The key pair provides the credentials the SSH client uses to connect to the master node. If you are running the Hive commands from a script in an Amazon S3 bucket, an EC2 key pair is optional.

Step 1: Create a Key Pair

To run Hive interactively to manage data in DynamoDB, you will need a key pair to connect to the Amazon EC2 instances launched by Amazon EMR (Amazon EMR). You will use this key pair to connect to the master node of the Amazon EMR job flow to run a HiveQL script (a language similar to SQL).

To generate a key pair

1. Sign in to the AWS Management Console and open the Amazon EC2 console at <https://console.amazonaws.cn/ec2/>.
2. In the upper right hand corner of the console, select a Region from the **Region** drop-down menu. This should be the same region that your DynamoDB database is in.
3. Click **Key Pairs** in the Navigation pane.
The console displays a list of key pairs associated with your account.
4. Click **Create Key Pair**.
5. Enter a name for the key pair, such as `mykeypair`, for the new key pair in the **Key Pair Name** field and click **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 instances that you launch with this key pair.

Important

If you lose the key pair, you cannot connect to your Amazon EC2 instances.

For more information about key pairs, see [Amazon Elastic Compute Cloud Key Pairs](#) in the *Amazon EC2 User Guide for Linux Instances*.

Step 2: Create a Cluster

In order for Hive to run on Amazon EMR, you must create a cluster with Hive enabled. This sets up the necessary applications and infrastructure for Hive to connect to DynamoDB. The following procedures explain how to create an interactive Hive cluster from the AWS Management Console and the CLI.

Topics

- [To start a cluster using the AWS Management Console \(p. 567\)](#)

To start a cluster using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon EMR console at <https://console.amazonaws.cn/elasticmapreduce/>.
2. Click **Create Cluster**.
3. In the **Create Cluster** page, in the **Cluster Configuration** section, verify the fields according to the following table.



Field	Action
Cluster name	Enter a descriptive name for your cluster. The name is optional, and does not need to be unique.
Termination protection	Choose Yes . Enabling termination protection ensures that the cluster does not shut down due to accident or error. For more information, see Protect a Cluster from Termination in the <i>Amazon EMR Developer Guide</i> . Typically, set this value to Yes only when developing an application (so you can debug errors that would have otherwise terminated the cluster) and to protect long-running clusters or clusters that contain data.
Logging	Choose Enabled . This determines whether Amazon EMR captures detailed log data to Amazon S3. For more information, see View Log Files in the <i>Amazon EMR Developer Guide</i> .
Log folder S3 location	Enter an Amazon S3 path to store your debug logs if you enabled logging in the previous field. When this value is set, Amazon EMR copies the log files from the EC2 instances in the cluster to Amazon S3. This prevents the log files from being lost when the cluster ends and the EC2 instances hosting the cluster are terminated. These logs are useful for troubleshooting purposes. For more information, see View Log Files in the <i>Amazon EMR Developer Guide</i> .

Field	Action
Debugging	<p>Choose Enabled.</p> <p>This option creates a debug log index in SimpleDB (additional charges apply) to enable detailed debugging in the Amazon EMR console. You can only set this when the cluster is created. For more information about Amazon SimpleDB, go to the Amazon SimpleDB product description page.</p>

4. In the **Software Configuration** section, verify the fields according to the following table.



Field	Action
Hadoop distribution	<p>Choose Amazon.</p> <p>This determines which distribution of Hadoop to run on your cluster. You can choose to run the Amazon distribution of Hadoop or one of several MapR distributions. For more information, see Using the MapR Distribution for Hadoop in the Amazon EMR Developer Guide.</p>
AMI version	<p>Choose the latest AMI version in the list.</p> <p>This determines the version of Hadoop and other applications such as Hive or Pig to run on your cluster. For more information, see Choose a Machine Image in the Amazon EMR Developer Guide.</p>
Applications to be installed - Hive	<p>A default Hive version should already be selected and displayed in the list. If it does not appear, choose it from the Additional applications list.</p> <p>For more information, see Analyze Data with Hive in the Amazon EMR Developer Guide.</p>
Applications to be installed - Pig	<p>A default Pig version should already be selected and displayed in the list. If it does not appear, choose it from the Additional applications list.</p> <p>For more information, see Process Data with Pig in the Amazon EMR Developer Guide.</p>

5. In the **Hardware Configuration** section, verify the fields according to the following table.

Note

The default maximum number of nodes *per AWS account* is twenty. For example, if you have two clusters running, the total number of nodes running for both clusters must be 20 or less. Exceeding this limit will result in cluster failures. If you need more than 20 nodes, you must submit a request to increase your Amazon EC2 instance limit. Ensure that your requested limit increase includes sufficient capacity for any temporary, unplanned increases in your needs. For more information, go to the [Request to Increase Amazon EC2 Instance Limit Form](#).



Field	Action
Network	<p>Choose Launch into EC2-Classic.</p> <p> Optionally, choose a VPC subnet identifier from the list to launch the cluster in an Amazon VPC. For more information, see Select a Amazon VPC Subnet for the Cluster (Optional) in the <i>Amazon EMR Developer Guide</i>.</p>
EC2 Availability Zone	<p>Choose No preference.</p> <p> Optionally, you can launch the cluster in a specific EC2 Availability Zone.</p> <p> For more information, see Regions and Availability Zones in the <i>Amazon EC2 User Guide</i>.</p>
Master - Amazon EC2 Instance Type	<p>For this tutorial, use the default EC2 instance type that is shown in this field.</p> <p>This specifies the EC2 instance types to use as master nodes. The master node assigns Hadoop tasks to core and task nodes, and monitors their status. There is always one master node in each cluster.</p> <p>For more information, see Instance Groups in the <i>Amazon EMR Developer Guide</i>.</p>
Request Spot Instances	<p>Leave this box unchecked.</p> <p>This specifies whether to run master nodes on Spot Instances. For more information, see Lower Costs with Spot Instances (Optional) in the <i>Amazon EMR Developer Guide</i>.</p>
Core - Amazon EC2 Instance Type	<p>For this tutorial, use the default EC2 instance type that is shown in this field.</p> <p>This specifies the EC2 instance types to use as core nodes. A core node is an EC2 instance that runs Hadoop map and reduce tasks and stores data using the Hadoop Distributed File System (HDFS). Core nodes are managed by the master node.</p> <p>For more information, see Instance Groups in the <i>Amazon EMR Developer Guide</i>.</p>
Count	Choose 2 .
Request Spot Instances	<p>Leave this box unchecked.</p> <p>This specifies whether to run core nodes on Spot Instances. For more information, see Lower Costs with Spot Instances (Optional) in the <i>Amazon EMR Developer Guide</i>.</p>
Task - Amazon EC2 Instance Type	<p>For this tutorial, use the default EC2 instance type that is shown in this field.</p> <p>This specifies the EC2 instance types to use as task nodes. A task node only processes Hadoop tasks and don't store data. You can add and remove them from a cluster to manage the EC2 instance capacity your cluster uses, increasing capacity to handle peak loads and decreasing it later. Task nodes only run a TaskTracker Hadoop daemon.</p> <p>For more information, see Instance Groups in the <i>Amazon EMR Developer Guide</i>.</p>
Count	Choose 0 .

Field	Action
Request Spot Instances	<p>Leave this box unchecked.</p> <p>This specifies whether to run task nodes on Spot Instances. For more information, see Lower Costs with Spot Instances (Optional) in the <i>Amazon EMR Developer Guide</i>.</p>

6. In the **Security and Access** section, complete the fields according to the following table.

Field	Action
EC2 key pair	<p>Choose the key pair that you created in Step 1: Create a Key Pair (p. 566).</p> <p>For more information, see Create SSH Credentials for the Master Node in the <i>Amazon EMR Developer Guide</i>.</p> <p>If you do not enter a value in this field, you will not be able to connect to the master node using SSH. For more information, see Connect to the Cluster in the <i>Amazon EMR Developer Guide</i>.</p>
IAM user access	<p>Choose No other IAM users.</p> <p> Optionally, choose All other IAM users to make the cluster visible and accessible to all IAM users on the AWS account. For more information, see Configure IAM User Permissions in the <i>Amazon EMR Developer Guide</i>.</p>
IAM role	<p>Choose Proceed without roles.</p> <p>This controls application access to the EC2 instances in the cluster.</p> <p>For more information, see Configure IAM Roles for Amazon EMR in the <i>Amazon EMR Developer Guide</i>.</p>

7. Review the **Bootstrap Actions** section, but note that you do not need to make any changes. There are no bootstrap actions necessary for this sample configuration.
- Optionally, you can use bootstrap actions, which are scripts that can install additional software and change the configuration of applications on the cluster before Hadoop starts. For more information, see [Create Bootstrap Actions to Install Additional Software \(Optional\)](#) in the *Amazon EMR Developer Guide*.
8. Review your configuration and if you are satisfied with the settings, click **Create Cluster**.
 9. When the cluster starts, you see the **Summary** pane.



Step 3: SSH into the Master Node

When the cluster's status is **WAITING**, the master node is ready for you to connect to it. With an active SSH session into the master node, you can execute command line operations.

To locate the public DNS name of the master node

- In the Amazon EMR console, select the cluster from the list of running clusters in the **WAITING** state.



The DNS name you use to connect to the instance is listed as **Master Public DNS Name**.

To connect to the master node using Mac OS X/Linux/UNIX

1. Go to the command prompt on your system. (On Mac OS X, use the **Terminal** program in */Applications/Utilities/Terminal*.)
2. Set the permissions on the `.pem` file for your Amazon EC2 key pair so that only the key owner has permissions to access the key. For example, if you saved the file as `mykeypair.pem` in the user's home directory, the command is:

```
chmod og-rwx ~/mykeypair.pem
```

If you do not perform this step, SSH returns an error saying that your private key file is unprotected and rejects the key. You only need to perform this step the first time you use the private key to connect.

3. To establish the connection to the master node, enter the following command line, which assumes the `.pem` file is in the user's home directory. Replace `master-public-dns-name` with the Master Public DNS Name of your cluster and replace `~/mykeypair.pem` with the location and filename of your `.pem` file.

```
ssh hadoop@master-public-dns-name -i ~/mykeypair.pem
```

A warning states that the authenticity of the host you are connecting to can't be verified.

4. Type `yes` to continue.

Note

If you are asked to log in, enter `hadoop`.

To install and configure PuTTY on Windows

1. Download PuTTYgen.exe and PuTTY.exe to your computer from <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>.
2. Launch PuTTYgen.
3. Click **Load**.
4. Select the PEM file you created earlier. Note that you may have to change the search parameters from file of type "PuTTY Private Key Files (*.ppk)" to "All Files (*.*)".
5. Click **Open**.
6. Click **OK** on the PuTTYgen notice telling you the key was successfully imported.
7. Click **Save private key** to save the key in the PPK format.
8. When PuTTYgen prompts you to save the key without a pass phrase, click **Yes**.
9. Enter a name for your PuTTY private key, such as `mykeypair.ppk`.
10. Click **Save**.
11. Close PuTTYgen.

To connect to the master node using PuTTY on Windows

1. Start PuTTY.

2. Select **Session** in the Category list. Enter `hadoop@DNS` in the Host Name field. The input looks similar to `hadoop@ec2-184-72-128-177.compute-1.amazonaws.com`.
3. In the Category list, expand **Connection**, expand **SSH**, and then select **Auth**. The **Options controlling the SSH authentication** pane appears.



4. For **Private key file for authentication**, click **Browse** and select the private key file you generated earlier. If you are following this guide, the file name is `mykeypair.ppk`.
 5. Click **Open**.
- A PuTTY Security Alert pops up.
6. Click **Yes** for the PuTTY Security Alert.

Note

If you are asked to log in, enter `hadoop`.

After you connect to the master node using either SSH or PuTTY, you should see a Hadoop command prompt and you are ready to start a Hive interactive session.

Step 4: Set Up a Hive Table to Run Hive Commands

Apache Hive is a data warehouse application you can use to query data contained in Amazon EMR clusters using a SQL-like language. Because we launched the cluster as a Hive application, Amazon EMR installs Hive on the EC2 instances it launches to process the cluster. For more information about Hive, go to <http://hive.apache.org/>.

If you've followed the previous instructions to set up a cluster and use SSH to connect to the master node, you are ready to use Hive interactively.

To run Hive commands interactively

1. At the command prompt for the current master node, type `hive`.

You should see a hive prompt: `hive>`

2. Enter a Hive command that maps a table in the Hive application to the data in DynamoDB. This table acts as a reference to the data stored in Amazon DynamoDB; the data is not stored locally in Hive and any queries using this table run against the live data in DynamoDB, consuming the table's read or write capacity every time a command is run. If you expect to run multiple Hive commands against the same dataset, consider exporting it first.

The following shows the syntax for mapping a Hive table to a DynamoDB table.

```
CREATE EXTERNAL TABLE hive_tablename (hive_column1_name column1_datatype,  
hive_column2_name column2_datatype...)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "dynamodb_tablename",  
"dynamodb.column.mapping" = "hive_column1_name:dynamodb_attribute1_name,hive_column2_name:dynamodb_attribute2_name..." );
```

When you create a table in Hive from DynamoDB, you must create it as an external table using the keyword `EXTERNAL`. The difference between external and internal tables is that the data in internal

tables is deleted when an internal table is dropped. This is not the desired behavior when connected to Amazon DynamoDB, and thus only external tables are supported.

For example, the following Hive command creates a table named *hivetable1* in Hive that references the DynamoDB table named *dynamodbt1*. The DynamoDB table *dynamodbt1* has a hash-and-range primary key schema. The hash key element is *name* (string type), the range key element is *year* (numeric type), and each item has an attribute value for *holidays* (string set type).

```
CREATE EXTERNAL TABLE hivetable1 (col1 string, col2 bigint, col3 array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbt1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");
```

Line 1 uses the HiveQL CREATE EXTERNAL TABLE statement. For *hivetable1*, you need to establish a column for each attribute name-value pair in the DynamoDB table, and provide the data type. These values *are not* case-sensitive, and you can give the columns any name (except reserved words).

Line 2 uses the STORED BY statement. The value of STORED BY is the name of the class that handles the connection between Hive and DynamoDB. It should be set to 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'.

Line 3 uses the TBLPROPERTIES statement to associate "hivetable1" with the correct table and schema in DynamoDB. Provide TBLPROPERTIES with values for the *dynamodb.table.name* parameter and *dynamodb.column.mapping* parameter. These values *are* case-sensitive.

Note

All DynamoDB attribute names for the table must have corresponding columns in the Hive table. Otherwise, the Hive table won't contain the name-value pair from DynamoDB. If you do not map the DynamoDB primary key attributes, Hive generates an error. If you do not map a non-primary key attribute, no error is generated, but you won't see the data in the Hive table. If the data types do not match, the value is null.

Then you can start running Hive operations on *hivetable1*. Queries run against *hivetable1* are internally run against the DynamoDB table *dynamodbt1* of your DynamoDB account, consuming read or write units with each execution.

When you run Hive queries against a DynamoDB table, you need to ensure that you have provisioned a sufficient amount of read capacity units.

For example, suppose that you have provisioned 100 units of read capacity for your DynamoDB table. This will let you perform 100 reads, or 409,600 bytes, per second. If that table contains 20GB of data (21,474,836,480 bytes), and your Hive query performs a full table scan, you can estimate how long the query will take to run:

$$21,474,836,480 / 409,600 = 52,429 \text{ seconds} = 14.56 \text{ hours}$$

The only way to decrease the time required would be to adjust the read capacity units on the source DynamoDB table. Adding more Amazon EMR nodes will not help.

In the Hive output, the completion percentage is updated when one or more mapper processes are finished. For a large DynamoDB table with a low provisioned read capacity setting, the completion percentage output might not be updated for a long time; in the case above, the job will appear to be 0% complete for several hours. For more detailed status on your job's progress, go to the Amazon EMR console; you will be able to view the individual mapper task status, and statistics for data reads.

You can also log on to Hadoop interface on the master node and see the Hadoop statistics. This will show you the individual map task status and some data read statistics. For more information, see the following topics:

- [Web Interfaces Hosted on the Master Node](#)
- [View the Hadoop Web Interfaces](#)

For more information about sample HiveQL statements to perform tasks such as exporting or importing data from DynamoDB and joining tables, see [Hive Command Examples for Exporting, Importing, and Querying Data in Amazon DynamoDB](#) in the *Amazon EMR Developer Guide*.

You can also create a file that contains a series of commands, launch a cluster, and reference that file to perform the operations. For more information, see [Interactive and Batch Modes](#) in the *Amazon EMR Developer Guide*.

To cancel a Hive request

When you execute a Hive query, the initial response from the server includes the command to cancel the request. To cancel the request at any time in the process, use the **Kill Command** from the server response.

1. Enter `Ctrl+C` to exit the command line client.
2. At the shell prompt, enter the **Kill Command** from the initial server response to your request.

Alternatively, you can run the following command from the command line of the master node to kill the Hadoop job, where `job-id` is the identifier of the Hadoop job and can be retrieved from the Hadoop user interface. For more information about the Hadoop user interface, see [How to Use the Hadoop User Interface](#) in the *Amazon EMR Developer Guide*.

```
hadoop job -kill job-id
```

Data Types for Hive and DynamoDB

The following table shows the available Hive data types and how they map to the corresponding DynamoDB data types.

Hive type	DynamoDB type
string	string (S)
bigint or double	number (N)
binary	binary (B)
array	number set (NS), string set (SS), or binary set (BS)

The bigint type in Hive is the same as the Java long type, and the Hive double type is the same as the Java double type in terms of precision. This means that if you have numeric data stored in DynamoDB that has precision higher than is available in the Hive datatypes, using Hive to export, import, or reference the DynamoDB data could lead to a loss in precision or a failure of the Hive query.

Exports of the binary type from DynamoDB to Amazon Simple Storage Service (Amazon S3) or HDFS are stored as a Base64-encoded string. If you are importing data from Amazon S3 or HDFS into the DynamoDB binary type, it should be encoded as a Base64 string.

Hive Options

You can set the following Hive options to manage the transfer of data out of Amazon DynamoDB. These options only persist for the current Hive session. If you close the Hive command prompt and reopen it later on the cluster, these settings will have returned to the default values.

Hive Options	Description
<i>dynamodb.read.percent</i>	<p>Set the rate of read operations to keep your DynamoDB provisioned throughput rate in the allocated range for your table. The value is between 0.1 and 1.5, inclusively.</p> <p>The value of 0.5 is the default read rate, which means that Hive will attempt to consume half of the read provisioned throughout resources in the table. Increasing this value above 0.5 increases the read request rate. Decreasing it below 0.5 decreases the read request rate. This read rate is approximate. The actual read rate will depend on factors such as whether there is a uniform distribution of keys in DynamoDB.</p> <p>If you find your provisioned throughput is frequently exceeded by the Hive operation, or if live read traffic is being throttled too much, then reduce this value below 0.5. If you have enough capacity and want a faster Hive operation, set this value above 0.5. You can also oversubscribe by setting it up to 1.5 if you believe there are unused input/output operations available.</p>
<i>dynamodb.write.percent</i>	<p>Set the rate of write operations to keep your DynamoDB provisioned throughput rate in the allocated range for your table. The value is between 0.1 and 1.5, inclusively.</p> <p>The value of 0.5 is the default write rate, which means that Hive will attempt to consume half of the write provisioned throughout resources in the table. Increasing this value above 0.5 increases the write request rate. Decreasing it below 0.5 decreases the write request rate. This write rate is approximate. The actual write rate will depend on factors such as whether there is a uniform distribution of keys in DynamoDB.</p> <p>If you find your provisioned throughput is frequently exceeded by the Hive operation, or if live write traffic is being throttled too much, then reduce this value below 0.5. If you have enough capacity and want a faster Hive operation, set this value above 0.5. You can also oversubscribe by setting it up to 1.5 if you believe there are unused input/output operations available or this is the initial data upload to the table and there is no live traffic yet.</p>
<i>dynamodb.endpoint</i>	Specify the endpoint in case you have tables in different regions. For more information about the available DynamoDB endpoints, see Regions and Endpoints .

Hive Options	Description
<code>dynamodb.max.map.tasks</code>	Specify the maximum number of map tasks when reading data from DynamoDB. This value must be equal to or greater than 1.
<code>dynamodb.retry.duration</code>	Specify the number of minutes to use as the timeout duration for retrying Hive commands. This value must be an integer equal to or greater than 0. The default timeout duration is two minutes.

These options are set using the `SET` command as shown in the following example.

```
SET dynamodb.throughput.read.percent=1.0;

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

If you are using the AWS SDK for Java, you can use the `-e` option of Hive to pass in the command directly, as shown in the last line of the following example.

```
steps.add(new StepConfig()
    .withName("Run Hive Script")
    .withHadoopJarStep(new HadoopJarStepConfig()
        .withJar("s3://us-west-2.elasticmapreduce/libs/script-runner/script-runner.jar")
        .withArgs("s3://us-west-2.elasticmapreduce/libs/hive/hive-script",
            "--base-path", "s3://us-west-2.elasticmapreduce/libs/hive/", "--run-hive-script",
            "--args", "-e", "SET dynamodb.throughput.read.percent=1.0;")));
```

Hive Command Examples for Exporting, Importing, and Querying Data in DynamoDB

The following examples use Hive commands to perform operations such as exporting data to Amazon S3 or HDFS, importing data to DynamoDB, joining tables, querying tables, and more.

Operations on a Hive table reference data stored in DynamoDB. Hive commands are subject to the DynamoDB table's provisioned throughput settings, and the data retrieved includes the data written to the DynamoDB table at the time the Hive operation request is processed by DynamoDB. If the data retrieval process takes a long time, some data returned by the Hive command may have been updated in DynamoDB since the Hive command began.

Hive commands `DROP TABLE` and `CREATE TABLE` only act on the local tables in Hive and do not create or drop tables in DynamoDB. If your Hive query references a table in DynamoDB, that table must already exist before you run the query. For more information on creating and deleting tables in DynamoDB, go to [Working with Tables in DynamoDB](#).

Note

When you map a Hive table to a location in Amazon S3, do not map it to the root path of the bucket, `s3://mybucket`, as this may cause errors when Hive writes the data to Amazon S3. Instead map the table to a subpath of the bucket, `s3://mybucket/mypath`.

Exporting Data from DynamoDB

You can use Hive to export data from DynamoDB.

To export a DynamoDB table to an Amazon S3 bucket

- Create a Hive table that references data stored in DynamoDB. Then you can call the INSERT OVERWRITE command to write the data to an external directory. In the following example, `s3://bucketname/path/subpath/` is a valid path in Amazon S3. Adjust the columns and datatypes in the CREATE command to match the values in your DynamoDB. You can use this to create an archive of your DynamoDB data in Amazon S3.

```
CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtale1",
 "dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays" );

INSERT OVERWRITE DIRECTORY 's3://bucketname/path/subpath/' SELECT *
FROM hiveTableName;
```

To export a DynamoDB table to an Amazon S3 bucket using formatting

- Create an external table that references a location in Amazon S3. This is shown below as `s3_export`. During the CREATE call, specify row formatting for the table. Then, when you use INSERT OVERWRITE to export data from DynamoDB to `s3_export`, the data is written out in the specified format. In the following example, the data is written out as comma-separated values (CSV).

```
CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtale1",
 "dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays" );

CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col array<string>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

To export a DynamoDB table to an Amazon S3 bucket without specifying a column mapping

- Create a Hive table that references data stored in DynamoDB. This is similar to the preceding example, except that you are not specifying a column mapping. The table must have exactly one column of type `map<string, string>`. If you then create an `EXTERNAL` table in Amazon S3 you can call the `INSERT OVERWRITE` command to write the data from DynamoDB to Amazon S3. You can use this to create an archive of your DynamoDB data in Amazon S3. Because there is no column mapping, you cannot query tables that are exported this way. Exporting data without specifying a column mapping is available in Hive 0.8.1.5 or later, which is supported on Amazon EMR AMI 2.2.x and later.

```
CREATE EXTERNAL TABLE hiveTableName (item map<string,string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtale1");

CREATE EXTERNAL TABLE s3TableName (item map<string, string>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3TableName SELECT *
FROM hiveTableName;
```

To export a DynamoDB table to an Amazon S3 bucket using data compression

- Hive provides several compression codecs you can set during your Hive session. Doing so causes the exported data to be compressed in the specified format. The following example compresses the exported files 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 hiveTableName (col1 string, col2 bigint, col3 array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtale1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays" );

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 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

To export a DynamoDB table to HDFS

- Use the following Hive command, where `hdfs://directoryName` is a valid HDFS path and `hiveTableName` is a table in Hive that references DynamoDB. This export operation is faster than exporting a DynamoDB table to Amazon S3 because Hive 0.7.1.1 uses HDFS as an intermediate step when exporting data to Amazon S3. The following example also shows how to set `dynamodb.throughput.read.percent` to 1.0 in order to increase the read request rate.

```
CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtble1",
               "dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays" );
SET dynamodb.throughput.read.percent=1.0;

INSERT OVERWRITE DIRECTORY 'hdfs://directoryName' SELECT * FROM hiveTableName;
```

You can also export data to HDFS using formatting and compression as shown above for the export to Amazon S3. To do so, simply replace the Amazon S3 directory in the examples above with an HDFS directory.

To read non-printable UTF-8 character data in Hive

- You can read and write non-printable UTF-8 character data with Hive by using the `STORED AS SEQUENCEFILE` clause when you create the table. A SequenceFile is 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 hiveTableName (col1 string, col2 bigint, col3 array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtble1",
               "dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays" );

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;
```

Importing Data to DynamoDB

When you write data to DynamoDB using Hive you should ensure that the number of write capacity units is greater than the number of mappers in the cluster. For example, clusters that run on m1.xlarge EC2 instances produce 8 mappers per instance. In the case of a cluster that has 10 instances, that would mean a total of 80 mappers. If your write capacity units are not greater than the number of mappers in the cluster, the Hive write operation may consume all of the write throughput, or attempt to consume more throughput than is provisioned. For more information about the number of mappers produced by each EC2 instance type, go to [Hadoop Configuration Reference](#) in the *Amazon EMR Developer Guide*. There, you will find a "Task Configuration" section for each of the supported configurations.

The number of mappers in Hadoop are controlled by the input splits. If there are too few splits, your write command might not be able to consume all the write throughput available.

If an item with the same key exists in the target DynamoDB table, it will be overwritten. If no item with the key exists in the target DynamoDB table, the item is inserted.

To import a table from Amazon S3 to DynamoDB

- You can use Amazon EMR (Amazon EMR) and Hive to write data from Amazon S3 to DynamoDB.

```
CREATE EXTERNAL TABLE s3_import(a_col string, b_col bigint, c_col ar  
ray<string>)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
LOCATION 's3://bucketname/path/subpath/';  
  
CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 ar  
ray<string>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtabel1",  
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");  
  
INSERT OVERWRITE TABLE 'hiveTableName' SELECT * FROM s3_import;
```

To import a table from an Amazon S3 bucket to DynamoDB without specifying a column mapping

- Create an EXTERNAL table that references data stored in Amazon S3 that was previously exported from DynamoDB. Before importing, ensure that the table exists in DynamoDB and that it has the same key schema as the previously exported DynamoDB table. In addition, the table must have exactly one column of type map<string, string>. If you then create a Hive table that is linked to DynamoDB, you can call the INSERT OVERWRITE command to write the data from Amazon S3 to DynamoDB. Because there is no column mapping, you cannot query tables that are imported this way. Importing data without specifying a column mapping is available in Hive 0.8.1.5 or later, which is supported on Amazon EMR AMI 2.2.3 and later.

```
CREATE EXTERNAL TABLE s3TableName (item map<string, string>)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'  
LOCATION 's3://bucketname/path/subpath/';  
  
CREATE EXTERNAL TABLE hiveTableName (item map<string,string>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
```

```
TBLPROPERTIES ( "dynamodb.table.name" = "dynamodbtale1" );  
  
INSERT OVERWRITE TABLE hiveTableName SELECT *  
FROM s3TableName;
```

To import a table from HDFS to DynamoDB

- You can use Amazon EMR and Hive to write data from HDFS to DynamoDB.

```
CREATE EXTERNAL TABLE hdfs_import(a_col string, b_col bigint, c_col array<string>)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
LOCATION 'hdfs://directoryName';  
  
CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 array<string>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtale1",  
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays" );  
  
INSERT OVERWRITE TABLE 'hiveTableName' SELECT * FROM hdfs_import;
```

Querying Data in DynamoDB

The following examples show the various ways you can use Amazon EMR to query data stored in DynamoDB.

To find the largest value for a mapped column (`max`)

- Use Hive commands like the following. In the first command, the CREATE statement creates a Hive table that references data stored in DynamoDB. The SELECT statement then uses that table to query data stored in DynamoDB. The following example finds the largest order placed by a given customer.

```
CREATE EXTERNAL TABLE hive_purchases(customerId bigint, total_cost double,  
 items_purchased array<String>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "Purchases",  
"dynamodb.column.mapping" = "customerId:CustomerId,total_cost:Cost,items_purchased:Items" );  
  
SELECT max(total_cost) from hive_purchases where customerId = 717;
```

To aggregate data using the GROUP BY clause

- 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. The following example returns a list of the largest orders from customers who have placed more than three orders.

```
CREATE EXTERNAL TABLE hive_purchases(customerId bigint, total_cost double,  
    items_purchased array<String>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "Purchases",  
    "dynamodb.column.mapping" = "customerId:CustomerId,total_cost:Cost,items_purchased:Items" );  
  
SELECT customerId, max(total_cost) from hive_purchases GROUP BY customerId  
HAVING count(*) > 3;
```

To join two DynamoDB tables

- The following example maps two Hive tables to data stored in DynamoDB. It then calls a join across those two tables. The join is computed on the cluster and returned. The join does not take place in DynamoDB. This example returns a list of customers and their purchases for customers that have placed more than two orders.

```
CREATE EXTERNAL TABLE hive_purchases(customerId bigint, total_cost double,  
    items_purchased array<String>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "Purchases",  
    "dynamodb.column.mapping" = "customerId:CustomerId,total_cost:Cost,items_purchased:Items" );  
  
CREATE EXTERNAL TABLE hive_customers(customerId bigint, customerName string,  
    customerAddress array<String>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "Customers",  
    "dynamodb.column.mapping" = "customerId:CustomerId,customerName:Name,customerAddress:Address" );  
  
Select c.customerId, c.customerName, count(*) as count from hive_customers c  
JOIN hive_purchases p ON c.customerId=p.customerId  
GROUP BY c.customerId, c.customerName HAVING count > 2;
```

To join two tables from different sources

- In the following example, Customer_S3 is a Hive table that loads a CSV file stored in Amazon S3 and *hive_purchases* is a table that references data in DynamoDB. The following example joins together customer data stored as a CSV file in Amazon S3 with order data stored in DynamoDB to return a set of data that represents orders placed by customers who have "Miller" in their name.

```
CREATE EXTERNAL TABLE hive_purchases(customerId bigint, total_cost double,  
    items_purchased array<String>)  
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'  
TBLPROPERTIES ("dynamodb.table.name" = "Purchases",  
    "dynamodb.column.mapping" = "customerId:CustomerId,total_cost:Cost,items_purchased:Items");  
  
CREATE EXTERNAL TABLE Customer_S3(customerId bigint, customerName string,  
    customerAddress array<String>)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
LOCATION 's3://bucketname/path/subpath/';  
  
Select c.customerId, c.customerName, c.customerAddress from  
Customer_S3 c  
JOIN hive_purchases p  
ON c.customerid=p.customerid  
where c.customerName like '%Miller%';
```

Note

In the preceding examples, the CREATE TABLE statements were included in each example for clarity and completeness. When running multiple queries or export operations against a given Hive table, you only need to create the table one time, at the beginning of the Hive session.

Optimizing Performance for Amazon EMR Operations in DynamoDB

Amazon EMR operations on a DynamoDB table count as read operations, and are subject to the table's provisioned throughput settings. Amazon EMR implements its own logic to try to balance the load on your DynamoDB table to minimize the possibility of exceeding your provisioned throughput. At the end of each Hive query, Amazon EMR returns information about the cluster used to process the query, including how many times your provisioned throughput was exceeded. You can use this information, as well as CloudWatch metrics about your DynamoDB throughput, to better manage the load on your DynamoDB table in subsequent requests.

The following factors influence Hive query performance when working with DynamoDB tables.

Provisioned Read Capacity Units

When you run Hive queries against a DynamoDB table, you need to ensure that you have provisioned a sufficient amount of read capacity units.

For example, suppose that you have provisioned 100 units of Read Capacity for your DynamoDB table. This will let you perform 100 reads, or 409,600 bytes, per second. If that table contains 20GB of data (21,474,836,480 bytes), and your Hive query performs a full table scan, you can estimate how long the query will take to run:

$$21,474,836,480 / 409,600 = 52,429 \text{ seconds} = 14.56 \text{ hours}$$

The only way to decrease the time required would be to adjust the read capacity units on the source DynamoDB table. Adding more nodes to the Amazon EMR cluster will not help.

In the Hive output, the completion percentage is updated when one or more mapper processes are finished. For a large DynamoDB table with a low provisioned Read Capacity setting, the completion percentage output might not be updated for a long time; in the case above, the job will appear to be 0% complete for several hours. For more detailed status on your job's progress, go to the Amazon EMR console; you will be able to view the individual mapper task status, and statistics for data reads.

You can also log on to Hadoop interface on the master node and see the Hadoop statistics. This will show you the individual map task status and some data read statistics. For more information, see the following topics:

- [Web Interfaces Hosted on the Master Node](#)
- [View the Hadoop Web Interfaces](#)

Read Percent Setting

By default, Amazon EMR manages the request load against your DynamoDB table according to your current provisioned throughput. However, when Amazon EMR returns information about your job that includes a high number of provisioned throughput exceeded responses, you can adjust the default read rate using the `dynamodb.read.percent` parameter when you set up the Hive table. For more information about setting the read percent parameter, see [Hive Options](#) in the *Amazon EMR Developer Guide*.

Write Percent Setting

By default, Amazon EMR manages the request load against your DynamoDB table according to your current provisioned throughput. However, when Amazon EMR returns information about your job that includes a high number of provisioned throughput exceeded responses, you can adjust the default write rate using the `dynamodb.write.percent` parameter when you set up the Hive table. For more information about setting the write percent parameter, see [Hive Options](#) in the *Amazon EMR Developer Guide*.

Retry Duration Setting

By default, Amazon EMR re-runs a Hive query if it has not returned a result within two minutes, the default retry interval. You can adjust this interval by setting the `dynamodb.retry.duration` parameter when you run a Hive query. For more information about setting the write percent parameter, see [Hive Options](#) in the *Amazon EMR Developer Guide*.

Number of Map Tasks

The mapper daemons that Hadoop launches to process your requests to export and query data stored in DynamoDB are capped at a maximum read rate of 1 MiB per second to limit the read capacity used. If you have additional provisioned throughput available on DynamoDB, you can improve the performance of Hive export and query operations by increasing the number of mapper daemons. To do this, you can either increase the number of EC2 instances in your cluster or increase the number of mapper daemons running on each EC2 instance.

You can increase the number of EC2 instances in a cluster by stopping the current cluster and re-launching it with a larger number of EC2 instances. You specify the number of EC2 instances in the **Configure EC2 Instances** dialog box if you're launching the cluster from the Amazon EMR console, or with the `--num-instances` option if you're launching the cluster from the CLI.

The number of map tasks run on an instance depends on the EC2 instance type. For more information about the supported EC2 instance types and the number of mappers each one provides, go to [Hadoop Configuration Reference](#) in the *Amazon EMR Developer Guide*. There, you will find a "Task Configuration" section for each of the supported configurations.

Another way to increase the number of mapper daemons is to change the `mapred.tasktracker.map.tasks.maximum` configuration parameter of Hadoop to a higher value. This has the advantage of giving you more mappers without increasing either the number or the size of EC2 instances, which saves you money. A disadvantage is that setting this value too high can cause the EC2 instances in your cluster to run out of memory. To set `mapred.tasktracker.map.tasks.maximum`, launch the cluster and specify the Configure Hadoop bootstrap action, passing in a value for `mapred.tasktracker.map.tasks.maximum` as one of the arguments of the bootstrap action. This is shown in the following example.

```
--bootstrap-action s3n://elasticmapreduce/bootstrap-actions/configure-hadoop \
--args -m, mapred.tasktracker.map.tasks.maximum=10
```

For more information about bootstrap actions, see [Using Custom Bootstrap Actions](#) in the *Amazon EMR Developer Guide*.

Parallel Data Requests

Multiple data requests, either from more than one user or more than one application to a single table may 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.

Avoid Exceeding Throughput

When running Hive queries against DynamoDB, take care not to exceed your provisioned throughput, because this will deplete capacity needed for your application's calls to `DynamoDB::Get`. To ensure that this is not occurring, you should regularly monitor the read volume and throttling on application calls to `DynamoDB::Get` by checking logs and monitoring metrics in Amazon CloudWatch.

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 a.m. PST, when the majority of users are asleep, and not updating records in your DynamoDB database.

Time-Based Tables

If the data is organized as a series of time-based DynamoDB tables, such as one table per day, you can export the data when the table becomes no longer active. You can use this technique to back up data to Amazon S3 on an ongoing fashion.

Archived Data

If you plan to run many Hive queries against the data stored in DynamoDB and your application can tolerate archived data, you may want to export the data to HDFS or Amazon S3 and run the Hive queries

against a copy of the data instead of DynamoDB. This conserves your read operations and provisioned throughput.

Viewing Hadoop Logs

If you run into an error, you can investigate what went wrong by viewing the Hadoop logs and user interface. For more information, see [How to Monitor Hadoop on a Master Node](#) and [How to Use the Hadoop User Interface](#) in the *Amazon EMR Developer Guide*.

Walkthrough: Using DynamoDB and Amazon Elastic MapReduce

Topics

- [Video \(p. 586\)](#)
- [Step-by-Step Instructions \(p. 586\)](#)

The integration of Amazon Elastic MapReduce (Amazon EMR) with DynamoDB enables several scenarios. For example, using a Hive cluster launched within Amazon EMR, you can export data to Amazon Simple Storage Service (Amazon S3) or upload it to a Native Hive Table. In this walkthrough, you'll learn how to set up a Hive cluster, export DynamoDB data to Amazon S3, upload data to a native Hive table, and execute complex queries for business intelligence reporting or data mining. You can run queries against the data without using a lot of DynamoDB capacity units or interfering with your running application.

Video

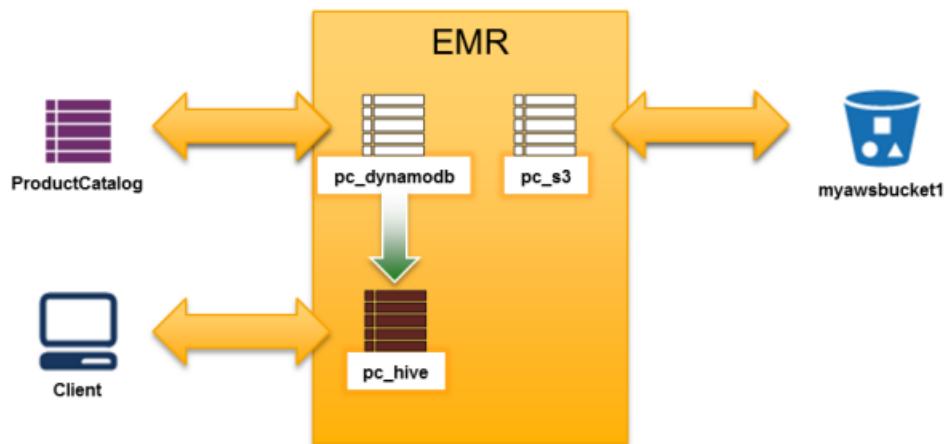
[Video: Using Amazon Elastic MapReduce \(Amazon EMR\) to Export and Analyze DynamoDB Data](#)

Step-by-Step Instructions

Topics

- [Setting Up the Environment \(p. 587\)](#)
- [Exporting Data to Amazon S3 \(p. 588\)](#)
- [Exporting DynamoDB Data to a Native Hive Table and Executing Queries \(p. 591\)](#)
- [Final Cleanup \(p. 594\)](#)

When you have completed this walkthrough, you will have a DynamoDB table with sample data, an Amazon S3 bucket with exported data, an Amazon EMR job flow, two Apache Hive external tables, and one native Hive table.



Setting Up the Environment

Upon completing this step, you will have the `ProductCatalog` table in DynamoDB, a bucket in Amazon S3, and an Amazon EMR job flow set up.

To set up the walkthrough environment

1. Create the `ProductCatalog` table in DynamoDB and upload sample data.

The `ProductCatalog` table used in the video is one of the tables you create by following the steps in the [Getting Started with DynamoDB](#) section to create and populate the DynamoDB tables. You need only follow the Prerequisites and Steps 1 through 3 to create and populate the data.

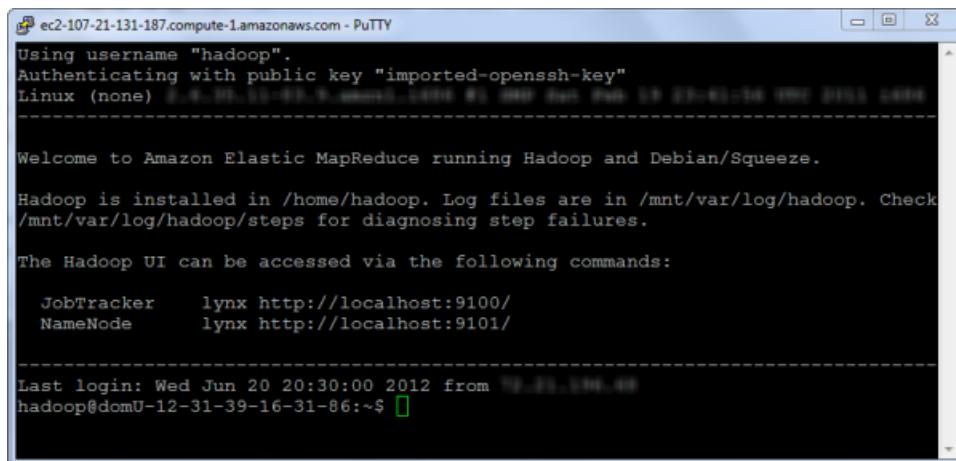
2. Create a bucket in Amazon S3.

For step-by-step instructions, go to the [Creating an Amazon S3 Bucket](#) topic in the *Amazon Simple Storage Service Getting Started Guide*.

3. Set up an Amazon EMR job flow.

This Amazon EMR job flow handles the queries between DynamoDB, Apache Hive, and Amazon S3. Follow the Prerequisites and Steps 1 through 3 in the [Exporting, Importing, Querying, and Joining Tables in DynamoDB Using Amazon EMR](#) section of the DynamoDB documentation to set up your job flow for these operations.

When you have completed the environment setup, your SSH session will look like this screen shot. You can now proceed with the rest of the walkthrough.



```
ec2-107-21-131-187.compute-1.amazonaws.com - PuTTY
Using username "hadoop".
Authenticating with public key "imported-openssh-key"
Linux (none)

Welcome to Amazon Elastic MapReduce running Hadoop and Debian/Squeeze.

Hadoop is installed in /home/hadoop. Log files are in /mnt/var/log/hadoop. Check /mnt/var/log/hadoop/steps for diagnosing step failures.

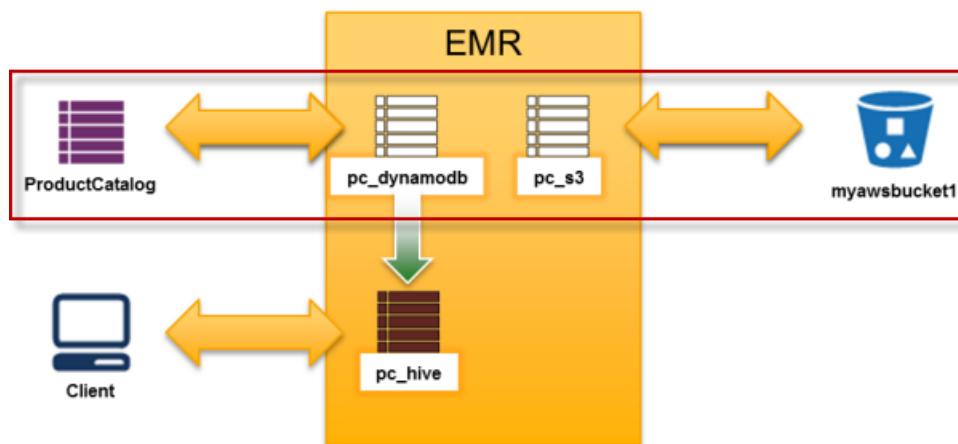
The Hadoop UI can be accessed via the following commands:

JobTracker lynx http://localhost:9100/
NameNode lynx http://localhost:9101/

Last login: Wed Jun 20 20:30:00 2012 from 107.21.131.187
hadoop@domU-12-31-39-16-31-86:~$
```

Exporting Data to Amazon S3

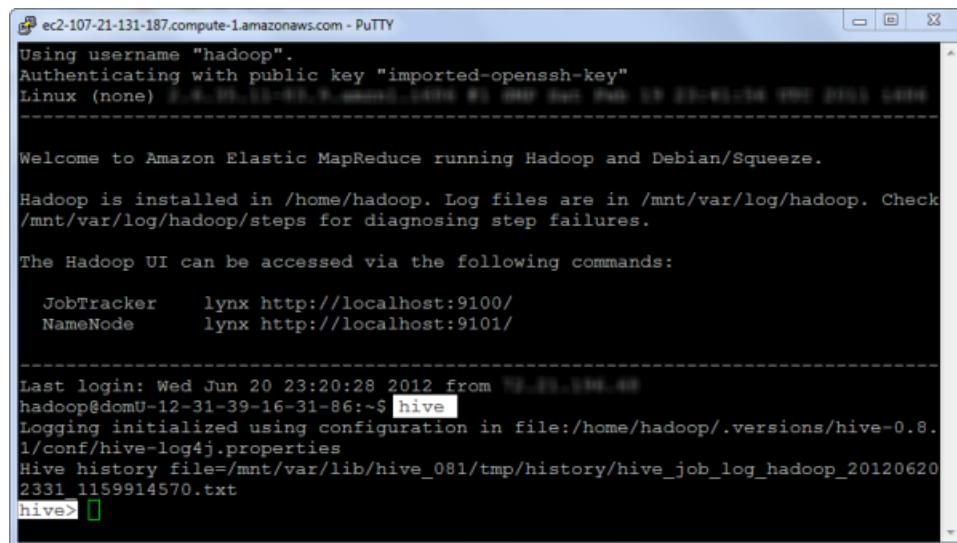
Now you are ready to export data from DynamoDB to an Amazon S3 bucket. As shown in the video, you need to create two external Hive tables. The first external table, `pc_dynamodb` (where "pc" is short for product catalog), maps to the DynamoDB table `ProductCatalog`. The external Hive table, `pc_s3`, maps to a folder (`catalog`) in an Amazon S3 bucket (`myawsbucket1`).



To create the first external table

1. Type `hive` to start a Hive command prompt.

When Hive is ready, you see a `hive>` prompt.



```

ec2-107-21-131-187.compute-1.amazonaws.com - PuTTY
Using username "hadoop".
Authenticating with public key "imported-openssh-key"
Linux (none)

-----
Welcome to Amazon Elastic MapReduce running Hadoop and Debian/Squeeze.

Hadoop is installed in /home/hadoop. Log files are in /mnt/var/log/hadoop. Check /mnt/var/log/hadoop/steps for diagnosing step failures.

The Hadoop UI can be accessed via the following commands:

JobTracker lynx http://localhost:9100/
NameNode lynx http://localhost:9101/

-----
Last login: Wed Jun 20 23:20:28 2012 from hadoop@domU-12-31-39-16-31-86:~$ hive
Logging initialized using configuration in file:/home/hadoop/.versions/hive-0.8.1/conf/hive-log4j.properties
Hive history file=/mnt/var/lib/hive_081/tmp/history/hive_job_log_hadoop_201206202331_1159914570.txt
hive>

```

2. Create the external Hive table pc_dynamodb that maps to the ProductCatalog table in DynamoDB.

Copy and paste the following code into your Hive session.

```

CREATE EXTERNAL TABLE pc_dynamodb (
    id bigint
    ,title string
    ,isbn string
    ,authors array<string>
    ,price bigint
    ,dimensions string
    ,pagecount bigint
    ,inpublication bigint
    ,productcategory string
    ,description string
    ,bicycletype string
    ,brand string
    ,gender string
    ,color array<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name"="ProductCatalog", "dynamodb.column.mapping" = "id:Id,title:Title
,isbn:ISBN,authors:Authors,price:Price,dimensions:Dimensions,page
count:PageCount
,inpublication:InPublication,productcategory:ProductCategory,descrip
tion:Description
,bicycletype:BicycleType,brand:Brand,gender:Gender,color:Color");

```

When it has completed creating the table, Hive responds `OK`. You can verify the existence of the table by typing `show tables;` at the command line.

3. Create a second external Hive table that maps to a folder in the specified Amazon S3 bucket.

Copy and paste the following code into your Hive session. The external Hive table name is `pc_s3`, and it maps to a folder in the Amazon S3 bucket `myawsbucket1/catalog`. Note that we specify `ROW FORMAT` to request comma-separated values in the resulting Amazon S3 object. Before copying and pasting this code, adjust the name of the Amazon S3 bucket in the last line to the name of the bucket and folder you created in the [To set up the walkthrough environment \(p. 587\)](#) procedure.

```
CREATE EXTERNAL TABLE pc_s3 (
    id bigint
    ,title string
    ,isbn string
    ,authors array<string>
    ,price bigint
    ,dimensions string
    ,pagecount bigint
    ,inpublication bigint
    ,productcategory string
    ,description string
    ,bicycletype string
    ,brand string
    ,gender string
    ,color array<string>
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://myawsbucket1/catalog/';
```

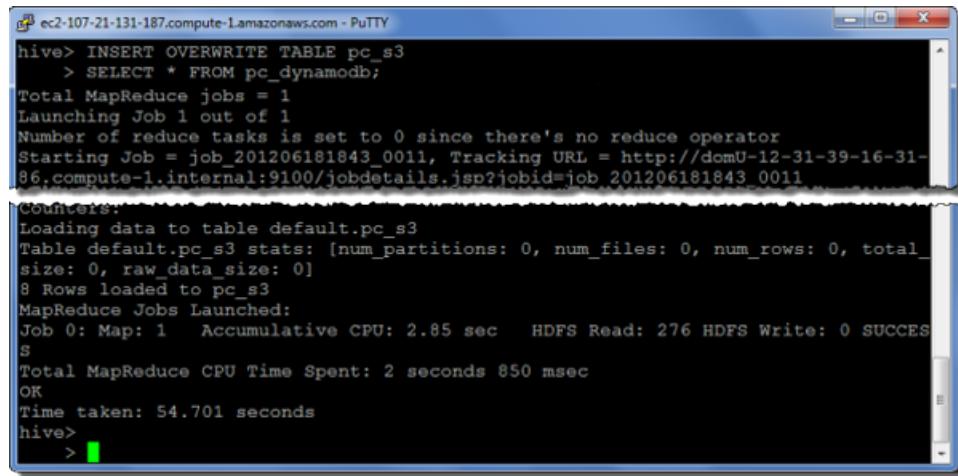
When it has completed creating the table, Hive responds **OK**. Now you can export the DynamoDB table data to an Amazon S3 bucket.

To export the data to your Amazon S3 bucket

1. Use an **INSERT** statement as follows:

```
INSERT OVERWRITE TABLE pc_s3
SELECT * FROM pc_dynamodb;
```

This statement selects data from the Dynamo DB table and inserts it into a folder in the specified Amazon S3 bucket through the mapped tables.



The screenshot shows a PuTTY terminal window titled "ec2-107-21-131-187.compute-1.amazonaws.com - PuTTY". The command entered is:

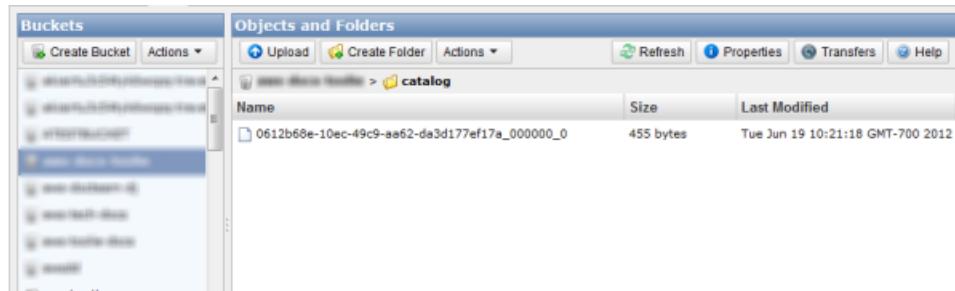
```
hive> INSERT OVERWRITE TABLE pc_s3
      > SELECT * FROM pc_dynamodb;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_201206181843_0011, Tracking URL = http://domU-12-31-39-16-31-86.compute-1.internal:9100/jobdetails.jsp?jobid=job_201206181843_0011
```

Counters:

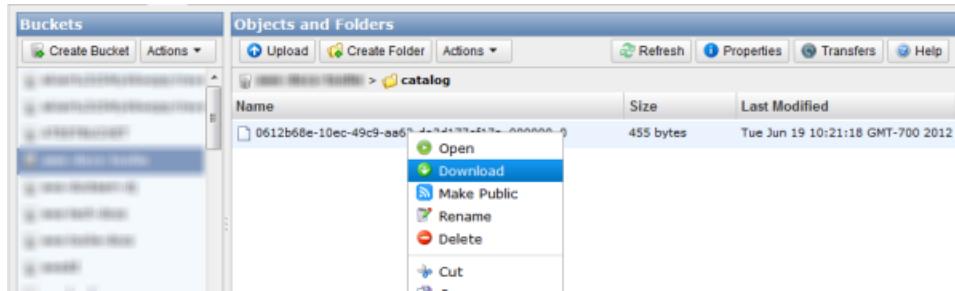
```
Loading data to table default.pc_s3
Table default.pc_s3 stats: [num_partitions: 0, num_files: 0, num_rows: 0, total_size: 0, raw_data_size: 0]
8 Rows loaded to pc_s3
MapReduce Jobs Launched:
Job 0: Map: 1   Accumulative CPU: 2.85 sec   HDFS Read: 276 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 2 seconds 850 msec
OK
Time taken: 54.701 seconds
hive> >
```

When the **INSERT** completes, the data is stored in Amazon S3.

2. Verify the resulting object by returning to the AWS Management Console at <https://console.aws.amazon.com/s3/home> and locating the bucket you created.



- To download the file, right-click the file, and then click **Download** in the context menu.



Click the **Download** button to save the file to your local drive.



- Open the file in a text editor to see the data. Depending on your platform, you may need to add a ".txt" file name extension to open the file.

```
0612b68e-10ec-49c9-aa62-da3d17...
205,20-Bike-205,\N,\N,500,\N,\N,\N,Bicycle,205 Description,Hybrid,Brand-Company C,B,Black;Red
203,19-Bike-203,\N,\N,300,\N,\N,\N,Bicycle,203 Description,Road,Brand-Company B,W,Black;Green;Red
201,18-Bike-201,\N,\N,100,\N,\N,\N,Bicycle,201 Description,Road,Mountain A,M,Black;Red
204,18-Bike-204,\N,\N,400,\N,\N,\N,Bicycle,204 Description,Mountain,Brand-Company B,W,Red
101,Book 101 Title,111-111111111,Author1,2,8.5 x 11.0 x 0.5,\N,1,Book,\N,\N,\N,\N,\N
```

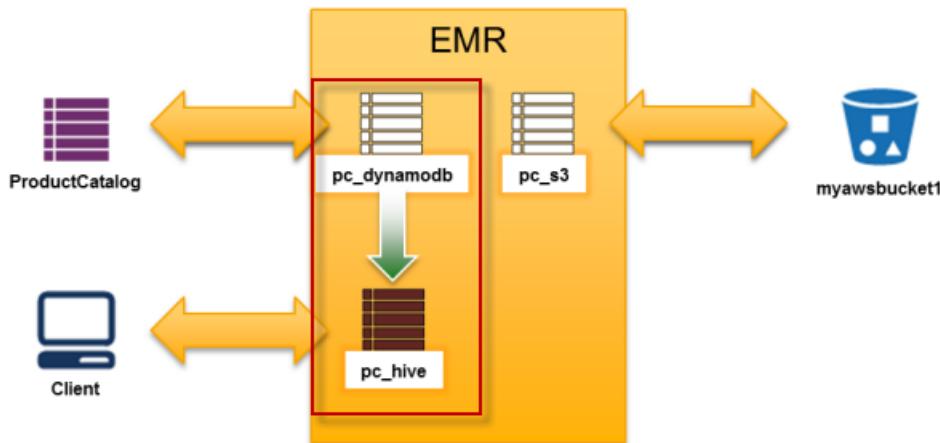
Note

You can import data from Amazon S3 to a DynamoDB table, too. This is a useful way to import existing data into a new DynamoDB table or perform periodic bulk uploads of data from another application. With Amazon EMR's support for scripting, you can save your scripts and run them according to a schedule.

Exporting DynamoDB Data to a Native Hive Table and Executing Queries

Next, you load data from DynamoDB into a native Hive table and execute a sample query. As shown in the video, this query is executed on the data stored natively. Uploading data consumes some provisioned

throughput in DynamoDB, but queries on the data stored natively in an Amazon EMR cluster do not consume DynamoDB provisioned throughput.



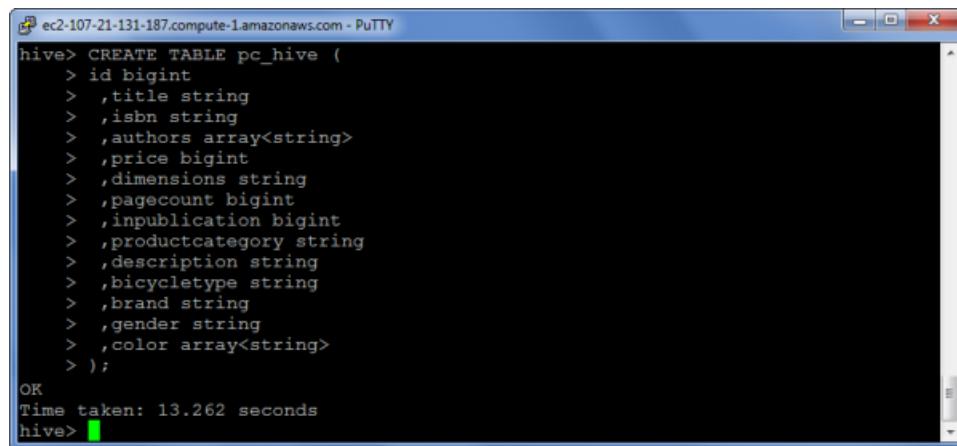
You already have an external Hive table (`pc_dynamodb`) mapped to our DynamoDB ProductCatalog table as shown in the preceding figure. Now you need only create a native Hive table where you will load the data for your query.

To create a native Hive table

1. Copy and paste the following code into your Hive session.

```
CREATE TABLE pc_hive (
    id bigint
    ,title string
    ,isbn string
    ,authors array<string>
    ,price bigint
    ,dimensions string
    ,pagecount bigint
    ,inpublication bigint
    ,productcategory string
    ,description string
    ,bicycletype string
    ,brand string
    ,gender string
    ,color array<string>
) ;
```

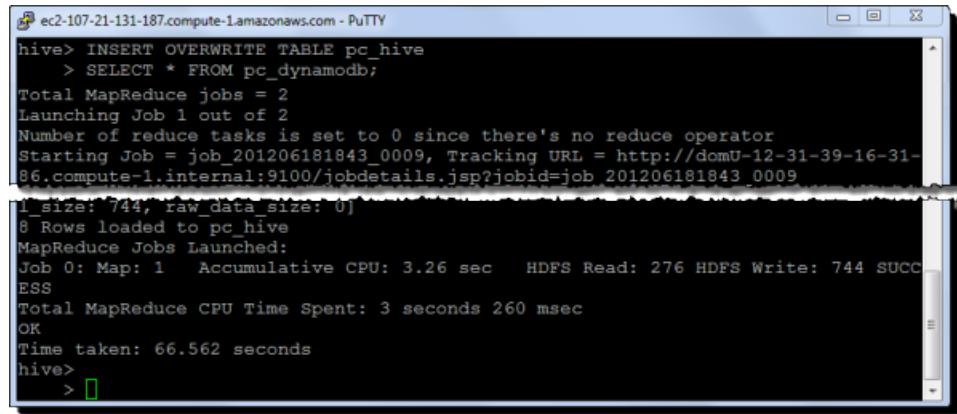
This statement creates the native Hive table. Notice that the `EXTERNAL` key word is not used.



```
hive> CREATE TABLE pc_hive (
    > id bigint
    > ,title string
    > ,isbn string
    > ,authors array<string>
    > ,price bigint
    > ,dimensions string
    > ,pagecount bigint
    > ,inpublication bigint
    > ,productcategory string
    > ,description string
    > ,bicycletype string
    > ,brand string
    > ,gender string
    > ,color array<string>
    > );
OK
Time taken: 13.262 seconds
hive>
```

2. Upload the DynamoDB table data into the new Hive table using an `INSERT` statement.

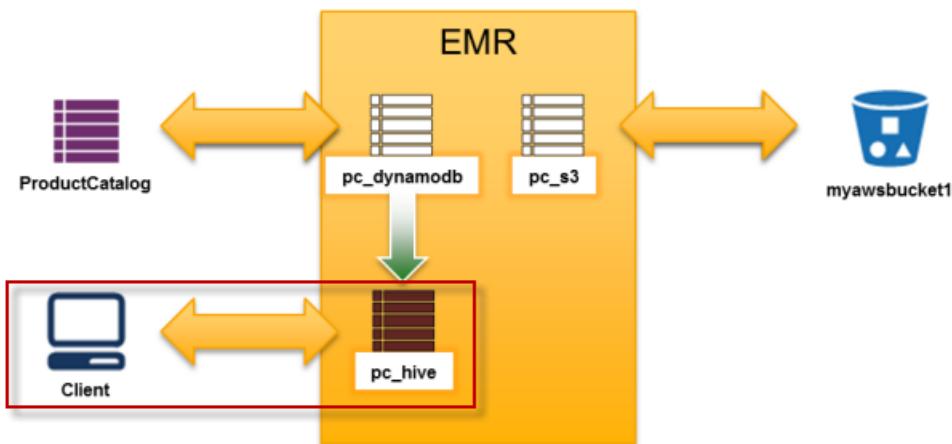
```
INSERT OVERWRITE TABLE pc_hive
SELECT * FROM pc_dynamodb;
```



```
hive> INSERT OVERWRITE TABLE pc_hive
    > SELECT * FROM pc_dynamodb;
Total MapReduce jobs = 2
Launching Job 1 out of 2
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_201206181843_0009, Tracking URL = http://domU-12-31-39-16-31-86.compute-1.internal:9100/jobdetails.jsp?jobid=job_201206181843_0009
l_size: 744, raw_data_size: 0)
8 Rows loaded to pc_hive
MapReduce Jobs Launched:
Job 0: Map: 1   Accumulative CPU: 3.26 sec   HDFS Read: 276 HDFS Write: 744 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 260 msec
OK
Time taken: 66.562 seconds
hive>
```

The data is now stored in the new `pc_hive` table.

You can now query the native table using Hive. For example, let's find the number of products in each product category.



To issue a SQL query on the native Hive table

- Copy and paste the following code into your Hive session.

```
SELECT ProductCategory, count(*)
FROM pc_hive
GROUP BY ProductCategory;
```

With this SELECT statement, you are querying the native Hive table, and not querying DynamoDB at all. Therefore, you are not using any DynamoDB provisioned throughput nor affecting your live tables.

A screenshot of a PuTTY terminal window titled "ec2-107-21-131-187.compute-1.amazonaws.com - PuTTY". The terminal displays the following output:

```
hive> SELECT ProductCategory, count(*)
> FROM pc_hive
> GROUP BY ProductCategory;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
Starting Job = job_201206181843_0010, Tracking URL = http://domU-12-31-39-16-31-
[redacted]
Ended Job = job_201206181843_0010
Counters:
MapReduce Jobs Launched:
Job 0: Map: 1 Reduce: 1 Accumulative CPU: 5.05 sec HDFS Read: 962 HDFS Write: 17 SUCCESS
Total MapReduce CPU Time Spent: 5 seconds 50 msec
OK
Bicycle 5
Book 3
Time taken: 65.143 seconds
hive> [redacted]
```

In the preceding screen shot, the query results are shown in the white box.

Tip

For more information about SQL statements using HiveQL, go to [SQL Operations](#) on the Getting Started page for [Apache Hive](#).

Final Cleanup

When you have completed this walkthrough, you can remove the DynamoDB table, the Amazon S3 bucket, and the Amazon EMR job flow to avoid incurring additional charges.

1. Delete the `ProductCatalog` table in DynamoDB.

Follow the instructions in the DynamoDB documentation for [Step 5: Delete Example Tables in DynamoDB](#).

2. Delete the bucket in Amazon S3.

Follow the instructions in the Amazon S3 documentation for [Deleting an Object](#).

3. Terminate the Amazon EMR job flow.

Open the AWS Management Console to <https://console.aws.amazon.com/elasticmapreduce/home> and right-click the job flow in the list. Choose **Terminate Job**, and then click the **Yes, Terminate** button.

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](#)

Limits in DynamoDB

The following table describes current limits within Amazon DynamoDB (or no limit, in some cases).

Note

Each limit listed below applies on a per-region basis.

Table names and secondary index names	For table and secondary index names, allowed characters are a-z, A-Z, 0-9, '_' (underscore), '-' (dash), and '.' (dot). Names can be between 3 and 255 characters long.
Table size	No practical limit in number of bytes or items.
Tables per account	By default, the number of tables per account is limited to 256 per region. However, you can request an increase in this limit. For more information, go to http://www.amazonaws.cn/support .
Hash or hash-and-range primary key: Number of hash key values	No practical limit.
Hash-and-range primary key: Number of range keys per hash value	No practical limit for tables without local secondary indexes. For a table with local secondary indexes, there is a limit on item collection sizes: For every distinct hash key value, the total sizes of all table and index items cannot exceed 10 GB. Depending on your item sizes, this may constrain the number of range keys per hash value. For more information, see Item Collection Size Limit (p. 313) .
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 minimum per table	1 read capacity unit and 1 write capacity unit.
Provisioned throughput minimum per global secondary index	1 read capacity unit and 1 write capacity unit.

Provisioned throughput limits	<p>DynamoDB is designed to scale without limits. However, there are some initial limits in place on provisioned throughput:</p> <p><i>US East (Northern Virginia) Region:</i></p> <ul style="list-style-type: none"> • Per table – 40,000 read capacity units or 40,000 write capacity units • Per account – 80,000 read capacity units or 80,000 write capacity units <p><i>All Other Regions:</i></p> <ul style="list-style-type: none"> • Per table – 10,000 read capacity units or 10,000 write capacity units • Per account – 20,000 read capacity units or 20,000 write capacity units <p>You can request an increase on any of these limits. For more information, go to http://www.amazonaws.cn/support.</p>
<code>UpdateTable</code> : Limits when increasing provisioned throughput on tables and global secondary indexes	<p>You can call <code>UpdateTable</code> as often as necessary to increase <code>ReadCapacityUnits</code> or <code>WriteCapacityUnits</code>. In a single <code>UpdateTable</code> operation, you can increase the provisioned throughput for a table, for any global secondary indexes on that table, or for any combination of these.</p> <p>The following conditions apply:</p> <ul style="list-style-type: none"> • You can increase <code>ReadCapacityUnits</code> or <code>WriteCapacityUnits</code> (or both), provided that you stay within your per-table and per-account limits. For more information, see the "Provisioned throughput limits" item in this section. • The new provisioned throughput settings do not take effect until the <code>UpdateTable</code> operation is complete. • You can call <code>UpdateTable</code> multiple times, until you reach the desired throughput capacity for your table or global secondary indexes.
<code>UpdateTable</code> : Limits when decreasing provisioned throughput on tables and global secondary indexes	<p>You can call <code>UpdateTable</code> to reduce provisioned throughput, but no more than four times in a single UTC calendar day. In an <code>UpdateTable</code> operation, you can decrease the provisioned throughput for a table, for any global secondary indexes on that table, or for any combination of these.</p> <p>For every table and global secondary index in an <code>UpdateTable</code> operation, you can decrease <code>ReadCapacityUnits</code> or <code>WriteCapacityUnits</code> (or both). The new provisioned throughput settings do not take effect until the <code>UpdateTable</code> operation is complete.</p>

Maximum concurrent CreateTable/UpdateTable/DeleteTable API requests	<p>In general, you can have up to 10 of these requests running at the same time. For example, the cumulative number of tables in the <i>CREATING</i>, <i>UPDATING</i> or <i>DELETING</i> state cannot exceed 10.</p> <p>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.</p>
Maximum number of secondary indexes per table	You can define up to 5 local secondary indexes and 5 global secondary indexes per table.
Maximum number of projected secondary index attributes per table	<p>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.</p> <p>In a <i>CreateTable</i> operation, if you specify a <i>ProjectionType</i> of <i>INCLUDE</i>, the total count of attributes specified in <i>NonKeyAttributes</i>, 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.</p> <p>This limit does not apply for secondary indexes with a <i>ProjectionType</i> of <i>KEYS_ONLY</i> or <i>ALL</i>.</p>
Attribute name lengths	<p>For the attribute names listed below, the name must be between 1 and 255 characters long, inclusive. The name can be any UTF-8 encodable character, but the total size of the UTF-8 string after encoding cannot exceed 255 bytes.</p> <ul style="list-style-type: none"> • Primary key attribute names. • The names of any user-specified projected attributes (applicable only to local secondary indexes). In a <i>CreateTable</i> operation, if you specify a <i>ProjectionType</i> of <i>INCLUDE</i>, then the names of the attributes in the <i>NonKeyAttributes</i> parameter are length-restricted. The <i>KEYS_ONLY</i> and <i>ALL</i> projection types are not affected. <p>For attributes that are not in the list above, there is a 64 KB limit on the length of the attribute name.</p>

Item size	<p>Cannot exceed 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.</p> <p>For attribute values that are of type binary, the application 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.</p> <p>These limits apply to items stored in tables, and also to items in secondary indexes.</p> <p>For each local secondary index on a table, there is a 400 KB limit on the total size of the following:</p> <ul style="list-style-type: none"> • 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.
Attribute values	Attribute values cannot be empty strings or empty sets.
Attribute name-value pairs per item	The cumulative size of attributes per item must be under 400 KB.
Maximum length of a hash key attribute value	2048 bytes.
Maximum length of a range key attribute value	1024 bytes.
String	All strings must conform to the UTF-8 encoding. Since UTF-8 is a variable width encoding, string sizes are determined using the UTF-8 bytes.
Number	<p>A number can have up to 38 digits precision and can be between 10^-128 to 10^+126. DynamoDB uses strings for the data transfer of numeric values in JSON notation. For more information, see JSON Is Used as a Transport Protocol Only (p. 478).</p> <p>If number precision is important, you should pass numbers to DynamoDB using strings that you convert from a number type.</p>
Document path maximum depth	DynamoDB supports nested attributes up to 32 levels deep.

Expression parameters	<p>The following limits apply when using expression parameters:</p> <ul style="list-style-type: none"> • Maximum length of an expression string: 4 KB. (For example, the size of the <i>ConditionExpression</i> <code>a=b</code> is three bytes.) • Maximum number of operators or functions allowed in an <i>UpdateExpression</i>: 300. (For example, the <i>UpdateExpression</i> <code>SET a = :val1 + :val2 + :val3</code> contains two "+" operators.) • Maximum number of operands for the <code>IN</code> comparator: 100. • Maximum length of substitution variables (sum of <i>ExpressionAttributeNames</i> and <i>ExpressionAttributeValues</i>): 2MB • Maximum length of any one expression attribute name or expression attribute value: 255 bytes. (For example, <code>#name</code> is five bytes; <code>:val</code> is four bytes.)
Number of values in an attribute set	No practical limit on the quantity of values, as long as the item containing the values fits within the 400 KB item size limit.
BatchGetItem item maximum per operation	Up to 100 items retrieved. The total size of all the items retrieved cannot exceed 16 MB.
BatchWriteItem item maximum per operation	Up to 25 <code>PutItem</code> or <code>DeleteItem</code> operations per <code>BatchWriteItem</code> call. The total size of all the items written cannot exceed 16 MB.
Query	The result set is limited to 1 MB per API call. You can use the <code>LastEvaluatedKey</code> from the query response to retrieve more results.
Scan	The maximum size of the scanned data set is 1 MB per API call. You can use the <code>LastEvaluatedKey</code> from the scan response to retrieve more results.

Document History for DynamoDB

The following table describes the important changes to the documentation since the last release of the *Amazon DynamoDB Developer Guide*.

- **API version:** 2012-08-10
- **Latest product release:** February 10, 2015
- **Latest documentation update:** February 10, 2015

Change	Description	Date Changed
Scan API for secondary indexes	In DynamoDB, a Scan operation reads all of the items in a table, applies user-defined filtering criteria, and returns the selected data items to the application. This same capability is now available for secondary indexes too. To scan a local secondary index or a global secondary index, you specify the index name and the name of its parent table. By default, an index Scan returns all of the data in the index; you can use a filter expression to narrow the results that are returned to the application. For more information, see Query and Scan Operations in DynamoDB (p. 183) in the <i>Amazon DynamoDB Developer Guide</i> and Scan in the <i>Amazon DynamoDB API Reference</i> .	In this release
Online operations for global secondary indexes	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. 260) .	January 27, 2015

Change	Description	Date Changed
Document model support with JSON	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 DynamoDB Data Types (p. 6) and Reading and Writing Items Using Expressions (p. 91) .	October 7, 2014
Flexible scaling	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. 597) .	October 7, 2014
Larger item sizes	The maximum item size in DynamoDB has increased from 64 KB to 400 KB. For more information, see Limits in DynamoDB (p. 597) .	October 7, 2014
Improved conditional expressions	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. 85) .	April 24, 2014
Query filter	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 hash key value and an optional range 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 Query and Scan Operations in DynamoDB (p. 183) .	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.</p> <p>For more information, see Using the AWS Management Console to Export and Import Data (p. 550) and Global Secondary Indexes (p. 253).</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: Object-persistence model • .NET: Document model and object-persistence model <p>These higher-level APIs are now documented here: Higher-Level Programming Interfaces for DynamoDB (p. 373).</p>	January 20, 2014
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 hash 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 range 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 in DynamoDB (p. 241) and Global Secondary Indexes (p. 253).</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 Fine-Grained Access Control for DynamoDB (p. 536).</p>	October 29, 2013
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 Provisioned Throughput in Amazon DynamoDB (p. 10).</p>	May 14, 2013

Change	Description	Date Changed
Parallel scans	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 Parallel Scan (p. 187) .	May 14, 2013
Local secondary indexes	DynamoDB adds support for local secondary indexes. You can define alternate range 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. 303) .	April 18, 2013
New API version	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, go to the Amazon DynamoDB API Reference .	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 Using IAM to Control Access to DynamoDB Resources (p. 527).</p>	April 4, 2013
PHP code samples updated for AWS SDK for PHP version 2	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 .	January 23, 2013
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 (Sao Paulo) Region. For the current list of supported endpoints, see Regions and Endpoints .	December 3, 2012
New endpoint	DynamoDB expands to the Asia Pacific (Sydney) Region. For the current list of supported endpoints, see Regions and Endpoints .	November 13, 2012

Change	Description	Date Changed
DynamoDB implements support for CRC32 checksums, supports strongly consistent batch gets, and removes restrictions on concurrent table updates.	<ul style="list-style-type: none"> • DynamoDB calculates a CRC32 checksum of the HTTP payload and returns this checksum in a new header, <code>x-amz-crc32</code>. For more information, see Making HTTP Requests to DynamoDB (p. 479). • 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. 666). • 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. 597). 	November 2, 2012
Best practices documentation	The Amazon DynamoDB Developer Guide identifies best practices for working with tables and items, along with recommendations for query and scan operations.	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 DynamoDB Data Types (p. 6). 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. 129) • Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API (p. 162) <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 Using the AWS SDKs with DynamoDB (p. 365).</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. For more information, see the Working with Items and Attributes (p. 356) topic in the Amazon DynamoDB Developer Guide.	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. 597) topic in the Amazon DynamoDB Developer Guide.	August 9, 2012
DynamoDB and Amazon Elastic MapReduce (Amazon EMR) integration video and walkthrough added	For more information, see Walkthrough: Using DynamoDB and Amazon Elastic MapReduce (p. 586) .	July 5, 2012
Signature Version 4 support	DynamoDB now supports Signature Version 4 for authenticating requests. To learn about authenticating your HTTP requests, see Making HTTP Requests to DynamoDB (p. 479) .	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 Getting Started with DynamoDB (p. 13) and DynamoDB Console (p. 354) sections have been updated for these features.	May 22, 2012
New endpoints	DynamoDB availability expands with new endpoints in the US West (Northern 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
BatchWriteItem API support	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. 671) . For information about working with items and using batch write feature using AWS SDKs, see Working with Items in DynamoDB (p. 85) and Using the AWS SDKs with DynamoDB (p. 365) .	April 19, 2012
Documented more error codes	For more information, see Handling Errors in DynamoDB Operations (p. 482) .	April 5, 2012
Updated Hive version to 0.7.1.3	Amazon Elastic MapReduce now supports a new version of Hive. For more information, see Querying and Joining Tables Using Amazon Elastic MapReduce (p. 564) .	March 13, 2012
New endpoint	DynamoDB expands to the Asia Pacific (Tokyo) Region. For the current list of supported endpoints, see Regions and Endpoints .	February 29, 2012

Change	Description	Date Changed
Updated Hive version to 0.7.1.2	Amazon Elastic MapReduce now supports a new version of Hive. For more information, see Querying and Joining Tables Using Amazon Elastic MapReduce (p. 564) .	February 28, 2012
ReturnedItemCount metric added	A new metric, <code>ReturnedItemCount</code> , provides the number of items returned in the response of a Query or Scan operation for DynamoDB is available for monitoring through CloudWatch. For more information, see Monitoring DynamoDB with CloudWatch (p. 519) .	February 24, 2012
Added a code snippet for iterating over scan results	The AWS SDK for PHP returns scan results as a SimpleXMLElement object. For an example of how to iterate through the scan results, see Scanning Tables Using the AWS SDK for PHP Low-Level API (p. 234) .	February 2, 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> <ul style="list-style-type: none"> <li data-bbox="592 804 1258 861">Working with Items Using the AWS SDK for Java Document API (p. 110). <li data-bbox="592 889 1258 946">Working with Items Using the AWS SDK for .NET Low-Level API (p. 132). <li data-bbox="592 973 1258 1030">Working with Items Using the AWS SDK for PHP Low-Level API (p. 167). 	January 25, 2012
Initial product release	DynamoDB is introduced as a new service in Beta release.	January 18, 2012

DynamoDB Appendix

Topics

- [Example Tables and Data \(p. 609\)](#)
- [Creating Example Tables and Uploading Data \(p. 614\)](#)
- [Reserved Words in DynamoDB \(p. 649\)](#)
- [Legacy Conditional Parameters \(p. 659\)](#)
- [Current API Version \(2012-08-10\) \(p. 665\)](#)
- [Previous API Version \(2011-12-05\) \(p. 666\)](#)

Example Tables and Data

The *Amazon DynamoDB Developer Guide* uses the following sample tables to illustrate working with tables, items and the query operations. The following table lists tables, their primary key attributes and their types.

Table Name	Primary Key Type	Hash Attribute Name and Type	Range Attribute Name and Type
ProductCatalog (<u>Id</u> , ...)	Hash	Attribute Name: Id Type: Number	-
Forum (<u>Name</u> , ...)	Hash	Attribute Name: Name Type: String	-
Thread (<u>ForumName</u> , <u>Subject</u> , ...)	Hash and Range	Attribute Name: ForumName Type: String	Attribute Name: Subject Type: String
Reply (<u>Id</u> , <u>ReplyDateTime</u> , ...)	Hash and Range	Attribute Name: Id Type: String	Attribute Name: Reply- DateTime Type: String

The Reply table has the following local secondary index:

Index Name	Attribute to Index	Projected Attributes
PostedBy-index	PostedBy	Table and Index Keys

The ProductCatalog table represents a table in which each product item is uniquely identified by an Id. Because each table is like a property bag, you can store all kinds of products in this table. For illustration, we store book and bicycle items. In a DynamoDB table, an attribute can be multivalued. For example, a book can have multiple authors. All the book items stored have an Authors attribute that stores one or more author names and the bicycle items have a Color multivalued attribute for the available colors.

The Forum, Thread, and Reply tables are modeled after the AWS forums. Each AWS service maintains one or more forums. Customers start a thread by posting a message that has a unique subject. Each thread might receive one or more replies at different times. These replies are stored in the Reply table. For more information, see [AWS Forums](#).

DynamoDB does not support table joins. Additionally, when accessing data, queries are the most efficient and table scans should be avoided because of performance issues. These should be taken into consideration when you design your table schemas. For example, you might want to join the Reply and Thread tables. The Reply table Id attribute is set up as a concatenation of the forum name and subject values with a "#" in between to enable efficient queries. If you have a reply item, you can parse the Id to find forum name and thread subject. You can then use these values to query the Forum or the Thread table as you need.

For more information about the DynamoDB data model, see [DynamoDB Data Model \(p. 3\)](#).

ProductCatalog Table - Sample Data

The following table shows the sample data that the code example in the getting started section uploads to the ProductCatalog table. For more information, see [Step 3: Load Data into Tables \(p. 19\)](#).

Id (Primary Key)	Other Attributes
101	<pre>{ Title = "Book 101 Title" ISBN = "111-1111111111" Authors = "Author 1" Price = -2 Dimensions = "8.5 x 11.0 x 0.5" PageCount = 500 InPublication = true ProductCategory = "Book" }</pre>
102	<pre>{ Title = "Book 102 Title" ISBN = "222-2222222222" Authors = ["Author 1", "Author 2"] Price = 20 Dimensions = "8.5 x 11.0 x 0.8" PageCount = 600 InPublication = true ProductCategory = "Book" }</pre>

Amazon DynamoDB Developer Guide
ProductCatalog Table - Sample Data

Id (Primary Key)	Other Attributes
103	<pre>{ Title = "Book 103 Title" ISBN = "333-3333333333" Authors = ["Author 1", "Author2", "Author 3"] Price = 200 Dimensions = "8.5 x 11.0 x 1.5" PageCount = 700 InPublication = false ProductCategory = "Book" }</pre>
201	<pre>{ Title = "18-Bicycle 201" Description = "201 description" BicycleType = "Road" Brand = "Brand-Company A" Price = 100 Gender = "M" Color = ["Red", "Black"] ProductCategory = "Bike" }</pre>
202	<pre>{ Title = "21-Bicycle 202" Description = "202 description" BicycleType = "Road" Brand = "Brand-Company A" Price = 200 Gender = "M" Color = ["Green", "Black"] ProductCategory = "Bike" }</pre>
203	<pre>{ Title = "19-Bicycle 203" Description = "203 description" BicycleType = "Road" Brand = "Brand-Company B" Price = 300 Gender = "W" Color = ["Red", "Green", "Black"] ProductCategory = "Bike" }</pre>

Id (Primary Key)	Other Attributes
204	<pre>{ Title = "18-Bicycle 204" Description = "204 description" BicycleType = "Mountain" Brand = "Brand-Company B" Price = 400 Gender = "W" Color = ["Red"] ProductCategory = "Bike" }</pre>
205	<pre>{ Title = "20-Bicycle 205" Description = "205 description" BicycleType = "Hybrid" Brand = "Brand-Company C" Price = 500 Gender = "B" Color = ["Red", "Black"] ProductCategory = "Bike" }</pre>

Forum Table - Sample Data

The following table shows the sample data that the code example in the getting started section uploads to the Forum table. For more information, see [Step 3: Load Data into Tables \(p. 19\)](#).

Name (Primary Key)	Other Attributes
"DynamoDB"	<pre>{ Category="Amazon Web Services" Threads=3 Messages=4 Views=1000 LastPostBy="User A" LastPostDateTime= "2012-01-03T00:40:57.165Z" }</pre>
"Amazon S3"	<pre>{ Category="AWS" Threads=1 }</pre>

Thread Table - Sample Data

The following table shows the sample data that the code example in the getting started section uploads to the Thread table. For more information, see [Step 3: Load Data into Tables \(p. 19\)](#).

Note that, the LastPostDateTime values shown in the sample data are for illustration only. The code example generates the date and time values so that your table has relatively current dates in your table.

Primary Key	Other Attributes
ForumName = "DynamoDB" Subject = "DynamoDB Thread 1"	{ Message = "DynamoDB thread 1 message text" LastPostedBy = "User A" Views = 0 Replies = 0 Answered = 0 Tags = ["index", "primarykey", "table"] LastPostDateTime = "2012-01-03T00:40:57.165Z" }
ForumName = "DynamoDB" Subject = "DynamoDB Thread 2"	{ Message = "DynamoDB thread 2 message text" LastPostedBy = "User A" Views = 0 Replies = 0 Answered = 0 Tags = ["index", "primarykey", "rangekey"] LastPostDateTime = "2012-01-03T00:40:57.165Z" }
ForumName = "Amazon S3" Subject = "Amazon S3 Thread 1"	{ Message = "Amazon S3 Thread 1 message text" LastPostedBy = "User A" Views = 0 Replies = 0 Answered = 0 Tags = ["largeobject", "multipart upload"] LastPostDateTime = "2012-01-03T00:40:57.165Z" }

Reply Sample Data

The following table shows the sample data that the code example in the getting started section uploads to the Reply table. For more information, see [Step 3: Load Data into Tables \(p. 19\)](#).

Note that, the LastPostDateTime values shown in the sample data are for illustration only. The code example generates the date and time values so that your table has relatively current dates in your table.

Primary Key	Other Attributes
Id = "DynamoDB#DynamoDB Thread 1" ReplyDateTime = "2011-12-11T00:40:57.165Z"	{ Message = "DynamoDB Thread 1 Reply 1 text" PostedBy = "User A" }
Id = "DynamoDB#DynamoDB Thread 1" ReplyDateTime = "2011-12-18T00:40:57.165Z"	{ Message = "DynamoDB Thread 1 Reply 1 text" PostedBy = "User A" }
Id = "DynamoDB#DynamoDB Thread 1" ReplyDateTime = "2011-12-25T00:40:57.165Z"	{ Message = "DynamoDB Thread 1 Reply 3 text" PostedBy = "User B" }
Id = "DynamoDB#DynamoDB Thread 2" ReplyDateTime = "2011-12-25T00:40:57.165Z"	{ Message = "DynamoDB Thread 2 Reply 1 text" PostedBy = "User A" }
Id = "DynamoDB#DynamoDB Thread 2" ReplyDateTime = "2012-01-03T00:40:57.165Z"	{ Message = "DynamoDB Thread 2 Reply 2" PostedBy = "User A" }

Creating Example Tables and Uploading Data

Topics

- [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 614\)](#)
- [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 623\)](#)
- [Creating Example Tables and Uploading Data Using the AWS SDK for PHP \(p. 641\)](#)

In the Getting Started, you first create tables using the DynamoDB console and then upload data using the code provided. This appendix provides code to both create the tables and upload 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 [Example Tables and Data \(p. 609\)](#). For step-by-step instructions to run this code using Eclipse, see [Running Java Examples for DynamoDB \(p. 367\)](#).

```
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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

    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: hash key and type

            // Parameter6/7: range key and 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);

        } catch (Exception e) {
            System.out.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
        }
    }

    private void deleteTable(String tableName) {
        Table table = dynamoDB.getTable(tableName);
        table.delete();
    }

    private void createTable(String tableName, Long readCapacity,
        Long writeCapacity, String hashKey, String hashKeyType,
        String rangeKey, String rangeKeyType) {
        CreateTableRequest request = new CreateTableRequest()
            .withTableName(tableName)
            .withAttributeDefinitions(AttributeDefinition.builder()
                .withAttributeName(hashKey)
                .withAttributeType(hashKeyType)
                .build())
            .withProvisionedThroughput(ProvisionedThroughput.builder()
                .withReadCapacityUnits(readCapacity)
                .withWriteCapacityUnits(writeCapacity)
                .build());
        if (rangeKey != null) {
            request.withKeySchema(KeySchemaElement.builder()
                .withAttributeName(rangeKey)
                .withKeyType(rangeKeyType)
                .build());
        }
        dynamoDB.createTable(request);
    }

    private void loadSampleProducts(String tableName) {
        Item item = Item.builder()
            .withString("Id", "1")
            .withString("Name", "Amazon Echo Dot")
            .withString("Description", "A compact smart speaker with built-in Alexa")
            .withNumber("Price", 50.0)
            .withNumber("StockLevel", 100)
            .withString("Category", "Electronics")
            .withString("ImageURL", "https://image.com/echo_dot.jpg")
            .withString("LastUpdated", dateFormatter.format(new Date()))
            .build();
        Table table = dynamoDB.getTable(tableName);
        table.putItem(item);
    }
}
```

```

        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 hashKeyName, String hashKeyType) {

    createTable(tableName, readCapacityUnits, writeCapacityUnits,
        hashKeyName, hashKeyType, null, null);
}

private static void createTable(
    String tableName, long readCapacityUnits, long writeCapacityUnits,
    String hashKeyName, String hashKeyType,
    String rangeKeyName, String rangeKeyType) {

    try {

        ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
        keySchema.add(new KeySchemaElement()
            .withAttributeName(hashKeyName)
            .withKeyType(KeyType.HASH));

        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<AttributeDefinition>();
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName(hashKeyName)
            .withAttributeType(hashKeyType));

        if (rangeKeyName != null) {
            keySchema.add(new KeySchemaElement()
                .withAttributeName(rangeKeyName)

```

```

        .withKeyType(KeyType.RANGE));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName(rangeKeyName)
    .withAttributeType(rangeKeyType));
}

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 Ar
rayList<LocalSecondaryIndex>();
    localSecondaryIndexes.add(new LocalSecondaryIndex()
        .withIndexName("PostedBy-Index")
        .withKeySchema(
            new KeySchemaElement().withAttributeName(hashKey
Name).withKeyType(KeyType.HASH),
            new KeySchemaElement().withAttributeName("PostedBy")
        .withKeyType(KeyType.RANGE))
        .withProjection(new Projection().withProjectionType(Projec
tionType.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)
    .withString("Gender", "M")
    // Men's
    .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)
.withString("Gender", "M")
.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)
.withString("Gender", "W")
// Women's
.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)
.withString("Gender", "W")
.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)
.withString("Gender", "B")
// Boy's
.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", "primarykey", "rangekey")));
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 timel = (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 [Example Tables and Data \(p. 609\)](#). For step-by-step instructions to run this code in Visual Studio, see [Running .NET Examples for DynamoDB \(p. 369\)](#).

```
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 = Projection
Type.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" ] = "333-3333333333";
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[ "Gender" ] = "M";
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[ "Gender" ] = "M"; // Mens.
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[ "Gender" ] = "W";
```

```
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["Gender"] = "W"; // Women.
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["Gender"] = "B"; // Boys.
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);

    }

}

}
```

Creating Example Tables and Uploading Data Using the AWS SDK for PHP

The following PHP code example creates tables. The resulting tables structure and data is shown in [Example Tables and Data \(p. 609\)](#). For step-by-step instructions to run this code, see [Running PHP Examples \(p. 371\)](#).

```
<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

$tableNames = array();

$tableName = 'ProductCatalog';
echo "Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Id',
            'AttributeType' => 'N'
        )
    ),
    'KeySchema' => array(
        array(

```

```
        'AttributeName' => 'Id',
        'KeyType' => 'HASH'
    )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 10,
    'WriteCapacityUnits' => 5
)
));
$stableNames[] = $tableName;

$tableName = 'Forum';
echo "Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Name',
            'AttributeType' => 'S'
        )
),
'KeySchema' => array(
        array(
            'AttributeName' => 'Name',
            'KeyType' => 'HASH'
        )
),
'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 10,
    'WriteCapacityUnits' => 5
)
));
$stableNames[] = $tableName;

$tableName = 'Thread';
echo "Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'ForumName',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'Subject',
            'AttributeType' => 'S'
        )
),
'KeySchema' => array(
        array(
            'AttributeName' => 'ForumName',
            'KeyType' => 'HASH'
        ),
        array(
            'AttributeName' => 'Subject',
            'KeyType' => 'RANGE'
        )
)
```

```
        )
    ),
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits'      => 10,
        'WriteCapacityUnits'    => 5
    )
));
$stableNames[] = $tableName;

$tableName = 'Reply';
echo "Creating table $tableName..." . PHP_EOL;

$response = $client->createTable(array(
    'TableName' => $tableName,
    'AttributeDefinitions' => array(
        array(
            'AttributeName' => 'Id',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'ReplyDateTime',
            'AttributeType' => 'S'
        ),
        array(
            'AttributeName' => 'PostedBy',
            'AttributeType' => 'S'
        )
),
    'LocalSecondaryIndexes' => array(
        array(
            'IndexName' => 'PostedBy-index',
            'KeySchema' => array(
                array(
                    'AttributeName' => 'Id',
                    'KeyType' => 'HASH'
                ),
                array(
                    'AttributeName' => 'PostedBy',
                    'KeyType' => 'RANGE'
                )
            ),
            'Projection' => array(
                'ProjectionType' => 'KEYS_ONLY',
            ),
        ),
    ),
    'KeySchema' => array(
        array(
            'AttributeName' => 'Id',
            'KeyType' => 'HASH'
        ),
        array(
            'AttributeName' => 'ReplyDateTime',
            'KeyType' => 'RANGE'
        )
),
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits'      => 10,
```

```

        'WriteCapacityUnits' => 5
    )
));
$tableNames[] = $tableName;

foreach($tableNames as $tableName) {
    echo "Waiting for table $tableName to be created." . PHP_EOL;
    $client->waitUntilTableExists(array('TableName' => $tableName));
    echo "Table $tableName has been created." . PHP_EOL;
}

?>

```

The following PHP code example uploads data to the tables. The resulting table structure and data is shown in [Example Tables and Data \(p. 609\)](#).

```

<?php

use Aws\DynamoDb\DynamoDbClient;

$client = DynamoDbClient::factory(array(
    'profile' => 'default',
    'region' => 'us-west-2' // replace with your desired region
));

# Setup some local variables for dates

date_default_timezone_set('UTC');

$oneDayAgo = date('Y-m-d H:i:s', strtotime('-1 days'));
$sevenDaysAgo = date('Y-m-d H:i:s', strtotime('-7 days'));
$fourteenDaysAgo = date('Y-m-d H:i:s', strtotime('-14 days'));
$twentyOneDaysAgo = date('Y-m-d H:i:s', strtotime('-21 days'));

$tableName = 'ProductCatalog';
echo "Adding data to the $tableName table..." . PHP_EOL;

$response = $client->batchWriteItem(array(
    'RequestItems' => array(
        $tableName => array(
            array(
                'PutRequest' => array(
                    'Item' => array(
                        'Id' => array('N' => '1101'),
                        'Title' => array('S' => 'Book 101 Title'),
                        'ISBN' => array('S' => '111-1111111111'),
                        'Authors' => array('SS' => array('Author1')),
                        'Price' => array('N' => '2'),
                        'Dimensions' => array('S' => '8.5 x 11.0 x 0.5'),
                        'PageCount' => array('N' => '500'),
                        'InPublication' => array('N' => '1'),
                        'ProductCategory' => array('S' => 'Book')
                    )
                ),
            ),
        ),
        array(

```

```

'PutRequest' => array(
    'Item' => array(
        'Id'                => array('N' => '102'),
        'Title'              => array('S' => 'Book 102 Title'),
        'ISBN'               => array('S' => '222-222222222222'),
        'Authors'             => array('SS' => array('Author1',
'Author2'))),
        'Price'              => array('N' => '20'),
        'Dimensions'          => array('S' => '8.5 x 11.0 x 0.8'),

        'PageCount'           => array('N' => '600'),
        'InPublication'       => array('N' => '1'),
        'ProductCategory'     => array('S' => 'Book')

    )
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'                => array('N' => '103'),
            'Title'              => array('S' => 'Book 103 Title'),
            'ISBN'               => array('S' => '333-333333333333'),
            'Authors'             => array('SS' => array('Author1',
'Author2'))),
            'Price'              => array('N' => '2000'),
            'Dimensions'          => array('S' => '8.5 x 11.0 x 1.5'),

            'PageCount'           => array('N' => '600'),
            'InPublication'       => array('N' => '0'),
            'ProductCategory'     => array('S' => 'Book')

        )
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'                => array('N' => '201'),
            'Title'              => array('S' => '18-Bike-201'),
            'Description'         => array('S' => '201 Description'),

            'BicycleType'        => array('S' => 'Road'),
            'Brand'               => array('S' => 'Mountain A'),
            'Price'               => array('N' => '100'),
            'Gender'              => array('S' => 'M'),
            'Color'               => array('SS' => array('Red',
'Black'))),
            'ProductCategory'     => array('S' => 'Bicycle')

        )
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'                => array('N' => '202'),
            'Title'              => array('S' => '21-Bike-202'),

```

```

        'Description'      => array('S' => '202 Description'),
        'BicycleType'     => array('S' => 'Road'),
        'Brand'           => array('S' => 'Brand-Company A'),
        'Price'           => array('N' => '200'),
        'Gender'          => array('S' => 'M'),
        'Color'           => array('SS' => array('Green',
        'Black'))),
        'ProductCategory' => array('S' => 'Bicycle')
    ),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'           => array('N' => '203'),
            'Title'        => array('S' => '19-Bike-203'),
            'Description'  => array('S' => '203 Description'),
            'BicycleType'  => array('S' => 'Road'),
            'Brand'         => array('S' => 'Brand-Company B'),
            'Price'         => array('N' => '300'),
            'Gender'        => array('S' => 'W'),
            'Color'         => array('SS' => array('Red', 'Green',
            'Black'))),
            'ProductCategory' => array('S' => 'Bicycle')
        )
    ),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'           => array('N' => '204'),
            'Title'        => array('S' => '18-Bike-204'),
            'Description'  => array('S' => '204 Description'),
            'BicycleType'  => array('S' => 'Mountain'),
            'Brand'         => array('S' => 'Brand-Company B'),
            'Price'         => array('N' => '400'),
            'Gender'        => array('S' => 'W'),
            'Color'         => array('SS' => array('Red')),
            'ProductCategory' => array('S' => 'Bicycle')
        )
    ),
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'           => array('N' => '205'),
            'Title'        => array('S' => '20-Bike-205'),
            'Description'  => array('S' => '205 Description'),
            'BicycleType'  => array('S' => 'Hybrid'),
            'Brand'         => array('S' => 'Brand-Company C'),
            'Price'         => array('N' => '500'),
        )
    ),
),

```

```

        'Gender'      => array('S' => 'B'),
        'Color'       => array('SS' => array('Red',
'Black'))),
        'ProductCategory' => array('S' => 'Bicycle')

    )
)
),
),
));
echo "done." . PHP_EOL;



|                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$tableName = 'Forum';                       | echo "Adding data to the \$tableName table..." . PHP_EOL;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| \$response = \$client->batchWriteItem(array( | 'RequestItems' => array(                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| \$tableName => array(                        | array(             'PutRequest' => array(                 'Item' => array(                     'Name'      => array('S' => 'Amazon DynamoDB'),                     'Category' => array('S' => 'Amazon Web Services'),                     'Threads'  => array('N' => '0'),                     'Messages' => array('N' => '0'),                     'Views'    => array('N' => '1000')                 )             )         ),         array(             'PutRequest' => array(                 'Item' => array(                     'Name'      => array('S' => 'Amazon S3'),                     'Category' => array('S' => 'Amazon Web Services'),                     'Threads'  => array('N' => '0')                 )             )         ),         )     ) )); echo "done." . PHP_EOL; |
| \$tableName = 'Reply';                       | echo "Adding data to the \$tableName table..." . PHP_EOL;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| \$response = \$client->batchWriteItem(array( | 'RequestItems' => array(                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| \$tableName => array(                        | array(             'PutRequest' => array(                 'Item' => array(                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |


```

```

        'Id'           => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 1'),
        'ReplyDateTime' => array('S' => $fourteenDaysAgo),
        'Message'       => array('S' => 'DynamoDB Thread 1 Reply
2 text'),
        'PostedBy'      => array('S' => 'User B')
    )
)
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'           => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 2'),
            'ReplyDateTime' => array('S' => $twentyOneDaysAgo),
            'Message'       => array('S' => 'DynamoDB Thread 2 Reply
3 text'),
            'PostedBy'      => array('S' => 'User B')
        )
    )
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'           => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 2'),
            'ReplyDateTime' => array('S' => $sevenDaysAgo),
            'Message'       => array('S' => 'DynamoDB Thread 2 Reply
2 text'),
            'PostedBy'      => array('S' => 'User A')
        )
    )
),
array(
    'PutRequest' => array(
        'Item' => array(
            'Id'           => array('S' => 'Amazon DynamoDB#DynamoDB
Thread 2'),
            'ReplyDateTime' => array('S' => $oneDayAgo),
            'Message'       => array('S' => 'DynamoDB Thread 2 Reply
1 text'),
            'PostedBy'      => array('S' => 'User A')
        )
    )
),
)
);
));

echo "done." . PHP_EOL;

?>

```

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 \(p. 94\)](#).

```
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
```

```
REEXP
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 describes the legacy conditional parameters in DynamoDB and how to build conditional expressions with them.

Note

New applications should use expression parameters instead. For more information, see [Reading and Writing Items Using Expressions \(p. 91\)](#).

The following table shows the legacy conditional parameters, and the expression parameters that have superseded them:

Legacy Conditional Parameter	Expression Parameter
AttributesToGet	ProjectionExpression
Expected and ConditionalOperator	ConditionExpression
AttributeUpdates	UpdateExpression
QueryFilter and ScanFilter	FilterExpression

DynamoDB does not allow mixing legacy conditional parameters and expression parameters in a single API call. For example, calling the `Query` API with `AttributesToGet` and `ConditionExpression` will result in an error.

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.
- `NONE` - 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: [ 100 ]  
    }  
}
```

- A product category begins with "Bo".

```
{  
    ProductCategory: {  
        ComparisonOperator: BEGINS_WITH,  
        AttributeValueList: [ "Bo" ]  
    }  
}
```

- A product is available in either red, green, or black:

```
{  
    Color: {  
        ComparisonOperator: EQ,  
        AttributeValueList: [  
            "Black", "Red", "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: [ 100, 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: [ 201, 203, 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`.

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 *AND*ed together.

```
{  
    ProductCategory: {  
        ComparisonOperator: EQ,  
        AttributeValueList: [ "Book" ]  
    },  
    PageCount: {  
        ComparisonOperator: GE,  
        AttributeValueList: [ 600 ]  
    }  
}
```

You can use *ConditionalOperator* 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: [ "Book" ]  
    },  
    PageCount: {  
        ComparisonOperator: GE,  
        AttributeValueList: [ 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: [ "Mountain" ]  
    },  
    Brand: {  
        ComparisonOperator: EQ,  
        AttributeValueList: [ "Brand-Company A" ]  
    },  
    Price: {  
        ComparisonOperator: GT,  
        AttributeValueList: [ 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 example using a legacy condition:

```
{  
    TableName: "Product",  
    Item: {  
        Id: 600,  
        Title: "Book 600 Title"  
    },  
    Expected: {  
        InPublication: {  
            Exists: true,  
            Value : false  
        }  
    }  
}
```

The following example does the same thing, but does not use a legacy condition:

```
{  
    TableName: "Product",  
    Item: {  
        Id: 600,  
        Title: "Book 600 Title"  
    },  
    Expected: {  
        InPublication: {  
            ComparisonOperator: EQ,  
            AttributeValueList: [ 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:

```
{  
    TableName: "Product",  
    Item: {  
        Id: 500,  
        Title: "Book 500 Title"  
    },  
    Expected: {  
        Id: {  
            Exists: false  
        }  
    }  
}
```

The following example does the same thing, but does not use a legacy condition:

```
{  
    TableName: "Product",  
    Item: {  
        Id: 500,  
        Title: "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 API Version (2012-08-10)

The current version of the DynamoDB API is 2012-08-10. For complete documentation, go to the [Amazon DynamoDB API Reference](#)

The following operations are supported:

- [BatchGetItem](#)
- [BatchWriteItem](#)
- [CreateTable](#)
- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [.GetItem](#)
- [ListTables](#)

- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)
- [UpdateTable](#)

Previous API Version (2011-12-05)

This section documents the operations available in DynamoDB API version 2011-12-05. This is the previous version of the API, which is being maintained for backward compatibility with existing applications.

New applications should use the current API version (2012-08-10). For more information, see [Current API Version \(2012-08-10\) \(p. 665\)](#).

We recommend that you migrate your applications to the latest API version (2012-08-10), since new DynamoDB features (such as secondary indexes) will not be backported to the previous API version.

Topics

- [BatchGetItem \(p. 666\)](#)
- [BatchWriteItem \(p. 671\)](#)
- [CreateTable \(p. 677\)](#)
- [DeleteItem \(p. 681\)](#)
- [DeleteTable \(p. 685\)](#)
- [DescribeTables \(p. 688\)](#)
- [.GetItem \(p. 691\)](#)
- [ListTables \(p. 693\)](#)
- [PutItem \(p. 695\)](#)
- [Query \(p. 700\)](#)
- [Scan \(p. 706\)](#)
- [UpdateItem \(p. 714\)](#)
- [UpdateTable \(p. 719\)](#)

BatchGetItem

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 [Capacity Units Calculations for Various Operations \(p. 57\)](#).

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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

Name	Description	Required
<i>Table</i>	The name of the table containing the items to get. The entry is simply a string specifying an existing table with no label. Type: String Default: None	Yes
<i>Table:Keys</i>	The primary key values that define the items in the specified table. For more information about primary keys, see Primary Key (p. 5) . Type: Keys	Yes
<i>Table:AttributesToGet</i>	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. Type: Array	No
<i>Table:ConsistentRead</i>	If set to true, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 855

{
    "Responses": {
        "Table1": {
            "Items": [
                {"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"N": "AttributeValue"}, "AttributeName3": {"SS": ["AttributeValue", "AttributeValue", "AttributeValue"]}},
                {"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"S": "AttributeValue"}, "AttributeName3": {"NS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}
            ],
            "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
<i>Responses</i>	Table names and the respective item attributes from the tables. Type: Map
<i>Table</i>	The name of the table containing the items. The entry is simply a string specifying the table with no label. Type: String
<i>Items</i>	Container for the attribute names and values meeting the operation parameters. Type: Map of attribute names to and their data types and values.
<i>ConsumedCapacityUnits</i>	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 Specifying Read and Write Requirements for Tables (p. 55) . Type: Number
<i>UnprocessedKeys</i>	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 <i>UnprocessedKeys</i> value is in the same form as a <i>RequestItems</i> parameter (so the value can be provided directly to a subsequent <code>BatchGetItem</code> operation). For more information, see the above <i>RequestItems</i> parameter. Type: Array

Name	Description
<i>UnprocessedKeys</i> : 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) . Type: Array of attribute name-value pairs.
<i>UnprocessedKeys</i> : 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.
<i>UnprocessedKeys</i> : 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
<i>ProvisionedThroughputExceededException</i>	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 \(p. 85\)](#).

Sample Request

The following sample requests attributes from two different tables.

```
// This header is abbreviated.
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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" }, "RangeKeyEle  
ment": { "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 the previous API version (2011-12-05). For the most recent API version, go to the [Amazon DynamoDB API Reference](#).

Description

This operation enables you to put or delete several items across multiple tables in a single API call.

To upload one item, you can use the PutItem API and to delete one item, you can use the DeleteItem API. However, when you want to upload or delete large amounts of data, such as uploading large amounts of data from Amazon Elastic MapReduce (Amazon EMR) or migrate data from another database in to DynamoDB, this API 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, the BatchWriteItem API provides an alternative where the API performs the specified put and delete operations 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, the API performs the specified operations in parallel giving you 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 \(p. 54\)](#).

When using this API, 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 APIs such as `PutItem` and `DeleteItem`. For example, the `DeleteItem` API 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 the `PutItem` and `DeleteItem` APIs, `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 Sample DynamoDB JSON Request and Response \(p. 481\).
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 the `BatchWriteItem` API 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 API.

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 \(p. 85\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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": {
                        "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` API 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 the previous API version (2011-12-05). For the most recent API version, go to 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. 688) API to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).  
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
<code>TableName</code>	The name of the table to create. Allowed characters are a-z, A-Z, 0-9, '_' (underscore), '-' (dash), and '.' (dot). Names can be between 3 and 255 characters long. Type: String	Yes

Name	Description	Required
<i>KeySchema</i>	<p>The primary key (simple or composite) structure for the table. A name-value pair for the <i>HashKeyElement</i> is required, and a name-value pair for the <i>RangeKeyElement</i> 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 AttributeType are "S" (string), "N" (numeric), or "B" (binary).</p> <p>Type: Map of <i>HashKeyElement</i>, or <i>HashKeyElement</i> and <i>RangeKeyElement</i> for a composite primary key.</p>	Yes
<i>ProvisionedThroughput</i>	<p>New throughput for the specified table, consisting of values for <i>ReadCapacityUnits</i> and <i>WriteCapacityUnits</i>. For details, see Specifying Read and Write Requirements for Tables (p. 55).</p> <p>Note For current maximum/minimum values, see Limits in DynamoDB (p. 597).</p> <p>Type: Array</p>	Yes
<i>ProvisionedThroughput:ReadCapacityUnits</i>	<p>Sets the minimum number of consistent <i>ReadCapacityUnits</i> 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 <i>ReadCapacityUnits</i> per second provides 100 eventually consistent <i>ReadCapacityUnits</i> per second.</p> <p>Type: Number</p>	Yes
<i>ProvisionedThroughput:WriteCapacityUnits</i>	<p>Sets the minimum number of <i>WriteCapacityUnits</i> 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: CSOC7TJPLR000KIRLG0HVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
```

```

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
<i>TableDescription</i>	A container for the table properties.
<i>CreationDateTime</i>	Date when the table was created in UNIX epoch time . Type: Number
<i>KeySchema</i>	The primary key (simple or composite) structure for the table. A name-value pair for the <i>HashKeyElement</i> is required, and a name-value pair for the <i>RangeKeyElement</i> is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5) . Type: Map of <i>HashKeyElement</i> , or <i>HashKeyElement</i> and <i>RangeKeyElement</i> for a composite primary key.
<i>ProvisionedThroughput</i>	Throughput for the specified table, consisting of values for <i>ReadCapacityUnits</i> and <i>WriteCapacityUnits</i> . See Specifying Read and Write Requirements for Tables (p. 55) . Type: Array
<i>ProvisionedThroughput :ReadCapacity-Units</i>	The minimum number of <i>ReadCapacityUnits</i> consumed per second before DynamoDB balances the load with other operations Type: Number
<i>ProvisionedThroughput :WriteCapacity-Units</i>	The minimum number of <i>ReadCapacityUnits</i> consumed per second before <i>WriteCapacityUnits</i> balances the load with other operations Type: Number
<i>TableName</i>	The name of the created table. Type: String

Name	Description
<i>TableStatus</i>	The current state of the table (<i>CREATING</i>). Once the table is in the <i>ACTIVE</i> state, you can put data in it. Use the DescribeTables (p. 688) API to check the status of the table. Type: String

Special Errors

Error	Description
ResourceInUseException	Attempt to recreate an already existing table.
LimitExceededException	The number of simultaneous table requests (cumulative number of tables in the <i>CREATING</i> , <i>DELETING</i> or <i>UPDATING</i> state) exceeds the maximum allowed. Note For current maximum/minimum values, see Limits in DynamoDB (p. 597) .

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 \(p. 54\)](#).

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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: CSOC7TJPLR0OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 688\)](#)
- [DeleteTable \(p. 685\)](#)

DeleteItem

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 Sample DynamoDB JSON Request and Response \(p. 481\).  
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
<i>TableName</i>	The name of the table containing the item to delete. Type: String	Yes
<i>Key</i>	The primary key that defines the item. For more information about primary keys, see Primary Key (p. 5) . Type: Map of <i>HashKeyElement</i> to its value and <i>RangeKeyElement</i> to its value.	Yes
<i>Expected</i>	Designates an attribute for a conditional delete. The <i>Expected</i> parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute has a particular value before deleting it. Type: Map of attribute names.	No
<i>Expected:AttributeName</i>	The name of the attribute for the conditional put. Type: String	No

Name	Description	Required
<i>Expected:AttributeName: ExpectedAttributeValue</i>	<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 <i>Expected</i> parameter and provide a <i>Value</i>, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify <i>{"Exists": true}</i>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"Color": {"Value": {"S": "Yellow"}}}</pre> <p>Note If you specify <i>{"Exists": true}</i> without an attribute value to check, DynamoDB returns an error.</p>	No
<i>ReturnValues</i>	<p>Use this parameter if you want to get the attribute name-value pairs before they were deleted. Possible parameter values are NONE (default) or ALL_OLD. If ALL_OLD is specified, the content of the old item is returned. If this parameter is not provided or is NONE, nothing is returned.</p> <p>Type: String</p>	No

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
```

```
content-length: 353
Date: Tue, 12 Jul 2011 21:31:03 GMT

{ "Attributes":
  { "AttributeName3": { "SS": [ "AttributeValue3" , "AttributeValue4" , "AttributeValue5" ] } ,
    "AttributeName2": { "S": "AttributeValue2" } ,
    "AttributeName1": { "N": "AttributeValue1" }
  },
  "ConsumedCapacityUnits":1
}
```

Name	Description
<i>Attributes</i>	If the <i>ReturnValues</i> parameter is provided as <i>ALL_OLD</i> in the request, DynamoDB returns an array of attribute name-value pairs (essentially, the deleted item). Otherwise, the response contains an empty set. Type: Array of attribute name-value pairs.
<i>ConsumedCapacityUnits</i>	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 Specifying Read and Write Requirements for Tables (p. 55) . Type: Number

Special Errors

Error	Description
ConditionalCheckFailedException	Conditional check failed. An expected attribute value was not found.

Examples

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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: U9809LI6BBFJA5N2R0TB0P017JVV4KQNSO5AEMVJF66Q9ASUAAJG
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 \(p. 695\)](#)

DeleteTable

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 \(p. 688\)](#) API to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.  

// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).  

POST / HTTP/1.1  

x-amz-target: DynamoDB_20111205.DeleteTable  

content-type: application/x-amz-json-1.0  

{ "TableName": "Table1"}
```

Name	Description	Required
<i>TableName</i>	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
<i>TableDescription</i>	A container for the table properties.
<i>CreationDateTime</i>	Date when the table was created. Type: Number

Name	Description
<i>KeySchema</i>	The primary key (simple or composite) structure for the table. A name-value pair for the <i>HashKeyElement</i> is required, and a name-value pair for the <i>RangeKeyElement</i> is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5) . Type: Map of <i>HashKeyElement</i> , or <i>HashKeyElement</i> and <i>RangeKeyElement</i> for a composite primary key.
<i>ProvisionedThroughput</i>	Throughput for the specified table, consisting of values for <i>ReadCapacityUnits</i> and <i>WriteCapacityUnits</i> . See Specifying Read and Write Requirements for Tables (p. 55) .
<i>ProvisionedThroughput:ReadCapacityUnits</i>	The minimum number of <i>ReadCapacityUnits</i> consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number
<i>ProvisionedThroughput:WriteCapacityUnits</i>	The minimum number of <i>WriteCapacityUnits</i> consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number
<i>TableName</i>	The name of the deleted table. Type: String
<i>TableStatus</i>	The current state of the table (<i>DELETING</i>). Once the table is deleted, subsequent requests for the table return <i>resource not found</i> . Use the DescribeTables (p. 688) API to check the status of the table. Type: String

Special Errors

Error	Description
ResourceInUseException	Table is in state <i>CREATING</i> or <i>UPDATING</i> and can't be deleted.

Examples

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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 \(p. 677\)](#)
- [DescribeTables \(p. 688\)](#)

DescribeTables

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 Sample DynamoDB JSON Request and Response \(p. 481\).
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{ "TableName": "Table1" }
```

Name	Description	Required
<i>TableName</i>	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
<i>Table</i>	Container for the table being described. Type: String
<i>CreationDateTime</i>	Date when the table was created in UNIX epoch time .
<i>ItemCount</i>	Number of items in the specified table. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
<i>KeySchema</i>	The primary key (simple or composite) structure for the table. A name-value pair for the <i>HashKeyElement</i> is required, and a name-value pair for the <i>RangeKeyElement</i> 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) .

Name	Description
<i>ProvisionedThroughput</i>	Throughput for the specified table, consisting of values for <i>LastIncreaseDateTime</i> (if applicable), <i>LastDecreaseDateTime</i> (if applicable), <i>ReadCapacityUnits</i> and <i>WriteCapacityUnits</i> . If the throughput for the table has never been increased or decreased, DynamoDB does not return values for those elements. See Specifying Read and Write Requirements for Tables (p. 55) . Type: Array
<i>TableName</i>	The name of the requested table. Type: String
<i>TableSizeBytes</i>	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
<i>TableStatus</i>	The current state of the table (<i>CREATING</i> , <i>ACTIVE</i> , <i>DELETING</i> or <i>UPDATING</i>). Once the table is in the <i>ACTIVE</i> state, you can add data.

Special Errors

No errors are specific to this API.

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 Sample DynamoDB JSON Request and Response \(p. 481\).
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 \(p. 677\)](#)
- [DeleteTable \(p. 685\)](#)
- [ListTables \(p. 693\)](#)

GetItem

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 [Data Read and Consistency Considerations \(p. 9\)](#).

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).  
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
<i>TableName</i>	The name of the table containing the requested item. Type: String	Yes
<i>Key</i>	The primary key values that define the item. For more information about primary keys, see Primary Key (p. 5) . Type: Map of <i>HashKeyElement</i> to its value and <i>RangeKeyElement</i> to its value.	Yes
<i>AttributesToGet</i>	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
<i>ConsistentRead</i>	If set to <code>true</code> , 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
<i>Item</i>	Contains the requested attributes. Type: Map of attribute name-value pairs.
<i>ConsumedCapacityUnits</i>	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 Specifying Read and Write Requirements for Tables (p. 55) . Type: Number

Special Errors

No errors specific to this API.

Examples

For examples using the AWS SDK, see [Working with Items in DynamoDB \(p. 85\)](#).

Sample Request

```
// This header is abbreviated.  
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).  
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 the previous API version (2011-12-05). For the most recent API version, go to 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 Sample DynamoDB JSON Request and Response \(p. 481\).  
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
<i>Limit</i>	A number of maximum table names to return. Type: Integer	No
<i>ExclusiveStartTableName</i>	The name of the table that starts the list. If you already ran a ListTables operation and received an <i>LastEvaluatedTableName</i> 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
<i>TableNames</i>	The names of the tables associated with the current account at the current endpoint. Type: Array

Name	Description
<i>LastEvaluatedTableName</i>	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 <i>ExclusiveStartTableName</i> 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 API.

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 Sample DynamoDB JSON Request and Response \(p. 481\).
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 \(p. 688\)](#)
- [CreateTable \(p. 677\)](#)
- [DeleteTable \(p. 685\)](#)

PutItem

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 this API, see [Working with Items in DynamoDB \(p. 85\)](#).

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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
<i>TableName</i>	The name of the table to contain the item. Type: String	Yes
<i>Item</i>	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) . Type: Map of attribute names to attribute values.	Yes

Name	Description	Required
<i>Expected</i>	<p>Designates an attribute for a conditional put. The <i>Expected</i> 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
<i>Expected:AttributeName</i>	<p>The name of the attribute for the conditional put.</p> <p>Type: String</p>	No
<i>Expected:AttributeName: ExpectedAttributeValue</i>	<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 <i>Expected</i> parameter and provide a <i>Value</i>, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify <i>{"Exists": true}</i>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"Color": {"Value": {"S": "Yellow" }}}}</pre> <p>Note If you specify <i>{"Exists": true}</i> without an attribute value to check, DynamoDB returns an error.</p>	No

Name	Description	Required
<i>ReturnValues</i>	<p>Use this parameter if you want to get the attribute name-value pairs before they were updated with the <i>PutItem</i> request. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>. If <code>ALL_OLD</code> is specified, and <i>PutItem</i> 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
<i>Attributes</i>	<p>Attribute values before the put operation, but only if the <i>ReturnValues</i> parameter is specified as <code>ALL_OLD</code> in the request.</p> <p>Type: Map of attribute name-value pairs.</p>
<i>ConsumedCapacityUnits</i>	<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 Specifying Read and Write Requirements for Tables (p. 55).</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 \(p. 85\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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 \(p. 714\)](#)
- [DeleteItem \(p. 681\)](#)
- [GetItem \(p. 691\)](#)
- [BatchGetItem \(p. 666\)](#)

Query

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 Sample DynamoDB JSON Request and Response \(p. 481\).  
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
<i>TableName</i>	The name of the table containing the requested items. Type: String	Yes

Name	Description	Required
<i>AttributesToGet</i>	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
<i>Limit</i>	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 <i>LastEvaluatedKey</i> 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 <i>LastEvaluatedKey</i> to apply in a subsequent operation to continue the query. Type: Number	No
<i>ConsistentRead</i>	If set to <code>true</code> , then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No
<i>Count</i>	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 <i>Limit</i> parameter to count-only queries. Do not set <i>Count</i> to true while providing a list of <i>AttributesToGet</i> ; otherwise, DynamoDB returns a validation error. For more information, see Count and ScannedCount (p. 186) . Type: Boolean	No
<i>HashKeyValue</i>	Attribute value of the hash component of the composite primary key. Type: String, Number, or Binary	Yes
<i>RangeKeyCondition</i>	A container for the attribute values and comparison operators to use for the query. A query request does not require a <i>RangeKeyCondition</i> . If you provide only the <i>HashKeyValue</i> , DynamoDB returns all items with the specified hash key element value. Type: Map	No
<i>RangeKeyCondition:</i> <i>AttributeValueList</i>	The attribute values to evaluate for the query parameters. The <i>AttributeValueList</i> contains one attribute value, unless a <i>BETWEEN</i> comparison is specified. For the <i>BETWEEN</i> comparison, the <i>AttributeValueList</i> contains two attribute values. Type: A map of <i>AttributeValue</i> to a <i>ComparisonOperator</i> .	No

Name	Description	Required
<i>RangeKeyCondition: ComparisonOperator</i>	<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 A. 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
<i>EQ</i>	<p><i>EQ</i> : Equal. For <i>EQ</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not equal { "NS" : ["6" , "2" , "1"] }.</p>	
<i>LE</i>	<p><i>LE</i> : Less than or equal. For <i>LE</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
<i>LT</i>	<p><i>LT</i> : Less than. For <i>LT</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
<i>GE</i>	<p><i>GE</i> : Greater than or equal. For <i>GE</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	

Name	Description	Required
	<p><i>GT</i> : Greater than.</p> <p>For <i>GT</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
	<p><i>BEGINS_WITH</i> : checks for a prefix.</p> <p>For <i>BEGINS_WITH</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> 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><i>BETWEEN</i> : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For <i>BETWEEN</i>, <i>AttributeValueList</i> must contain two <i>AttributeValue</i> 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 <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not compare to { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
<i>ScanIndexForward</i>	<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 Default is <code>true</code> (ascending).</p>	No
<i>ExclusiveStartKey</i>	<p>Primary key of the item from which to continue an earlier query. An earlier query might provide this value as the <i>LastEvaluatedKey</i> if that query operation was interrupted before completing the query; either because of the result set size or the <i>Limit</i> parameter. The <i>LastEvaluatedKey</i> can be passed back in a new query request to continue the operation from that point.</p> <p>Type: <i>HashKeyElement</i>, or <i>HashKeyElement</i> and <i>RangeKeyElement</i> for a composite primary key.</p>	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
```

```
content-type: application/x-amz-json-1.0
content-length: 308

{ "Count":2, "Items": [ {
    "AttributeName1": { "S": "AttributeValue1" },
    "AttributeName2": { "N": "AttributeValue2" },
    "AttributeName3": { "S": "AttributeValue3" }
}, {
    "AttributeName1": { "S": "AttributeValue3" },
    "AttributeName2": { "N": "AttributeValue4" },
    "AttributeName3": { "S": "AttributeValue3" },
    "AttributeName5": { "B": "dmFsdWU=" }
}], "LastEvaluatedKey": { "HashKeyElement": { "AttributeValue3": "S" },
    "RangeKeyElement": { "AttributeValue4": "N" } },
    "ConsumedCapacityUnits":1
}
```

Name	Description
Items	Item attributes meeting the query parameters. Type: Map of attribute names to and their data types and values.
Count	Number of items in the response. For more information, see Count and ScannedCount (p. 186) . Type: Number
<i>LastEvaluatedKey</i>	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. The <i>LastEvaluatedKey</i> is <i>null</i> when the entire query result set is complete (i.e. the operation processed the “last page”). Type: <i>HashKeyElement</i> , or <i>HashKeyElement</i> and <i>RangeKeyElement</i> for a composite primary key.
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 Specifying Read and Write Requirements for Tables (p. 55) . Type: Number

Special Errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see [Query and Scan Operations in DynamoDB \(p. 183\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).  
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 Sample DynamoDB JSON Request and Response \(p. 481\).  
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 \(p. 706\)](#)

Scan

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 Sample DynamoDB JSON Request and Response
```

```
sponse (p. 481).
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
<i>TableName</i>	The name of the table containing the requested items. Type: String	Yes
<i>AttributesToGet</i>	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
<i>Limit</i>	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 <i>LastEvaluatedKey</i> 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 <i>LastEvaluatedKey</i> to apply in a subsequent operation to continue the scan. For more information see Limit (p. 186) . Type: Number	No
<i>Count</i>	If set to <i>true</i> , 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. Do not set <i>Count</i> to <i>true</i> while providing a list of <i>AttributesToGet</i> , otherwise DynamoDB returns a validation error. For more information, see Count and ScannedCount (p. 186) . Type: Boolean	No

Name	Description	Required
<i>ScanFilter</i>	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. Type: A map of attribute names to values with comparison operators.	No
<i>ScanFilter:AttributeValueList</i>	The values and conditions to evaluate the scan results for the filter. Type: A map of <i>AttributeValue</i> to a <i>Condition</i> .	No
<i>ScanFilter:ComparisonOperator</i>	The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a scan operation. 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. Type: String or Binary	No
<i>EQ</i> : Equal.	For <i>EQ</i> , <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> 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"] }.	
<i>NE</i> : Not Equal.	For <i>NE</i> , <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> 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"] }.	
<i>LE</i> : Less than or equal.	For <i>LE</i> , <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.	

Name	Description	Required
	<p><i>LT</i> : Less than.</p> <p>For <i>LT</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
	<p><i>GE</i> : Greater than or equal.</p> <p>For <i>GE</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
	<p><i>GT</i> : Greater than.</p> <p>For <i>GT</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> of type String, Number, or Binary (not a set). If an item contains an <i>AttributeValue</i> of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not compare to { "NS" : ["6" , "2" , "1"] }.</p>	
	<i>NOT_NULL</i> : Attribute exists.	
	<i>NULL</i> : Attribute does not exist.	
	<p><i>CONTAINS</i> : checks for a subsequence, or value in a set.</p> <p>For <i>CONTAINS</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> 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><i>NOT_CONTAINS</i> : checks for absence of a subsequence, or absence of a value in a set.</p> <p>For <i>NOT_CONTAINS</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> 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><i>BEGINS_WITH</i> : checks for a prefix.</p> <p>For <i>BEGINS_WITH</i>, <i>AttributeValueList</i> can contain only one <i>AttributeValue</i> 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><i>IN</i> : checks for exact matches.</p> <p>For <i>IN</i>, <i>AttributeValueList</i> can contain more than one <i>AttributeValue</i> 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>	
	<p><i>BETWEEN</i> : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For <i>BETWEEN</i>, <i>AttributeValueList</i> must contain two <i>AttributeValue</i> 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 <i>AttributeValue</i> 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>	
<i>ExclusiveStartKey</i>	<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 <i>Limit</i> parameter. The <i>LastEvaluatedKey</i> can be passed back in a new scan request to continue the operation from that point.</p> <p>Type: <i>HashKeyElement</i>, or <i>HashKeyElement</i> and <i>RangeKeyElement</i> 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 Count and ScannedCount (p. 186) . Type: Number
ScannedCount	Number of items in the complete scan before any filters are applied. A high <i>ScannedCount</i> value with few, or no, <i>Count</i> results indicates an inefficient Scan operation. For more information, see Count and ScannedCount (p. 186) . Type: Number
<i>LastEvaluatedKey</i>	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 <i>LastEvaluatedKey</i> is <i>null</i> when the entire scan result set is complete (i.e. the operation processed the “last page”).

Name	Description
<i>ConsumedCapacityUnits</i>	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements for Tables (p. 55) . Type: Number

Special Errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see [Query and Scan Operations in DynamoDB \(p. 183\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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": "1970"}, "fans": {"SS": ["Linda", "Elton"]}, "name": {"S": "Elton John"}, "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 Sample DynamoDB JSON Request and Response \(p. 481\).  
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 Sample DynamoDB JSON Request and Response \(p. 481\).  
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" } , "RangeKeyEle
    ment":{ "N": "250" } },
    "ConsumedCapacityUnits":0.5
  "ScannedCount":2
}
```

Related Actions

- [Query \(p. 700\)](#)
- [BatchGetItem \(p. 666\)](#)

UpdateItem

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 API, see [Working with Items in DynamoDB \(p. 85\)](#).

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateItem  
content-type: application/x-amz-json-1.0  
  
{ "TableName" : "Table1" ,  
  "Key" :  
    { "HashKeyElement" : { "S" : "AttributeValue1" } ,
```

```

        "RangeKeyElement": {"N": "AttributeValue2"}},
        "AttributeUpdates": {"AttributeName3": {"Value": {"S": "AttributeValue3_New"}, "Action": "PUT"}},
        "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue3_Current"}}},
        "ReturnValues": "ReturnValuesConstant"
    }
}

```

Name	Description	Required
<i>TableName</i>	The name of the table containing the item to update. Type: String	Yes
<i>Key</i>	The primary key that defines the item. For more information about primary keys, see Primary Key (p. 5) . Type: Map of <i>HashKeyElement</i> to its value and <i>RangeKeyElement</i> to its value.	Yes
<i>AttributeUpdates</i>	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. Type: Map of attribute name, value, and an action for the attribute update.	
<i>AttributeUpdates:Action</i>	Specifies how to perform the update. Possible values: PUT (default), ADD or DELETE. The semantics are explained in the UpdateItem description. Type: String Default: PUT	No
<i>Expected</i>	Designates an attribute for a conditional update. The <i>Expected</i> 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. Type: Map of attribute names.	No
<i>Expected:AttributeName</i>	The name of the attribute for the conditional put. Type: String	No

Name	Description	Required
<i>Expected:AttributeName : ExpectedAttributeValue</i>	<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 <i>Expected</i> parameter and provide a <i>Value</i>, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify <i>{ "Exists": true }</i>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"Color": {"Value": {"S": "Yellow"}}}</pre> <p>Note If you specify <i>{ "Exists": true }</i> without an attribute value to check, DynamoDB returns an error.</p>	No
<i>ReturnValues</i>	<p>Use this parameter if you want to get the attribute name-value pairs before they were updated with the <i>UpdateItem</i> request. Possible parameter values are NONE (default) or ALL_OLD, UPDATED_OLD, ALL_NEW or UPDATED_NEW. If ALL_OLD is specified, and <i>UpdateItem</i> overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is NONE, nothing is returned. If ALL_NEW is specified, then all the attributes of the new version of the item are returned. If UPDATED_NEW 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
<i>Attributes</i>	A map of attribute name-value pairs, but only if the <i>ReturnValues</i> parameter is specified as something other than <i>NONE</i> in the request. Type: Map of attribute name-value pairs.
<i>ConsumedCapacityUnits</i>	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements for Tables (p. 55) . Type: Number

Special Errors

Error	Description
ConditionalCheckFailedException	Conditional check failed. Attribute ("+ name +") value is ("+ value +") but was expected ("+ expValue +")
ResourceNotFoundException	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see [Working with Items in DynamoDB \(p. 85\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
```

```
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 \(p. 695\)](#)
- [DeleteItem \(p. 681\)](#)

UpdateTable

Important

This section refers to the previous API version (2011-12-05). For the most recent API version, go to 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 [Specifying Read and Write Requirements for Tables \(p. 55\)](#).

The provisioned throughput values can be upgraded or downgraded based on the maximums and minimums listed in [Limits in DynamoDB \(p. 597\)](#).

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 Sample DynamoDB JSON Request and Response \(p. 481\).
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
<i>TableName</i>	The name of the table to update. Type: String	Yes
<i>ProvisionedThroughput</i>	New throughput for the specified table, consisting of values for <i>ReadCapacityUnits</i> and <i>WriteCapacityUnits</i> . See Specifying Read and Write Requirements for Tables (p. 55) . Type: Array	Yes
<i>ProvisionedThroughput :ReadCapacityUnits</i>	Sets the minimum number of consistent <i>ReadCapacityUnits</i> consumed per second for the specified table before DynamoDB balances the load with other operations. Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent <i>ReadCapacityUnits</i> per second provides 100 eventually consistent <i>ReadCapacityUnits</i> per second. Type: Number	Yes
<i>ProvisionedThroughput :WriteCapacityUnits</i>	Sets the minimum number of <i>WriteCapacityUnits</i> consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number	Yes

Responses

Syntax

```

HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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
<i>CreationDateTime</i>	Date when the table was created. Type: Number
<i>KeySchema</i>	The primary key (simple or composite) structure for the table. A name-value pair for the <i>HashKeyElement</i> is required, and a name-value pair for the <i>RangeKeyElement</i> 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 <i>HashKeyElement</i> , or <i>HashKeyElement</i> and <i>RangeKeyElement</i> for a composite primary key.
<i>ProvisionedThroughput</i>	Current throughput settings for the specified table, including values for <i>LastIncreaseDateTime</i> (if applicable), <i>LastDecreaseDateTime</i> (if applicable), Type: Array
<i>TableName</i>	The name of the updated table. Type: String

Name	Description
<i>TableStatus</i>	The current state of the table (<i>CREATING</i> , <i>ACTIVE</i> , <i>DELETING</i> or <i>UPDATING</i>), which should be <i>UPDATING</i> . Use the DescribeTables (p. 688) API to check the status of the table. Type: String

Special Errors

Error	Description
<i>ResourceNotFoundException</i>	The specified table was not found.
<i>ResourceInUseException</i>	The table is not in the <i>ACTIVE</i> state.

Examples

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see Sample DynamoDB JSON Request and Response \(p. 481\).
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 \(p. 677\)](#)
- [DescribeTables \(p. 688\)](#)
- [DeleteTable \(p. 685\)](#)

AWS Glossary

For the latest AWS terminology, see the [AWS Glossary](#) in the *AWS General Reference*.