Amazon DynamoDB

Developer Guide

API Version 2011-12-05

Amazon Web Services

Amazon DynamoDB: Developer Guide

Amazon Web Services

What is Amazon DynamoDB?

Topics

Welcome to the Amazon DynamoDB Developer Guide. Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. If you are a developer, you can use Amazon DynamoDB to create a database table that can store and retrieve any amount of data, and serve any level of request traffic. Amazon 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 launch a new Amazon 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 Amazon 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 Amazon DynamoDB, we recommend that you begin by reading the following sections:

- **What is Amazon 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 Amazon DynamoDB.
- **Getting Started with Amazon DynamoDB (p. 11)—**The Getting Started section walks you through the process of creating sample tables, uploading data, and performing some basic database operations.

Beyond the Getting Started section, you'll probably want to learn more about Amazon DynamoDB operations. The following sections provide detailed information about working with Amazon DynamoDB using the AWS Software Development Kits (SDKs) for Java, Microsoft .NET, and PHP:

- Working with Tables in Amazon DynamoDB (p. 64)
- Working with Items in Amazon DynamoDB (p. 102)
- Query and Scan in Amazon DynamoDB (p. 182)

The AWS SDKs for Java and .NET provide an object persistence model API that you can use to map your client-side classes to Amazon DynamoDB tables. The .NET SDK also provides a helper class to further simplify your development work. For more information, including working samples, see Using the AWS SDKs with Amazon DynamoDB (p. 259).

In addition to .NET, Java, and PHP, the other AWS SDKs also support Amazon DynamoDB, including Android, iOS, and Ruby. For links to the complete set of AWS SDKs, see Sample Code & Libraries.

Learn how you can use Amazon Elastic MapReduce to analyze datasets that are stored in DynamoDB and to archive the results in Amazon Simple Storage Service (Amazon S3), all while keeping the original dataset in DynamoDB intact. For more information, see Exporting, Importing, Querying, and Joining Tables in Amazon DynamoDB Using Amazon EMR (p. 230).

Monitor tables using Amazon CloudWatch metrics—Amazon DynamoDB displays key operational metrics for your table in the AWS Management Console. The service also integrates with Amazon CloudWatch, so you can see your request throughput and latency for each Amazon DynamoDB table and easily track your resource consumption. For more information, see Monitoring Amazon DynamoDB Tables with Amazon CloudWatch (p. 223).

Service Highlights for Amazon DynamoDB

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 launch a new Amazon 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. Amazon 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.

Amazon 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.

Amazon 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 Amazon DynamoDB features:

- **Scalable —** Amazon DynamoDB is designed for seamless throughput and storage scaling.
  - **Provisioned Throughput —** When creating a table, simply specify how much request 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 throughput requirements change, simply update your table request capacity using the AWS Management Console or the Amazon 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 —** Amazon DynamoDB scales horizontally and seamlessly scales a single table over hundreds of servers.
- **Fast, Predictable Performance—** Average service-side latencies for Amazon 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—** Amazon 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—** Amazon 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 —** Amazon DynamoDB does not have a fixed schema. Instead, each data item may have a different number of attributes. Multiple data types (strings, numbers, and sets) add richness to the data model.
- **Strong Consistency, Atomic Counters—** Unlike many non-relational databases, Amazon DynamoDB makes development easier by allowing you to use strong consistency on reads to ensure you are always reading the latest values. Amazon DynamoDB supports multiple native data types (numbers, strings, 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—** Amazon 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—** Amazon 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—** Amazon DynamoDB displays key operational metrics for your table in the AWS Management Console. The service also integrates with Amazon CloudWatch so you can see your request throughput and latency for each Amazon DynamoDB table, and easily track your resource consumption.
- **Elastic MapReduce Integration—** Amazon 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 Amazon 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. Amazon DynamoDB and Amazon RDS), do complex analysis over this combined dataset, and store the results of this work in Amazon S3.

Amazon DynamoDB Data Model

Topics

- Amazon DynamoDB Data Types (p. 6)

Data Model Concepts - Tables, Items, and Attributes

Amazon 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. Amazon DynamoDB is a NoSQL database. That is, except for the required primary key, an Amazon DynamoDB table is schema-less. Individual items in an Amazon DynamoDB table can have any number of attributes, although there is a limit of 64 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 catalog of products in Amazon DynamoDB. So you might 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 product 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"
}
```

Example items

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

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

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 Color multi-valued attribute. The Id is the only required attribute. Note that attribute values are shown using JSON-like syntax for illustration purposes.

Amazon 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. Amazon 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. Amazon DynamoDB builds an unordered hash index on this primary key attribute. In the preceding example, the ProductCatalog has Id as its primary key. It is the hash attribute.
- **Hash and Range Type Primary Key—**In this case, the primary key is made of two attributes. The first attributes is the hash attribute and the second one is the range attribute. Amazon DynamoDB builds an unordered hash index on the hash primary key attribute and a sorted range index on the range primary key attribute. For example, Amazon Web Services maintain several forums (see Discussion Forums). Each forum has many threads of discussion and each thread has many replies. You can potentially model this by creating the following three tables:

Table Name	Primary Key Type	Hash Attribute Name	Range Attribute Name
Forum (Name, ...)	Hash	Attribute Name: Name	-
Thread (ForumName, Subject, ...)	Hash and Range	Attribute Name: ForumName	Attribute Name: Subject
Reply (Id, ReplyDateTime, ...)	Hash and Range	Attribute Name: Id	Attribute Name: 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 the Subject is the range attribute.

The Reply table has reply Id as the hash attribute and the ReplyDateTime as the range attribute. The reply Id identifies the thread to which the reply belongs. When designing Amazon DynamoDB tables you have to take into account the fact that Amazon DynamoDB does not support cross-table joins. For example, 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 use these tables to illustrate the Amazon DynamoDB functionality. For information about these tables and sample data stored in these tables, see Example Tables and Data in Amazon DynamoDB (p. 445).

Amazon DynamoDB Data Types

Amazon DynamoDB supports the following data types:

- **Scalar data types—**Number and String.
- **Multi-valued types—**String Set and Number Set.

For example, in the ProductCatalog table, the Id is a Number type attribute and Authors is a String Set type attribute.

String

Strings are Unicode with UTF8 binary encoding. There is no 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 Amazon DynamoDB (p. 257). Also, the length of the attribute is constrained by the item size limit. For the limit, see, Limits in Amazon DynamoDB (p. 257).

String value comparison is used when returning ordered results in the Scan (p. 425) and Query (p. 417) API. Comparison is based on ASCII character code values. For example, "a" is greater that "A" , and "aa" is greater than "B". For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters.

Number

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

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

String and Number Sets

Amazon DynamoDB also supports both Number Sets and String Sets. Multi-valued attributes such as Authors attribute in a book item and Color attribute of a product item are examples of string set type attributes. Note that, because it is a set, the values in the set must be unique. String Sets and Number Sets are not ordered; the order of the values returned in a set is not preserved.

Supported Operations in Amazon DynamoDB

To work with tables and items (see Amazon DynamoDB Data Model (p. 3)), Amazon DynamoDB offers the following set of operations:

Table Operations

Amazon DynamoDB provides operations to create, update and delete tables. After the table is created, you can use the update operation to increase or decrease a table's provisioned throughput. Amazon DynamoDB also supports an operation to retrieve table information (the describe table operation) including the current status of the table, the primary key, and when the table was created. The list table operation enables you to get a list of tables in your account in the region of the endpoint you are using to communicate with Amazon DynamoDB. For more information, see Working with Tables in Amazon DynamoDB (p. 64).

Item Operations

Item operations enable you to add, update and delete items from a table. The update operation allows you to update existing attribute values, add new attributes, and delete existing attributes from an item. You can 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 greater than $20.

Amazon DynamoDB provides an operation to retrieve one item (get operation) or multiple items (batch get operation). You can use the batch get operation to retrieve items from multiple tables. For more information, see Working with Items in Amazon DynamoDB (p. 102).

Query and Scan

The query operation enables you to query a table using the hash attribute and an optional range filter. You can query only the tables whose primary key is of hash-and-range type. Query is the most efficient way to search for items from a table.

Amazon DynamoDB also supports a table scan operation. To get the best performance out of Amazon DynamoDB, you should design your tables so that you can use the query operation, mostly, and use scan only where appropriate. You can filter results of both the query and scan operations using filters. For more information, see Query and Scan in Amazon DynamoDB (p. 182).

Data Read and Consistency Considerations

Amazon DynamoDB maintains multiple copies of each item to ensure durability. When Amazon DynamoDB returns an operation successful response to your write request, Amazon DynamoDB ensures the write is durable on multiple servers. However, it takes time for the update to propagate to all copies. That is, the data is eventually consistent, meaning that your read request immediately after a write operation might not show the latest change. However, Amazon DynamoDB offers you the option to request the most up-to-date version of the data. To support varied application requirements, Amazon DynamoDB supports both eventually consistent and consistent read options:

Eventually consistent read

When you read data (get an item, batch get, query or scan operations), the response might not reflect the results of a recently completed write operation (put, update or a delete). That is, 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 get item operations perform an eventually consistent read, and you can request, optionally, a consistent read. The scan and batch get operations are always eventually consistent. For more information about the API, see API Reference for Amazon DynamoDB (p. 371).

Consistent read

When you issue a consistent read request, Amazon DynamoDB returns a response with the most up-to-date data that reflects updates by all prior related write operations to which Amazon DynamoDB returned a successful response. A 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 consistent read result by specifying optional parameters in your request.

Conditional Updates and Concurrency Control

In a multiuser environment, how do you ensure data updates made by one client don't overwrite updates made by another client? The "lost update" is a classic database concurrency issue. Suppose two clients read the same item. Both clients get a copy of that item from Amazon 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.

Amazon DynamoDB supports a "conditional write" feature that allows you to specify a condition when updating an item. Amazon 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.

Amazon 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. Amazon DynamoDB write operations support incrementing or decrementing existing attribute values.

For more information, see Working with Items in Amazon DynamoDB (p. 102).

Provisioned Throughput in Amazon DynamoDB

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

A unit of Write Capacity enables you to perform one write per second for items of up to 1KB in size. Similarly, a unit of Read Capacity enables you to perform one strongly consistent read per second (or two eventually consistent reads per second) of items of up to 1KB in size. Larger items will require more capacity. You can calculate the number of units of read and write capacity you need by estimating the number of reads or writes you need to do per second and multiplying by the size of your items (rounded up to the nearest KB).

```
Units of Capacity required for writes = Number of item writes per second x item
 size (rounded up to the nearest KB)
```

```
Units of Capacity required for reads* = Number of item reads per second x item
 size (rounded up to the nearest KB)
```

* If you use eventually consistent reads you'll get twice the throughput in terms of reads per second.

Note that,

- If your items are less than 1KB in size, then each unit of Read Capacity will give you 1 read/second of capacity and each unit of Write Capacity will give you 1 write/second of capacity. For example, if your items are 512 bytes and you need to read 100 items per second from your table, then you need to provision 100 units of Read Capacity.
- If your items are larger than 1KB in size, then you should calculate the number of units of Read Capacity and Write Capacity that you need. For example, if your items are 1.5KB and you want to do 100 reads/second, then you would need to provision 100 (read per second) x 2 (1.5KB rounded up to the nearest whole number) = 200 units of Read Capacity.

Note that the required number of units of Read Capacity is determined by the number of items being read per second, not the number of API calls. For example, if you need to read 500 items per second from your table, and if your items are 1KB or less, then you need 500 units of Read Capacity. It doesn't matter if you do 500 individual GetItem calls or 50 BatchGetItems calls that each return 10 items.

If your request throughput exceeds your provisioned capacity, it may be throttled. However, the AWS Management Console charts your provisioned and utilized throughput capacity, and lets you make changes easily in anticipation of traffic changes.

For more information about providing the provisioned throughput requirements for a table, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65).

Accessing Amazon 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 Amazon DynamoDB web service API. When using the API, you must write the necessary code to sign and authenticate your requests. For more information on the API, see API Reference for Amazon DynamoDB (p. 371).

Alternatively, you can simplify application development by using the AWS SDKs that wrap the DynamoDB API calls. You provide your credentials, and these libraries take care of authentication and request signing. For more information about using the AWS SDKs, see Using the AWS SDKs with Amazon DynamoDB (p. 259).

Amazon DynamoDB also provides a management console that you can use to create, update, and delete tables without writing any code. You can also use the management console to monitor the performance of your tables. With Amazon CloudWatch metrics in the console, you can monitor table throughput and other performance metrics. For more information, go to Amazon DynamoDB console.

Regions and Endpoints for Amazon DynamoDB

By default, the AWS SDKs and console for Amazon DynamoDB reference the US-East (Northern Virginia) Region. As Amazon 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 Amazon DynamoDB

Topics

The following video introduces you to some Amazon DynamoDB concepts and how to create and monitor a table.

Video: Getting Started with Amazon DynamoDB.

Step 1: Before You Begin with Amazon DynamoDB

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 Amazon 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 Amazon DynamoDB

1. Go to http://aws.amazon.com, and then click **Sign Up Now**.
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, you must determine the programming language that you want to use and download the appropriate AWS SDK for your development platform.

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

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://aws.amazon.com/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 AWS Toolkit for Visual Studio. The toolkit provides AWS Explorer for Visual Studio and project templates that you can use for development . Go to http://aws.amazon.com/sdkfornet and click **Download AWS .NET SDK**. By default, the installation script installs both the AWS SDK and the AWS 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 you 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://aws.amazon.com/sdkforphp and follow the instructions on that page to download the AWS SDK for PHP.

Step 2: Create Example Tables in Amazon DynamoDB

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

Use case 1: Product Catalog

Suppose you want to store product information in Amazon DynamoDB. Each product you store has its own set of properties, and accordingly, you need to store different information about each of these products. Amazon DynamoDB is a NoSQL database. That is, 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 (Id, ...)	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. In the next step, you will write a simple query to retrieve data from these tables.

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

Creating Tables

You can create tables using either the Amazon DynamoDB console or write code using the Amazon DynamoDB API. For this getting started, you will use the console to create the tables. These tables are also used in other sections of this guide.

Note

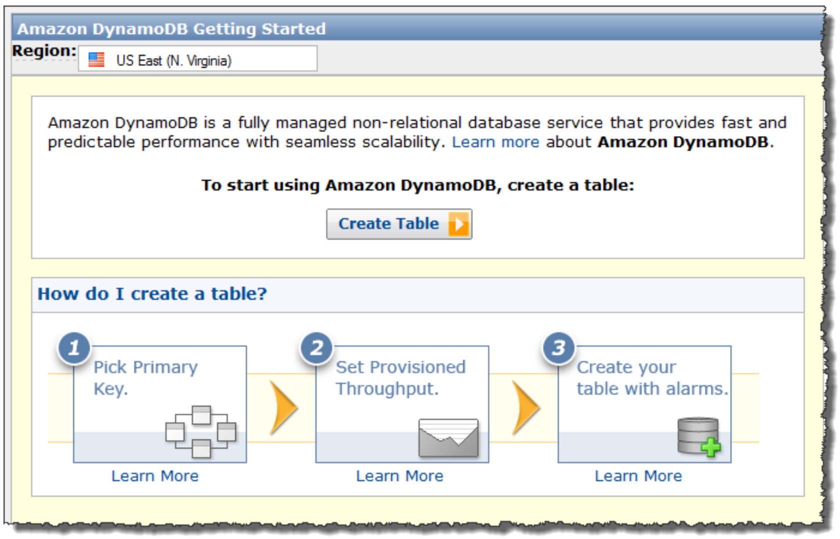
In these getting started steps, you use these tables to explore some of the basic Amazon DynamoDB operations. However, these tables are also used in other examples throughout this reference. If you delete these tables and later want to recreate them, you can repeat this getting started step, or programmatically 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 for Amazon DynamoDB (p. 450).

To Create the Sample Tables

Use the following procedure to create the four tables.

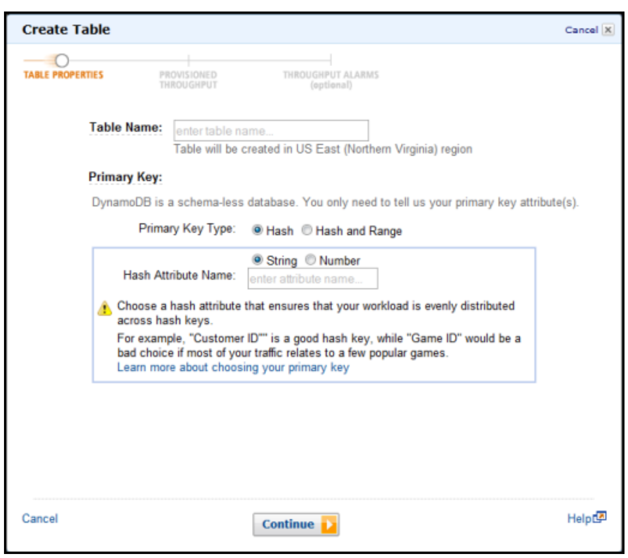
1. Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodb/.

   For first time users, the following wizard opens.

2. Click **Create Table**.

   The following **Create Table - TABLE PROPERTIES** wizard opens.

3. **Set the table name and its primary key**

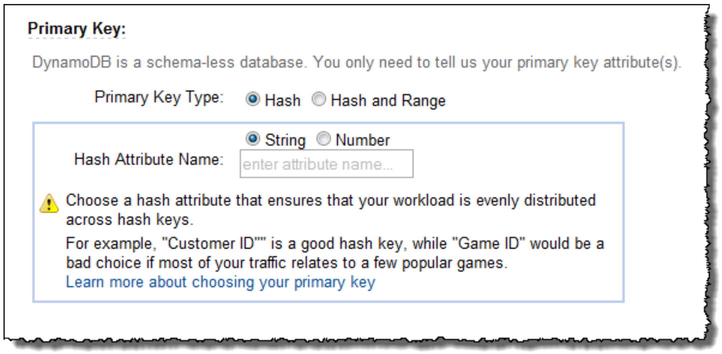
   a. Specify the table name in the **Table Name** field.

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

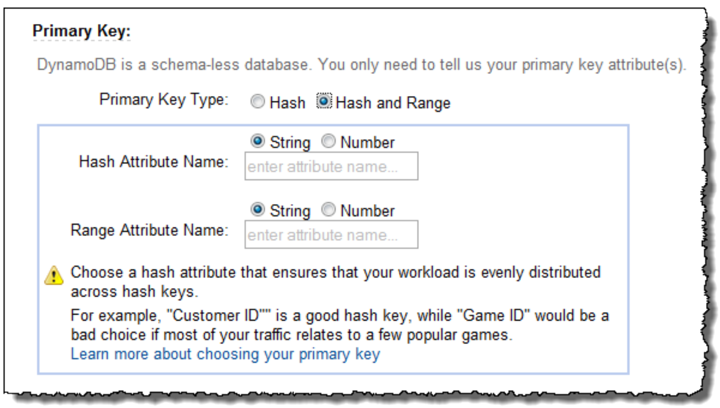
b. Select primary key type.

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 the Hash type, specify the attribute name and select the attribute type.



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

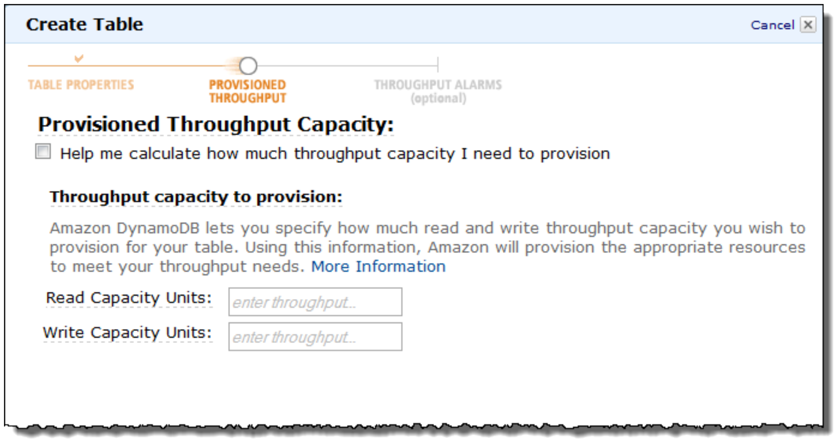


e. Click **Continue**.

4. **Specify the provisioned throughput**

   1. 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 (Provisioned Throughput) (p. 65). However, for the getting started exercise, you will set finite values.



2. 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 1 KB read operations and up to five 1 KB write operations per second. For more information, see Amazon DynamoDB Data Model (p. 3).

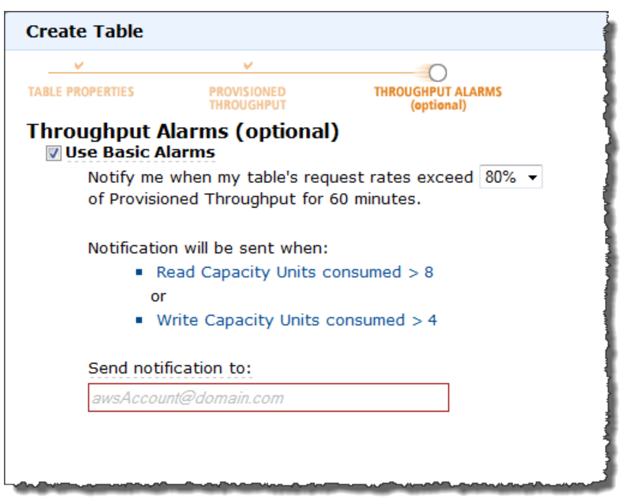
5. **Configure Amazon CloudWatch Alarms**

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

     This automatically configures Amazon 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 separated by commas.

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

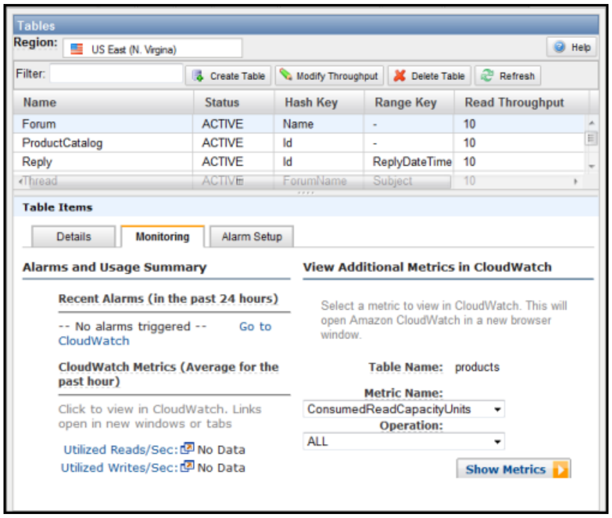
     For more information about Amazon CloudWatch alarms, see the Amazon CloudWatch Developer Guide.



6. Click **Create Table**.

Repeat the procedure to create the remaining tables.

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.



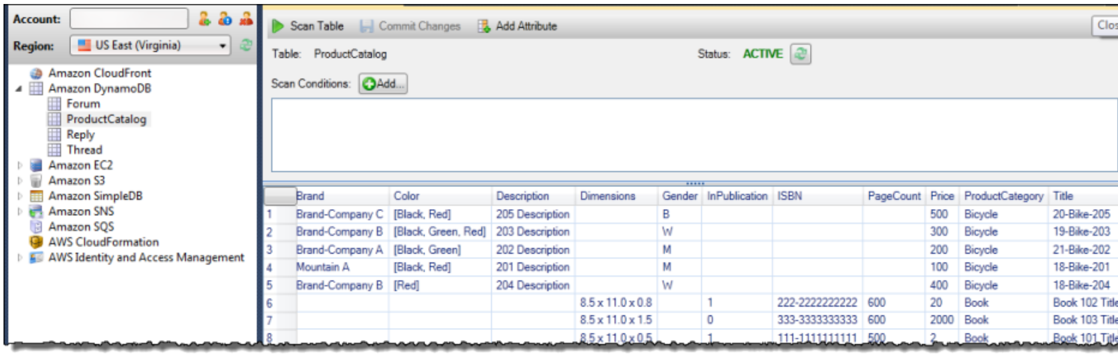
Step 3: Load Data into Tables in Amazon DynamoDB

Topics

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 Amazon DynamoDB.

Verify Data Load

If you are using Visual Studio or Eclipse, you can use the AWS Explorer to see all of your tables and data as shown in the following screen shot.



Load Data into Tables Using the AWS SDK for Java in Amazon DynamoDB

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 to run the sample, see Running Java Examples for Amazon DynamoDB (p. 260).

Example - Upload Sample Items Using the AWS SDK for Java

```
import java.text.SimpleDateFormat;

import java.util.Arrays;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;


import com.amazonaws.AmazonServiceException;

import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.model.AttributeValue;

import com.amazonaws.services.dynamodb.model.PutItemRequest;


public class AmazonDynamoDBSampleData_GettingStarted {


    static AmazonDynamoDBClient client;

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

    static String productCatalogTableName = "ProductCatalog";

    static String forumTableName = "Forum";

    static String threadTableName = "Thread";

    static String replyTableName = "Reply";


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

        createClient();


        try {


            uploadSampleProducts(productCatalogTableName);
```

```
            uploadSampleForums(forumTableName);

            uploadSampleThreads(threadTableName);

            uploadSampleReplies(replyTableName);


        } catch (AmazonServiceException ase) {

            System.err.println("Data load script failed.");

        }

    }


    private static void createClient() throws Exception {

        AWSCredentials credentials = new PropertiesCredentials(

                AmazonDynamoDBSampleData_GettingStarted.class.getResourceAs
Stream("AwsCredentials.properties"));


        client = new AmazonDynamoDBClient(credentials);

    }


    private static void uploadSampleProducts(String tableName) {


        try {

            // Add books.

            Map<String, AttributeValue> item = new HashMap<String, Attribute
Value>();

            item.put("Id", new AttributeValue().withN("101"));

            item.put("Title", new AttributeValue().withS("Book 101 Title"));

            item.put("ISBN", new AttributeValue().withS("111-1111111111"));

            item.put("Authors", new AttributeValue().withSS(Arrays.asList("Au
thor1")));

            item.put("Price", new AttributeValue().withN("2"));

            item.put("Dimensions", new AttributeValue().withS("8.5 x 11.0 x
0.5"));
```

```
            item.put("PageCount", new AttributeValue().withN("500"));

            item.put("InPublication", new AttributeValue().withN("1"));

            item.put("ProductCategory", new AttributeValue().withS("Book"));



           PutItemRequest itemRequest = new PutItemRequest().withTableName(table
Name).withItem(item);

            client.putItem(itemRequest);

            item.clear();



            item.put("Id", new AttributeValue().withN("102"));

            item.put("Title", new AttributeValue().withS("Book 102 Title"));

            item.put("ISBN", new AttributeValue().withS("222-2222222222"));

            item.put("Authors", new AttributeValue().withSS(Arrays.asList("Au
thor1", "Author2")));

            item.put("Price", new AttributeValue().withN("20"));

            item.put("Dimensions", new AttributeValue().withS("8.5 x 11.0 x
0.8"));

            item.put("PageCount", new AttributeValue().withN("600"));

            item.put("InPublication", new AttributeValue().withN("1"));

            item.put("ProductCategory", new AttributeValue().withS("Book"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();



            item.put("Id", new AttributeValue().withN("103"));

            item.put("Title", new AttributeValue().withS("Book 103 Title"));

            item.put("ISBN", new AttributeValue().withS("333-3333333333"));

            item.put("Authors", new AttributeValue().withSS(Arrays.asList("Au
thor1", "Author2")));
```

```
            // Intentional. Later we run scan to find price error. Find items
> 1000 in price.

            item.put("Price", new AttributeValue().withN("2000"));

            item.put("Dimensions", new AttributeValue().withS("8.5 x 11.0 x
1.5"));

            item.put("PageCount", new AttributeValue().withN("600"));

            item.put("InPublication", new AttributeValue().withN("0"));

            item.put("ProductCategory", new AttributeValue().withS("Book"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();



            // Add bikes.

            item.put("Id", new AttributeValue().withN("201"));

            item.put("Title", new AttributeValue().withS("18-Bike-201")); //
Size, followed by some title.

            item.put("Description", new AttributeValue().withS("201 Descrip
tion"));

            item.put("BicycleType", new AttributeValue().withS("Road"));

            item.put("Brand", new AttributeValue().withS("Mountain A")); //
Trek, Specialized.

            item.put("Price", new AttributeValue().withN("100"));

            item.put("Gender", new AttributeValue().withS("M")); // Men's

            item.put("Color", new AttributeValue().withSS(Arrays.asList("Red",
 "Black")));

           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();
```

```
            item.put("Id", new AttributeValue().withN("202"));

            item.put("Title", new AttributeValue().withS("21-Bike-202"));

            item.put("Description", new AttributeValue().withS("202 Descrip
tion"));

            item.put("BicycleType", new AttributeValue().withS("Road"));

            item.put("Brand", new AttributeValue().withS("Brand-Company A"));

            item.put("Price", new AttributeValue().withN("200"));

            item.put("Gender", new AttributeValue().withS("M"));

           item.put("Color", new AttributeValue().withSS(Arrays.asList("Green",
 "Black")));

           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();


            item.put("Id", new AttributeValue().withN("203"));

            item.put("Title", new AttributeValue().withS("19-Bike-203"));

            item.put("Description", new AttributeValue().withS("203 Descrip
tion"));

            item.put("BicycleType", new AttributeValue().withS("Road"));

            item.put("Brand", new AttributeValue().withS("Brand-Company B"));

            item.put("Price", new AttributeValue().withN("300"));

            item.put("Gender", new AttributeValue().withS("W")); // Women's

            item.put("Color", new AttributeValue().withSS(Arrays.asList("Red",
 "Green", "Black")));

           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));
```

```
            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);
            client.putItem(itemRequest);
            item.clear();


            item.put("Id", new AttributeValue().withN("204"));
            item.put("Title", new AttributeValue().withS("18-Bike-204"));
            item.put("Description", new AttributeValue().withS("204 Descrip
tion"));
            item.put("BicycleType", new AttributeValue().withS("Mountain"));
            item.put("Brand", new AttributeValue().withS("Brand-Company B"));
            item.put("Price", new AttributeValue().withN("400"));
            item.put("Gender", new AttributeValue().withS("W"));
           item.put("Color", new AttributeValue().withSS(Arrays.asList("Red")));

           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);
            client.putItem(itemRequest);
            item.clear();


            item.put("Id", new AttributeValue().withN("205"));
            item.put("Title", new AttributeValue().withS("20-Bike-205"));
            item.put("Description", new AttributeValue().withS("205 Descrip
tion"));
            item.put("BicycleType", new AttributeValue().withS("Hybrid"));
            item.put("Brand", new AttributeValue().withS("Brand-Company C"));
            item.put("Price", new AttributeValue().withN("500"));
            item.put("Gender", new AttributeValue().withS("B")); // Boy's
            item.put("Color", new AttributeValue().withSS(Arrays.asList("Red",
 "Black")));
```

```
           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);




        }   catch (AmazonServiceException ase) {

            System.err.println("Failed to create item in " + tableName);

        }


    }


    private static void uploadSampleForums(String tableName) {

        try {

            // Add forums.

            Map<String, AttributeValue> forum = new HashMap<String, Attribute
Value>();

            forum.put("Name", new AttributeValue().withS("Amazon DynamoDB"));

            forum.put("Category", new AttributeValue().withS("Amazon Web Ser
vices"));

            forum.put("Threads", new AttributeValue().withN("2"));

            forum.put("Messages", new AttributeValue().withN("4"));

            forum.put("Views", new AttributeValue().withN("1000"));



            PutItemRequest forumRequest = new PutItemRequest().withTable
Name(tableName).withItem(forum);

            client.putItem(forumRequest);

            forum.clear();
```

```
            forum.put("Name", new AttributeValue().withS("Amazon S3"));
            forum.put("Category", new AttributeValue().withS("Amazon Web Ser
vices"));
            forum.put("Threads", new AttributeValue().withN("0"));


            forumRequest = new PutItemRequest().withTableName(tableName).with
Item(forum);
            client.putItem(forumRequest);


        }   catch (AmazonServiceException ase) {
            System.err.println("Failed to create item in " + tableName);
        }
    }


    private static void uploadSampleThreads(String tableName) {
        try {
            long time1 = (new Date()).getTime() - (7L*24L*60L*60L*1000L); // 7
 days ago
            long time2 = (new Date()).getTime() - (14L*24L*60L*60L*1000L); //
14 days ago
            long time3 = (new Date()).getTime() - (21L*24L*60L*60L*1000L); //
21 days ago


            Date date1 = new Date();
            date1.setTime(time1);


            Date date2 = new Date();
            date2.setTime(time2);


            Date date3 = new Date();
            date3.setTime(time3);
```

```
            // Add threads.

            Map<String, AttributeValue> forum = new HashMap<String, Attribute
Value>();

            forum.put("ForumName", new AttributeValue().withS("Amazon Dy
namoDB"));

            forum.put("Subject", new AttributeValue().withS("DynamoDB Thread
1"));

            forum.put("Message", new AttributeValue().withS("DynamoDB thread 1
 message"));

            forum.put("LastPostedBy", new AttributeValue().withS("User A"));

            forum.put("LastPostedDateTime", new AttributeValue().withS(date
Formatter.format(date2)));

            forum.put("Views", new AttributeValue().withN("0"));

            forum.put("Replies", new AttributeValue().withN("0"));

            forum.put("Answered", new AttributeValue().withN("0"));

          forum.put("Tags", new AttributeValue().withSS(Arrays.asList("index",
 "primarykey", "table")));



            PutItemRequest forumRequest = new PutItemRequest().withTable
Name(tableName).withItem(forum);

            client.putItem(forumRequest);



            forum.clear();



            forum.put("ForumName", new AttributeValue().withS("Amazon Dy
namoDB"));

            forum.put("Subject", new AttributeValue().withS("DynamoDB Thread
2"));

            forum.put("Message", new AttributeValue().withS("DynamoDB thread 2
 message"));

            forum.put("LastPostedBy", new AttributeValue().withS("User A"));

            forum.put("LastPostedDateTime", new AttributeValue().withS(date
Formatter.format(date3)));

            forum.put("Views", new AttributeValue().withN("0"));
```

```
            forum.put("Replies", new AttributeValue().withN("0"));

            forum.put("Answered", new AttributeValue().withN("0"));

          forum.put("Tags", new AttributeValue().withSS(Arrays.asList("index",
 "primarykey", "rangekey")));



            forumRequest = new PutItemRequest().withTableName(tableName).with
Item(forum);

            client.putItem(forumRequest);



            forum.clear();



            forum.put("ForumName", new AttributeValue().withS("Amazon S3"));

            forum.put("Subject", new AttributeValue().withS("S3 Thread 1"));

            forum.put("Message", new AttributeValue().withS("S3 Thread 3 mes
sage"));

            forum.put("LastPostedBy", new AttributeValue().withS("User A"));

            forum.put("LastPostedDateTime", new AttributeValue().withS(date
Formatter.format(date1)));

            forum.put("Views", new AttributeValue().withN("0"));

            forum.put("Replies", new AttributeValue().withN("0"));

            forum.put("Answered", new AttributeValue().withN("0"));

          forum.put("Tags", new AttributeValue().withSS(Arrays.asList("largeo
bjects", "multipart upload")));



            forumRequest = new PutItemRequest().withTableName(tableName).with
Item(forum);



            client.putItem(forumRequest);





        }   catch (AmazonServiceException ase) {

            System.err.println("Failed to create item in " + tableName);
```

```
        }


    }


    private static void uploadSampleReplies(String tableName) {

        try {

            long time0 = (new Date()).getTime() - (1L*24L*60L*60L*1000L); // 1
 day ago

            long time1 = (new Date()).getTime() - (7L*24L*60L*60L*1000L); // 7
 days ago

            long time2 = (new Date()).getTime() - (14L*24L*60L*60L*1000L); //
14 days ago

            long time3 = (new Date()).getTime() - (21L*24L*60L*60L*1000L); //
21 days ago


            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);


            // Add threads.

            Map<String, AttributeValue> reply = new HashMap<String, Attribute
Value>();

           reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 1"));
```

```
            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date3)));

            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 1
 Reply 1 text"));

            reply.put("PostedBy", new AttributeValue().withS("User A"));



            PutItemRequest replyRequest = new PutItemRequest().withTable
Name(tableName).withItem(reply);

            client.putItem(replyRequest);



            reply.clear();



            reply = new HashMap<String, AttributeValue>();

           reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 1"));

            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date2)));

            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 1
 Reply 2 text"));

            reply.put("PostedBy", new AttributeValue().withS("User B"));



            replyRequest = new PutItemRequest().withTableName(tableName).with
Item(reply);

            client.putItem(replyRequest);



            reply.clear();



            reply = new HashMap<String, AttributeValue>();

           reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 2"));

            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date1)));

            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 2
 Reply 1 text"));
```

```
            reply.put("PostedBy", new AttributeValue().withS("User A"));


            replyRequest = new PutItemRequest().withTableName(tableName).with
Item(reply);

            client.putItem(replyRequest);


            reply.clear();


            reply = new HashMap<String, AttributeValue>();

          reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 2"));

            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date0)));

            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 2
 Reply 2 text"));

            reply.put("PostedBy", new AttributeValue().withS("User A"));


            replyRequest = new PutItemRequest().withTableName(tableName).with
Item(reply);


            client.putItem(replyRequest);


        }   catch (AmazonServiceException ase) {

            System.err.println("Failed to create item in " + tableName);

        }

    }

}
```

Load Data into Tables Using the AWS SDK for.NET in Amazon DynamoDB

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 helper API to upload sample data. For step-by-step instructions to run the sample, see Running .NET Examples for Amazon DynamoDB (p. 306)

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

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

namespace amazon.dynamodb.documentation
{
  class Program
  {
    private static AmazonDynamoDBClient client;

    static void Main(string[] args)
    {
      try
      {
        client = new AmazonDynamoDBClient();

        // Upload data (using the .NET SDK helper API to upload data)
        UploadSampleProducts();
        UploadSampleForums();
        UploadSampleThreads();
        UploadSampleReplies();

        Console.WriteLine("Data uploaded... 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 UploadSampleProducts()
    {
      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 UploadSampleForums()
{
  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 UploadSampleThreads()

    {

      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 UploadSampleReplies()

    {

      Table replyTable = Table.LoadTable(client, "Reply");


      // Reply 1 - thead 1.

      var thread1Reply1 = new Document();

     thread1Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash attrib
ute.

     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 attrib
ute.

     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 attrib
ute.
     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 attrib
ute.
     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 attrib
ute.
     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 in Amazon DynamoDB

Note

This topic assumes that you are already following the instructions for Getting Started with Amazon DynamoDB (p. 11) and have the AWS SDK for PHP properly installed. Review the instructions in Running PHP Examples for Amazon DynamoDB (p. 369) if you need more information about setting up the SDK.

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 (to keep it in the free usage tier). You might see ProvisionedThroughputExceeded errors in the response from Amazon DynamoDB. However, the AWS SDKs retry requests for this error, and eventually all of the data is committed to the table.

```
<?php

// If necessary, reference the sdk.class.php file.

// For example, the following line assumes the sdk.class.php file is

// in an sdk sub-directory relative to this file

require_once dirname(__FILE__) . '/sdk/sdk.class.php';


// Instantiate the class

$dynamodb = new AmazonDynamoDB();


####################################################################

# Setup some local variables for dates


$one_day_ago = date('Y-m-d H:i:s', strtotime("-1 days"));

$seven_days_ago = date('Y-m-d H:i:s', strtotime("-7 days"));

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

$twenty_one_days_ago = date('Y-m-d H:i:s', strtotime("-21 days"));


####################################################################

# Adding data to the table


echo PHP_EOL . PHP_EOL;

echo "# Adding data to the table..." . PHP_EOL;


# Adding data to the table

echo "# Adding data to the table..." . PHP_EOL;
```

```
// Set up batch requests

$queue = new CFBatchRequest();

$queue->use_credentials($dynamodb->credentials);



// Add items to the batch

$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'101'              ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Book 101 Title'   ),

        'ISBN'            => array( AmazonDynamoDB::TYPE_STRING           =>
'111-1111111111'   ),

        'Authors'         => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Author1')   ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'2'                ),

        'Dimensions'      => array( AmazonDynamoDB::TYPE_STRING           =>
'8.5 x 11.0 x 0.5' ),

        'PageCount'       => array( AmazonDynamoDB::TYPE_NUMBER           =>
'500'              ),

        'InPublication'   => array( AmazonDynamoDB::TYPE_NUMBER           =>
'1'                ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Book'             )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
```

```
'102'                       ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Book 102 Title'            ),

        'ISBN'            => array( AmazonDynamoDB::TYPE_STRING           =>
'222-2222222222'            ),

        'Authors'         => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Author1', 'Author2') ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'20'                        ),

        'Dimensions'      => array( AmazonDynamoDB::TYPE_STRING           =>
'8.5 x 11.0 x 0.8'          ),

        'PageCount'       => array( AmazonDynamoDB::TYPE_NUMBER           =>
'600'                       ),

        'InPublication'   => array( AmazonDynamoDB::TYPE_NUMBER           =>
'1'                         ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Book'                      )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'103'                       ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Book 103 Title'            ),

        'ISBN'            => array( AmazonDynamoDB::TYPE_STRING           =>
'333-3333333333'            ),

        'Authors'         => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Author1', 'Author2') ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'2000'                      ),

        'Dimensions'      => array( AmazonDynamoDB::TYPE_STRING           =>
'8.5 x 11.0 x 1.5'          ),

        'PageCount'       => array( AmazonDynamoDB::TYPE_NUMBER           =>
'600'                       ),
```

```
        'InPublication'   => array( AmazonDynamoDB::TYPE_NUMBER           =>
'0'                          ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Book'                       )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'201'                 ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'18-Bike-201'         ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'201 Description'     ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Road'                ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Mountain A'          ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'100'                 ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'M'                   ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'             )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(
```

```
        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'202'                    ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'21-Bike-202'            ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'202 Description'        ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Road'                   ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company A'        ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'200'                    ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'M'                      ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Green', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'                )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'203'                           ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'19-Bike-203'                   ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'203 Description'               ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Road'                          ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company B'               ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'300'                           ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
```

```
'W'                            ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red', 'Green', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'                      )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'204'            ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'18-Bike-204'     ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'204 Description' ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Mountain'        ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company B' ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'400'             ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'W'               ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red')      ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'         )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(
```

```
        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'205'                 ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'20-Bike-205'         ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'205 Description'     ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Hybrid'              ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company C'     ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'500'                 ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'B'                   ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'             )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Forum',

    'Item' => array(

        'Name'     => array( AmazonDynamoDB::TYPE_STRING => 'Amazon DynamoDB'
    ), // Hash Key

       'Category' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Web Services'
 ),

        'Threads'  => array( AmazonDynamoDB::TYPE_NUMBER => '0'
    ),

        'Messages' => array( AmazonDynamoDB::TYPE_NUMBER => '0'
    ),

        'Views'    => array( AmazonDynamoDB::TYPE_NUMBER => '1000'
    ),

    )

));
```

```
$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Forum',

    'Item' => array(

        'Name'     => array( AmazonDynamoDB::TYPE_STRING => 'Amazon S3'
    ), // Hash Key

       'Category' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Web Services'
 ),

        'Threads'  => array( AmazonDynamoDB::TYPE_NUMBER => '0'
    )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Reply',

    'Item' => array(

        'Id'            => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 1' ), // Hash Key

        'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING => $four
teen_days_ago                 ), // Range Key

        'Message'       => array( AmazonDynamoDB::TYPE_STRING => 'DynamoDB
Thread 1 Reply 2 text'    ),

        'PostedBy'      => array( AmazonDynamoDB::TYPE_STRING => 'User B'
                       ),

    )

));

$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Reply',

    'Item' => array(

        'Id'            => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 2' ), // Hash Key

        'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING =>
$twenty_one_days_ago                    ), // Range Key
```

```
        'Message'        => array( AmazonDynamoDB::TYPE_STRING  => 'DynamoDB
Thread 2 Reply 3 text'    ),

        'PostedBy'       => array( AmazonDynamoDB::TYPE_STRING => 'User B'
                       ),

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Reply',

    'Item' => array(

        'Id'             => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 2' ), // Hash Key

       'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING => $seven_days_ago
                    ), // Range Key

        'Message'        => array( AmazonDynamoDB::TYPE_STRING  => 'DynamoDB
Thread 2 Reply 2 text'    ),

        'PostedBy'       => array( AmazonDynamoDB::TYPE_STRING => 'User A'
                       ),

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Reply',

    'Item' => array(

        'Id'             => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 2' ), // Hash Key

        'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING => $one_day_ago
                      ), // Range Key

        'Message'        => array( AmazonDynamoDB::TYPE_STRING  => 'DynamoDB
Thread 2 Reply 1 text'    ),

        'PostedBy'       => array( AmazonDynamoDB::TYPE_STRING => 'User A'
                       ),

    )

));
```

```
// Execute the batch of requests in parallel
$responses = $dynamodb->batch($queue)->send();


// Check for success...
if ($responses->areOK())
{
    echo "The data has been added to the table." . PHP_EOL;
}
    else
{
    print_r($responses);
}
?>
```

If you run this in a browser, the browser should display information indicating a successful operation, including a 200 response code and the `ConsumedCapacityUnits`.

Step 4: Try a Query in Amazon DynamoDB

Topics

In this step, you will try a simple query against the tables that you created in the preceding step.

Try a Query Using the AWS SDK for Java in Amazon DynamoDB

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 `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 to test the following code example, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.io.IOException;

import java.text.SimpleDateFormat;

import java.util.Arrays;

import java.util.Date;

import java.util.Map;


import com.amazonaws.AmazonServiceException;

import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.model.AttributeValue;

import com.amazonaws.services.dynamodb.model.ComparisonOperator;

import com.amazonaws.services.dynamodb.model.Condition;

import com.amazonaws.services.dynamodb.model.GetItemRequest;

import com.amazonaws.services.dynamodb.model.GetItemResult;

import com.amazonaws.services.dynamodb.model.Key;

import com.amazonaws.services.dynamodb.model.QueryRequest;

import com.amazonaws.services.dynamodb.model.QueryResult;


public class AmazonDynamoDBSampleData_TryQuery {


    static AmazonDynamoDBClient client;


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

        try {
```

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

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

        // Query replies posted in the past 15 days for a forum thread.
        findRepliesInLast15DaysWithConfig("Reply", forumName, threadSubject);
        }
        catch (AmazonServiceException ase) {
            System.err.println(ase.getMessage());
        }
    }

    private static void createClient() throws IOException {

        AWSCredentials credentials = new PropertiesCredentials(
                AmazonDynamoDBSampleData_TryQuery.class.getResourceAs
Stream("AwsCredentials.properties"));

        client = new AmazonDynamoDBClient(credentials);
    }

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

            GetItemRequest getItemRequest = new GetItemRequest()
                .withTableName(tableName)
                .withKey(new Key()
```

```
                    .withHashKeyElement(new AttributeValue().withN(id)))
                .withAttributesToGet(Arrays.asList("Id", "ISBN", "Title", "Au
thors"));


            GetItemResult result = client.getItem(getItemRequest);


            // Check the response.
            System.out.println("Printing item after retrieving it....");
            printItem(result.getItem());
    }


    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);


        Key lastKeyEvaluated = null;
        do {


            Condition rangeKeyCondition = new Condition()
            .withComparisonOperator(ComparisonOperator.GT.toString())
           .withAttributeValueList(new AttributeValue().withS(twoWeeksAgoStr));
```

```
            QueryRequest queryRequest = new QueryRequest().withTableName(table
Name)
             .withHashKeyValue(new AttributeValue().withS(replyId))
             .withRangeKeyCondition(rangeKeyCondition)
             .withAttributesToGet(Arrays.asList("Message", "ReplyDateTime",
"PostedBy"))
             .withLimit(1).withExclusiveStartKey(lastKeyEvaluated);


            QueryResult result = client.query(queryRequest);
             for (Map<String, AttributeValue> item : result.getItems()) {
                 printItem(item);
             }
             lastKeyEvaluated = result.getLastEvaluatedKey();
        } while (lastKeyEvaluated != null);
    }


    private static void printItem(Map<String, AttributeValue> attributeList) {

        for (Map.Entry<String, AttributeValue> item : attributeList.entrySet())
 {
             String attributeName = item.getKey();
             AttributeValue value = item.getValue();
             System.out.println(attributeName
                     + " "
                     + (value.getS() == null ? "" : "S=[" + value.getS() + "]")

                     + (value.getN() == null ? "" : "N=[" + value.getN() + "]")

                     + (value.getSS() == null ? "" : "SS=[" + value.getSS() +
"]")
                     + (value.getNS() == null ? "" : "NS=[" + value.getNS() +
"] \n"));
        }
```

```
    }
}
```

Try a Query Using the AWS SDK for .NET in Amazon DynamoDB

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 to test the following code example, see Running .NET Examples for Amazon DynamoDB (p. 306).

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDB.Model;
using Amazon.Runtime;
using Amazon.Util;


namespace Amazon.DynamoDB.Documentation
{
  class Program
  {
    private static AmazonDynamoDBClient client;


    static void Main(string[] args)
    {
      try
      {
        client = new AmazonDynamoDBClient();
```

```
        // 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 Key { HashKeyElement = new AttributeValue { N = id.ToString()
} }
      };
      var response = client.GetItem(request);


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

```
                        response.GetItemResult.ConsumedCapacityUnits);


      PrintItem(response.GetItemResult.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);


      Key lastKeyEvaluated = null;

      do

      {

        var request = new QueryRequest

        {

          TableName = "Reply",

          HashKeyValue = new AttributeValue { S = replyId },

          RangeKeyCondition = new Condition

          {

            ComparisonOperator = "GT",

            AttributeValueList = new List<AttributeValue>()

          {
```

```
            new AttributeValue { S = twoWeeksAgoString }

          }

          },



          // Optional parameter.

         AttributesToGet = new List<string> { "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,

        };



        var response = client.Query(request);



       Console.WriteLine("No. of reads used (by query in FindRepliesForAThread
SpecifyLimit) {0}\n",

                          response.QueryResult.ConsumedCapacityUnits);

        foreach (Dictionary<string, AttributeValue> item

          in response.QueryResult.Items)

        {

          PrintItem(item);

        }

        lastKeyEvaluated = response.QueryResult.LastEvaluatedKey;



      } while (lastKeyEvaluated != null);

      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.ToAr
ray()) + "]") +
            (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToAr
ray()) + "]")
        );
      }
      Console.WriteLine("************************************************");
    }
  }
}
```

Try a Query Using the AWS SDK for PHP in Amazon DynamoDB

Note

This topic assumes you are already following the instructions for Getting Started with Amazon DynamoDB (p. 11) and have the AWS SDK for PHP properly installed. Review the instructions in Running PHP Examples for Amazon DynamoDB (p. 369) if you need more information about setting up the SDK.

Example - Query for Items in your Amazon 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

// If necessary, reference the sdk.class.php file.

// For example, the following line assumes the sdk.class.php file is

// in an sdk sub-directory relative to this file

require_once dirname(__FILE__) . '/sdk/sdk.class.php';


// Instantiate the class.

$dynamodb = new AmazonDynamoDB();


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


$response = $dynamodb->query(array(

    'TableName' => 'Reply',

    'HashKeyValue' => array(

        AmazonDynamoDB::TYPE_STRING => 'Amazon DynamoDB#DynamoDB Thread 2',

    ),

    'RangeKeyCondition' => array(

       'ComparisonOperator' => AmazonDynamoDB::CONDITION_GREATER_THAN_OR_EQUAL,

        'AttributeValueList' => array(

            array(

                AmazonDynamoDB::TYPE_STRING => $fourteen_days_ago

            )

        )

    )

));
```

```
// Response code 200 indicates success
 print_r($response);


?>
```

Step 5: Delete Example Tables in Amazon DynamoDB

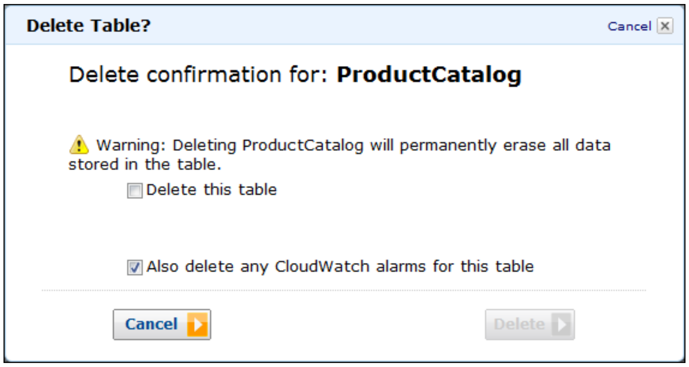
These tables are also used in various sections of this developer guide to illustrate table and item operations using various AWS SDKs. Don't delete these tables if you are reading the rest of the developer guide. However, if you don't plan to use these tables, you should delete them to avoid getting charged for resources you don't use.

You can also delete tables programmatically. For more information, see Working with Tables in Amazon DynamoDB (p. 64).

To Delete the Sample Tables

1. Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Select the table that you want to delete.
3. Click **Delete Table** in the Tables wizard.

   The following **Delete Table** wizard opens.



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

   This deletes the table from Amazon DynamoDB and the Amazon CloudWatch alarms configured for the table.

Where Do I Go from Here?

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

Working with Tables in Amazon DynamoDB

Topics

When creating a table, 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 such as the table name, primary key, list of its column names and their data types. All records stored in the table must have same set of columns. Amazon DynamoDB is a NoSQL database. That is, except for the required primary key, an Amazon DynamoDB table is schema-less. Individual items in an Amazon DynamoDB table can have any number of attributes, although there is a limit of 64 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. Amazon DynamoDB supports the following two types of primary keys:

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

Specifying Read and Write Requirements (Provisioned Throughput)

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

To ensure high availability and low latency responses, Amazon DynamoDB requires that you specify your required read and write throughput values when you create a table. Amazon 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 Amazon DynamoDB console or the API.

Amazon DynamoDB allocates and reserves resources to handle your throughput requirements with sustained low latency and you pay for the hourly reservation of these resource. 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 update table request to increase or decrease the provisioned throughput of an existing table:

- **Read capacity units—** It is the number of consistent reads of items up to 1 KB in size per second. For example, when you request 10 read capacity units, you are requesting a throughput of 10 consistent reads of 1 KB size per second throughput for that table. For more information about consistent read, see Data Read and Consistency Considerations (p. 7).
- **Write capacity units—** It is the number of 1 KB writes/second. For example, when you request 10 write capacity units, you are requesting a throughput of 10 writes of 1 KB size per second for that table.

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

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

- **Item size—** Amazon DynamoDB allocates resources for your table based on the capacity units that you specify. These capacity units are based on 1 KB data read or written per request. For example, if your table has 1 KB items, then each item read/write operation consumes 1 capacity unit. However, if you items are larger than 1 KB, then each read/write operation might consume more capacity units, in which case you can perform fewer reads/writes per second. For example, if you request 10 read capacity units throughput for a table, but your items are 2 KB in size, then you will get a maximum of 5 consistent reads/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—**By the preceding capacity unit definition, the capacity units are based on consistent read operations, which require more effort and consume about twice as many resources as the eventually consistent reads. For example, a table that has 10 read capacity units of provisioned throughput would provide either 10 consistent reads per second of 1 KB items, or 20 eventually consistent reads per second of the same items. So, whether your application requires consistent reads or eventually consistent reads is a factor in your determination of the read capacity units for your table. Note that, by default, DynamoDB read operations are eventually consistent. Some of these operations allow you to specify consistent read.

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 Amazon 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 Amazon DynamoDB Console.

Also, note that this throughput scheme requires you to spread read/write requests across table partitions stored on different servers. For example, you might provision 1 million reads/second throughput. However, if you send 1 million requests for the same item in the table, the table partitioning scheme cannot help to produce the expected throughput.

The following table compares some provisioned throughput values for different average item sizes, read request rates, and consistency combinations.

Expected Item Size	Consistency	Desired Reads Per Second	Provisioned Throughput Required
1KB	Consistent	50	50
2KB	Consistent	50	100
1KB	Eventually Consistent	50	25
2KB	Eventually Consistent	50	50

Item sizes are rounded up to the next 1 KB multiple. For example, an item of 3,500 bytes consumes the same throughput as a 4 KB item.

Amazon 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 data, exceeds the current reserved capacity, Amazon DynamoDB returns an error that indicates that the provisioned throughput level has been exceeded.

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 (p. 439) API to increase the `provisionedThroughput` value. For the current maximum Provisioned Throughput values per table or account, see Limits in Amazon DynamoDB (p. 257).

Set your Provisioned throughput using the `provisionedThroughput` parameter. For information about setting the `provisionedThroughput` parameter, see CreateTable (p. 394) and UpdateTable (p. 439).

For information about using provisioned throughput, see Provisioned Throughput Guidelines in Amazon DynamoDB (p. 68).

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)

The basic rule is that if your request reads or writes an item of 1 KB in size, you consume 1 capacity unit. If an operation reads or writes 10 KB of items, then the operation consumes 10 capacity units. This section describes how Amazon DynamoDB computes the item size for the purpose of determining capacity units cosumed by an operation. In case of a read operation the section describes the impact of consist vs. the eventually consistent read on the capacity unit consumed by the read operation.

Item Size Calculations

For each request you send, Amazon DynamoDB computes the capacity units consumed by that operation. Item size is one of the factors it uses in computing the capacity units consumed. The size of an item is sum of lengths of its attribute names and values. This section describes how Amazon DynamoDB determines the size of item(s) involved in an operation.

The get, put, and delete operations involve one item. However, batch get, query and scan operations can return multiple items.

For operations that involve only one item, Amazon DynamoDB rounds the item size up to the next 1 KB. For example, if you get, put, or delete an item of 1.6 KB in size, Amazon DynamoDB rounds the items size to 2 KB. This rounding also applies to batch get operation, which operates on several items. Amazon DynamoDB rounds the size of each individual item returned in the batch. For example, if you use the batch get operation to retrieve 2 items of 1.2 KB and 3.6 KB, Amazon DynamoDB rounds these items sizes to 2 KB and 4 KB respectively, resulting a total size for the operation of 6 KB.

A query or scan can return multiple items. By default Amazon DynamoDB returns up to 1 MB of items for a query or scan. In this case Amazon DynamoDB computes the total item size for the request by computing the sum of all items sizes and then rounding to the next KB. For example, suppose your query returns 10 items whose combined size is 10.2 KB. Amazon DynamoDB rounds the item size for the operation to 11 KB, for the purpose of computing capacity units consumed by that operation. Note that unlike for single item operations, this size is not necessarily proportional to the number of items. Instead it is the cumulative size of processed items, rounded up to the next KB increment. For example, if your query returns 1,500 items of 64 bytes each, the cumulative size is 94 KB, not 1,500 KB.

In case of a scan operation, it is not the size of items returned by scan, rather it is the size of items evaluated by Amazon DynamoDB. That is, for a scan requests, Amazon DynamoDB evaluates up to 1 MB of items and returns only the items that satisfy the scan condition.

Any of the preceding operations that return items allow you to request a subset of attributes to retrieve. However, this has no impact on the item size calculations. Also, 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 Amazon DynamoDB has to read each item to increment the count.

The put operation adds an item to the table. However, if an item with the same primary key exists in the table, the operation replaces the item. In this case, the item size is the larger of the two. Similarly for an update operation, Amazon DynamoDB uses larger of the new and old item as the item size. When you send a request to delete an item, Amazon DynamoDB uses the size of the deleted item.

Note

When computing the size of an item, the size of the attribute names is included as well. In other words, the size of an item is the cumulative size of all attribute names and values. The read capacity consumption of a workload executing queries returning several items each can be optimized by making the items as small as possible. The easiest way to achieve that is to minimize the length of the attribute names. Additionally, you can also reduce item size by storing less frequently accessed item attributes in a separate table.

Note

In computing the storage used by the table, Amazon DynamoDB add 100 bytes of overhead to each item for indexing purposes. The Describe Table API returns 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 on pricing, go to detail page.

Read Operation and Consistency

For a read operation, the preceding calculations assume consistent read requests. For an eventually consistent read request, the operation consumes only half the capacity units. That is, for your eventual consistent read, if total item size is 10 KB, the operation consumes only 5 capacity units.

Provisioned Throughput Guidelines in Amazon DynamoDB

Topics

The following factors influence provisioned throughput performance.

Uniform Workload

Provisioned throughput is dependent on the primary key selection, and the workload patterns on individual items. When storing data, Amazon DynamoDB divides a table's items into multiple partitions, and distributes the data primarily based on 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.

```
Total provisioned throughput/partitions = 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, traffic is concentrated on a small number of partitions –

potentially only one partition. If the workload is heavily unbalanced, meaning disproportionately focused on one or a few partitions, the operations will not achieve the overall provisioned throughput level. To get the most out of Amazon DynamoDB throughput, build tables where the hash key element has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible.

This behavior does not imply that you need to access all of the hash keys, or even that the percentage of accessed hash keys needs to be high to achieve your throughput level. But, be aware that when your workload accesses more distinct hash keys, those operations are spread out across the partitioned space in a manner that better utilizes your allocated throughput level. In general, you utilize throughput more efficiently as the ratio of hash keys accessed to total hash keys in a table grows.

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

Hash key value	Efficiency
User ID, where the application has many users.	Good
Status code, where there are only a few possible status codes.	Bad
Device ID, where even if there are a lot of devices being tracked, one is by far more popular than all the others.	Bad

When the number of hash key values in a single table is very few, consider distributing your write operations across more distinct hash values. In other words, consider the primary key elements to avoid one "hot" (heavily requested) hash key value that slows overall performance.

For example, consider a composite primary hash and range key table where the hash key represents a device ID, and where device ID "D17" is particularly heavily requested. To increase the read and write throughput for this "hot" hash key, pick a random number chosen from a fixed set (for example 1 to 200) and concatenate it with the device ID (so you get `D17.1`, `D17.2` through `D17.200`). Due to randomization, writes for device ID "D17" are spread evenly across the multiple hash key values, yielding better parallelism and higher overall throughput.

This strategy greatly improves the write throughput, but reads for a specific item become harder since you don't know which of the 200 keys contains the item. You can improve this strategy to get better read characteristics: instead of choosing a completely random number, choose a number that you are able to calculate from something intrinsic to the item. For example, if the item represents a person that has the device, calculate the hash key suffix from their name, or user ID. This calculation should compute a number between 1 and 200 that is fairly evenly distributed given any set of names (or user IDs.) A simple calculation generally suffices (such as, the product of the ASCII values for the letters in the person's name modulo 200 + 1). Now, the writes are spread evenly across the hash keys (and thus partitions). *And* you can easily perform a get operation, because you can determine the hash key you need when you want to retrieve a specific "device owner" value. Query operations still need to run against all `D17.x` keys, and your application needs some logic on the client side to merge all of the query results for each hash key (200 in this case). But, the schema avoids having one "hot" hash key taking all of the workload.

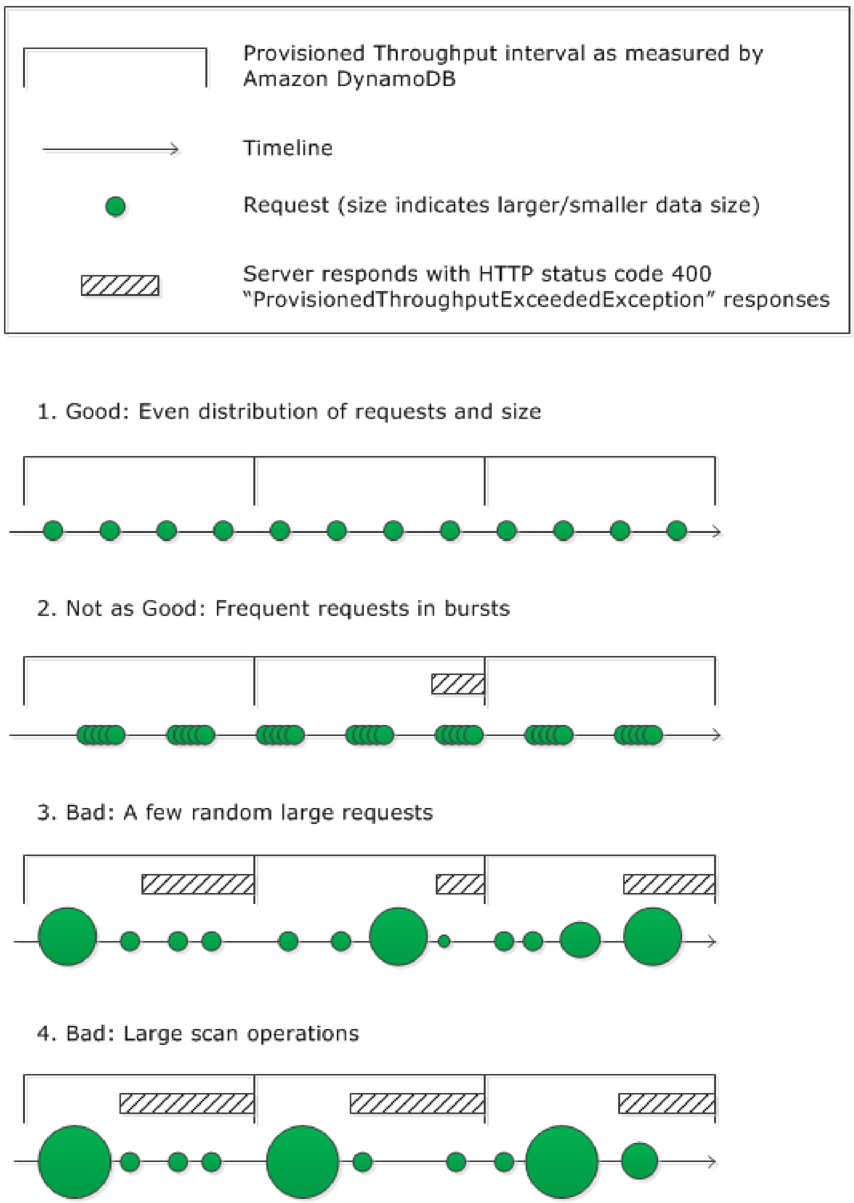
To increase the provisioned throughput, use UpdateTable (p. 439). For more information about hash key elements, see Primary Key (p. 5).

Scan Considerations

When you create a table, you set its read and write capacity unit requirements. These are based on a 1 KB data size (that is, the number of 1 KB data read/write requests per second) and consistent read. However, scan operation can return up to 1 MB (1 page) of data. That is, a single scan request consumes up to 1 MB / 1 KB = 500 capacity units (because scan returns only eventually consistent result which takes half the capacity units of a consistent read), which is a sudden burst of usage of the configured capacity units for the table. This sudden use of capacity units by a scan starves your other potentially more important requests for the same table from using the available capacity units. As a result, you likely get the "ProvisionThroughputExceeded" 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. So the request is hitting the same partition causing its capacity units to be consumed, throttling other requests to that partition. If the request to read data is spread across multiple partitions, then the operation would not throttle a specific partition.

The following diagram illustrates the impact of a sudden burst of capacity unit usage by scan/query operation and its impact on your other requests against the same table.



Instead of using a large scan operation, you can minimize the impact of a scan operation on a table's provisioned throughput using the following techniques.

- **Reduce page side**
  For example, instead of using the 1 MB of default page size, you could optionally set a smaller page size. Each scan/query request that has a smaller page size uses fewer available capacity units. This creates a "pause" between each request. For example, if you set the page size to 10 items, and each item is 1 KB, then a scan request consumes only 10 capacity units for consistent reads and 5 capacity units for eventually consistent reads. This allows your other critical requests to succeed without throttling. Amazon DynamoDB API supports `Limit` parameter to set the page size in your request.
- **Isolate scan operations**
  Amazon 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 (p. 439) API. 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. 383).

Data Upload Considerations

There are times when you load data from other data sources into Amazon DynamoDB. Typically, Amazon 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 Amazon 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, Amazon 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 Amazon 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
...	...

This sequence of upload uses different hash key in the sequence of upload keeping more Amazon DynamoDB servers busy simultaneously and improves your throughput performance.

Time Series Data and Access Patterns

For each table that you create, you specify the throughput requirements. Amazon 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 latest month's or week's data, 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.

So storing "hot" items in one table with higher throughput and "cold" items with reduced throughput requirements help you save on resources. You can remove old items by simply deleting the tables. You can optionally backup these table to other storage options such as Amazon Simple Storage Service (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.

Working with Tables Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

Topics

You can use the AWS SDK for Java 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 Amazon DynamoDB API. For more information, see API Reference for Amazon DynamoDB (p. 371).

The following are the common steps for table operations using the Java 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 an `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 (Provisioned Throughput) (p. 65). The following Java code snippet 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 Java low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
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.
3. Execute the `AmazonDynamoDBClient.CreateTable` method by providing the request object as a parameter.

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

```
client = new AmazonDynamoDBClient(credentials);
String tableName = "ProductCatalog";
```

```
KeySchemaElement hashKey = new KeySchemaElement().withAttributeName("Id").withAt
tributeType("N");
KeySchema ks = new KeySchema().withHashKeyElement(hashKey);

ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()
  .withReadCapacityUnits(10L)
  .withWriteCapacityUnits(10L);

CreateTableRequest request = new CreateTableRequest()
  .withTableName(tableName)
  .withKeySchema(ks)
  .withProvisionedThroughput(provisionedThroughput);

CreateTableResult result = client.createTable(request);
```

You must wait until Amazon DynamoDB creates the table and sets the table status to `ACTIVE`. The `createTable` request returns a `CreateTableResult` from which you can get the `TableDescription` property that provides the necessary table information.

```
TableDescription tableDescription = result.getTableDescription();

System.out.printf("%s: %s \t ReadCapacityUnits: %d \t WriteCapacityUnits: %d",

  tableDescription.getTableStatus(),
  tableDescription.getTableName(),
  tableDescription.getProvisionedThroughput().getReadCapacityUnits(),
  tableDescription.getProvisionedThroughput().getWriteCapacityUnits());
```

You can also call the `describeTable` method of the client to get table information at anytime.

```
TableDescription tableDescription = client.describeTable(
  new DescribeTableRequest().withTableName(tableName)).getTable();
```

Updating a Table

You can update only the provisioned throughput values of an existing table. Depending on you 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 once in a 24 hour period. For additional guidelines and limitations, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65).

The following are the steps to update a table using the Java low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
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 Java code snippet demonstrates the preceding steps.

```
client = new AmazonDynamoDBClient(credentials);
String tableName = "ProductCatalog";

ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()
  .withReadCapacityUnits(15L)
  .withWriteCapacityUnits(12L);

UpdateTableRequest updateTableRequest = new UpdateTableRequest()
  .withTableName(tableName)
  .withProvisionedThroughput(provisionedThroughput);

UpdateTableResult result = client.updateTable(updateTableRequest);
```

Deleting a Table

The following are the steps to delete a table using the Java low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
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 Java code snippet demonstrates the preceding steps.

```
client = new AmazonDynamoDBClient(credentials);
String tableName = "ProductCatalog";

DeleteTableRequest deleteTableRequest = new DeleteTableRequest()
  .withTableName(tableName);
DeleteTableResult result = client.deleteTable(deleteTableRequest);
```

Listing Tables

To list tables in your account using the AWS SDK for Java low-level API, create an instance of the `AmazonDynamoDBClient` and execute the `listTables` method. The ListTables (p. 411) API 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 Java 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.

```
client = new AmazonDynamoDBClient(credentials);

// Initial value for the first page of table names.
String lastEvaluatedTableName = null;
do {

    ListTablesRequest listTablesRequest = new ListTablesRequest()
```

```
    .withLimit(10)
    .withExclusiveStartTableName(lastEvaluatedTableName);

    ListTablesResult result = client.listTables(listTablesRequest);
    lastEvaluatedTableName = result.getLastEvaluatedTableName();

    for (String name : result.getTableNames()) {
        System.out.println(name);
    }

} while (lastEvaluatedTableName != null);
```

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

The following Java example uses the AWS SDK for Java low-level 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 Amazon DynamoDB (p. 260).

```
import com.amazonaws.AmazonServiceException;

import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.model.CreateTableRequest;

import com.amazonaws.services.dynamodb.model.CreateTableResult;

import com.amazonaws.services.dynamodb.model.DeleteTableRequest;

import com.amazonaws.services.dynamodb.model.DeleteTableResult;

import com.amazonaws.services.dynamodb.model.DescribeTableRequest;

import com.amazonaws.services.dynamodb.model.KeySchema;

import com.amazonaws.services.dynamodb.model.KeySchemaElement;

import com.amazonaws.services.dynamodb.model.ListTablesRequest;

import com.amazonaws.services.dynamodb.model.ListTablesResult;

import com.amazonaws.services.dynamodb.model.ProvisionedThroughput;

import com.amazonaws.services.dynamodb.model.TableDescription;

import com.amazonaws.services.dynamodb.model.TableStatus;

import com.amazonaws.services.dynamodb.model.UpdateTableRequest;
```

```
import com.amazonaws.services.dynamodb.model.UpdateTableResult;

public class LowLevelTableExample {

    static String tableName = "ExampleTable";
    static AmazonDynamoDBClient client;

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

        // You need a client to send requests.
        createClient();

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

        deleteExampleTable();
    }

    static void createClient() throws Exception {
        AWSCredentials credentials = new PropertiesCredentials(
        LowLevelTableExample.class.getResourceAsStream("AwsCredentials.proper
ties"));

        client = new AmazonDynamoDBClient(credentials);
    }

    static void createExampleTable() {
```

```
        KeySchemaElement hashKey = new KeySchemaElement().withAttribute
Name("Id").withAttributeType("N");
        KeySchema ks = new KeySchema().withHashKeyElement(hashKey);


        // Provide the initial provisioned throughput values as Java long data
 types
        ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()

            .withReadCapacityUnits(5L)
            .withWriteCapacityUnits(6L);
        CreateTableRequest request = new CreateTableRequest()
            .withTableName(tableName)
            .withKeySchema(ks)
            .withProvisionedThroughput(provisionedThroughput);


        CreateTableResult result = client.createTable(request);


        waitUntilTableReady(tableName);


        getTableInformation();


    }


    static void listMyTables() {
        String lastEvaluatedTableName = null;
        do {


            ListTablesRequest listTablesRequest = new ListTablesRequest()
            .withLimit(10)
            .withExclusiveStartTableName(lastEvaluatedTableName);
```

```
            ListTablesResult result = client.listTables(listTablesRequest);

            lastEvaluatedTableName = result.getLastEvaluatedTableName();


            for (String name : result.getTableNames()) {

                System.out.println(name);

            }


        } while (lastEvaluatedTableName != null);

    }


    static void getTableInformation() {


        TableDescription tableDescription = client.describeTable(

              new DescribeTableRequest().withTableName(tableName)).getTable();

        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().getReadCapacity
Units(),

                tableDescription.getProvisionedThroughput().getWriteCapacity
Units());

    }


    static void updateExampleTable() {
```

```
       ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()

        .withReadCapacityUnits(6L)
        .withWriteCapacityUnits(7L);

        UpdateTableRequest updateTableRequest = new UpdateTableRequest()
            .withTableName(tableName)
            .withProvisionedThroughput(provisionedThroughput);

        UpdateTableResult result = client.updateTable(updateTableRequest);
        waitUntilTableReady(tableName);
    }

    static void deleteExampleTable() {
        DeleteTableRequest deleteTableRequest = new DeleteTableRequest()
            .withTableName(tableName);
        DeleteTableResult result = client.deleteTable(deleteTableRequest);
        waitForTableToBeDeleted(tableName);
    }

    private static void waitUntilTableReady(String tableName) {
       System.out.println("Waiting for " + tableName + " to become ACTIVE...");


        long startTime = System.currentTimeMillis();
        long endTime = startTime + (10L * 60L * 1000L);
        while (System.currentTimeMillis() < endTime) {
            try {Thread.sleep(1000L * 20L);} catch (Exception e) {}
            try {
                TableDescription tableDescription = client.describeTable(new
DescribeTableRequest().withTableName(tableName)).getTable();
```

```
                String tableStatus = tableDescription.getTableStatus();


                System.out.println("  - current state: " + tableStatus);


                if (tableStatus.equals(TableStatus.ACTIVE.toString())) {
                    return;
                }
            } catch (AmazonServiceException ase) {
                // Describe table is eventual consistent.
               if (ase.getErrorCode().equalsIgnoreCase("ResourceNotFoundExcep
tion") == false) {
                    throw ase;
                }
            }
        }
       throw new RuntimeException("Table " + tableName + " never went active");

    }


    private static void waitForTableToBeDeleted(String tableName) {
        System.out.println("Waiting for " + tableName + " while status DELET
ING...");


        long startTime = System.currentTimeMillis();
        long endTime = startTime + (10L * 60L * 1000L);
        while (System.currentTimeMillis() < endTime) {
            try {Thread.sleep(1000 * 20);} catch (Exception e) {}
            try {
              DescribeTableRequest request = new DescribeTableRequest().withT
ableName(tableName);
                TableDescription tableDescription = client.describeTable(re
```

```
quest).getTable();

                String tableStatus = tableDescription.getTableStatus();

                System.out.println("  - current state: " + tableStatus);

                if (tableStatus.equals(TableStatus.ACTIVE.toString())) {

                    return;

                }

            } catch (AmazonServiceException ase) {

               if (ase.getErrorCode().equalsIgnoreCase("ResourceNotFoundExcep
tion") == true) {

                    System.out.println("Table " + tableName + " is not found.
It was deleted.");

                    return;

                }

                else {

                    throw ase;

                }

            }

        }

        throw new RuntimeException("Table " + tableName + " did not go active
after 10 minutes.");

    }



}
```

Working with Tables Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB

Topics

- Example: Create, Update, Delete, and List Tables Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB (p. 86)

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 Amazon DynamoDB API. For more information, see API Reference for Amazon DynamoDB (p. 371).

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 (Provisioned Throughput) (p. 65).

The following are the steps to create a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
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.

```
client = new AmazonDynamoDBClient(credentials);
string tableName = "ProductCatalog";

var request = new CreateTableRequest
{
  KeySchema = new KeySchema
  {
    HashKeyElement = new KeySchemaElement
    {
      AttributeName = "Id",
      AttributeType = "N"
    },
    RangeKeyElement = new KeySchemaElement
     {
       AttributeName = "ReplyDateTime",
       AttributeType = "N"
```

```
      }
  },
  ProvisionedThroughput = new ProvisionedThroughput
  {
    ReadCapacityUnits = 10,
    WriteCapacityUnits = 5
  },
  TableName = tableName
};

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

You must wait until Amazon 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 you 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 once 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 (Provisioned Throughput) (p. 65).

The following are the steps to update a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
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.

```
client = new AmazonDynamoDBClient(credentials);
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 (the client).
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.

```
client = new AmazonDynamoDBClient(credentials);
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 (p. 411) API 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.

```
client = new AmazonDynamoDBClient(credentials);

// 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 for Amazon DynamoDB

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 with Amazon DynamoDB (p. 305).

```
using System;

using Amazon.DynamoDB.Model;

using Amazon.Runtime;



namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static string tableName = "ExampleTable";

    private static AmazonDynamoDBClient client;



    static void Main(string[] args)

    {

      // You need client to send requests.

      client = new AmazonDynamoDBClient();
```

```
    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
    {
      KeySchema = new KeySchema
      {
        HashKeyElement = new KeySchemaElement
        {
          AttributeName = "Id",
          AttributeType = "N"
```

```
        },
        RangeKeyElement = new KeySchemaElement
        {
          AttributeName = "ReplyDateTime",
          AttributeType = "N"
        }
      },
      ProvisionedThroughput = new ProvisionedThroughput
      {
        ReadCapacityUnits = 5,
        WriteCapacityUnits = 6
      },
      TableName = tableName
    };

    var response = client.CreateTable(request);

    var result = response.CreateTableResult;
    var tableDescription = result.TableDescription;
    Console.WriteLine("{1}: {0} \t ReadsPerSec: {2} \t WritesPerSec: {3}",
                    tableDescription.TableStatus,
                    tableDescription.TableName,
                   tableDescription.ProvisionedThroughput.ReadCapacityUnits,
                 tableDescription.ProvisionedThroughput.WriteCapacityUnits);

    string status = tableDescription.TableStatus;
    Console.WriteLine(tableName + " - " + status);
```

```
      WaitUntilTableReady(tableName);


    }


    private static void ListMyTables()
    {
      Console.WriteLine("\n*** listing tables ***");
      string lastTableNameEvaluated = null;
      do
      {
        var request = new ListTablesRequest
        {
          Limit = 2,
          ExclusiveStartTableName = lastTableNameEvaluated
        };

        var response = client.ListTables(request);
        ListTablesResult result = response.ListTablesResult;
        foreach (string name in result.TableNames)
          Console.WriteLine(name);

        lastTableNameEvaluated = result.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.DescribeTableResult.Table;
    Console.WriteLine("Name: {0}", description.TableName);
    Console.WriteLine("# of items: {0}", description.ItemCount);
    Console.WriteLine("Provision Throughput (reads/sec): {0}",
                     description.ProvisionedThroughput.ReadCapacityUnits);
    Console.WriteLine("Provision Throughput (writes/sec): {0}",
                     description.ProvisionedThroughput.WriteCapacityUnits);

  }

  private static void UpdateExampleTable()
  {
    Console.WriteLine("\n*** Updating table ***");
    var request = new UpdateTableRequest()
    {
      TableName = tableName,
      ProvisionedThroughput = new ProvisionedThroughput()
      {
        ReadCapacityUnits = 6,
        WriteCapacityUnits = 7
      }
    };
```

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

      WaitUntilTableReady(tableName);
    }

    private static void DeleteExampleTable()
    {
      Console.WriteLine("\n*** Deleting table ***");
      var request = new DeleteTableRequest
      {
        TableName = tableName
      };

      var response = client.DeleteTable(request);

      var result = response.DeleteTableResult;
      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.DescribeTableResult.Table.TableName,
                         res.DescribeTableResult.Table.TableStatus);
          status = res.DescribeTableResult.Table.TableStatus;
        }
        catch (ResourceNotFoundException resouceNotFound)
        {
          // 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 for Amazon DynamoDB

Topics

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 Amazon DynamoDB API. For more information, see API Reference for Amazon DynamoDB (p. 371).

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

1. Create an instance of the `AmazonDynamoDB` client class.
2. Provide the parameters for an Amazon DynamoDB operation to the client instance, including any optional parameters.
3. Load the response from Amazon 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 (Provisioned Throughput) (p. 65). 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 `AmazonDynamoDB` class (the client).
2. Provide the parameters for the `create_table` operation to the client instance.

   You must provide the table name, its primary key, and the provisioned throughput values.
3. Load the response into a local variable, such as `$create_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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$create_response = $dynamodb->create_table(array(
    'TableName' => 'ProductCatalog',
    'KeySchema' => array(
        'HashKeyElement' => array(
            'AttributeName' => 'Id',
            'AttributeType' => AmazonDynamoDB::TYPE_NUMBER
        )
    ),
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 10,
        'WriteCapacityUnits' => 5
    )
));

print_r($create_response);
```

You must wait until Amazon DynamoDB creates the table and sets the table status to `ACTIVE` before you can put data into the table. You can use the AWS SDK for PHP's `describe_table` function to poll for the table's status until it is in the `ACTIVE` state. For more information, see the Amazon DynamoDB API DescribeTable (p. 406).

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

```
// Poll and sleep until the table is ready.
$count = 0;
do {
sleep(1);
$count++;

$describe_response = $dynamodb->describe_table(array(
    'TableName' => $table_name1
));
}
while ((string) $describe_response->body->Table->TableStatus !== 'ACTIVE');

// Success?
print_r($describe_response);
```

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 once in a 24 hour period. For additional guidelines and limitations, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65).

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

1. Create an instance of the `AmazonDynamoDB` client class.
2. Provide the parameters for the `update_table` 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 `$update_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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$update_response = $dynamodb->update_table(array(
    'TableName' => 'ProductCatalog',
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 8,
        'WriteCapacityUnits' => 5
    )
));
```

```
// Success?
print_r($update_response);
```

Deleting a Table

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

1. Create an instance of the `AmazonDynamoDB` client class.
2. Provide the parameters for the `delete_table` 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 `$delete_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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$delete_response = $dynamodb->delete_table(array(
    'TableName' => 'ProductCatalog'
));

// Success?
print_r($delete_response);
```

Listing Tables

To list tables in your account using the AWS SDK for PHP, create an instance of the `AmazonDynamoDB` client class and execute the `list_tables` operation. The ListTables (p. 411) API requires no parameters.

The following PHP code snippet gets all of the table names for the current account.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

echo PHP_EOL . PHP_EOL;
echo '# A list of all tables in the current account:' . PHP_EOL;

$list_response = $dynamodb->list_tables();

var_dump($list_response->body->TableNames()->map_string());
```

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, Amazon DynamoDB returns a `LastEvalutedTableName` value. Use the `LastEvalutedTableName` value for the `ExclusiveStartTableName` parameter to get the next page of results.

The following PHP code snippet demonstrates how to get the `LastEvalutedTableName` value for the *`ExclusiveStartTableName`* parameter, using a *`Limit`* value of 2 table names per page.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$tables = array();

// Collect all table names
do {
    $list_response = $dynamodb->list_tables(array(
        'Limit' => 2,
        'ExclusiveStartTableName' => isset($list_response) ? (string)
$list_response->body->LastEvaluatedTableName : null
    ));
 print_r($list_response);
    $tables = array_merge($tables, $list_response->body->TableNames()-
>map_string());
}
while ($list_response->body->LastEvaluatedTableName);

// Success?
print_r(count($tables));
```

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

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 for Amazon DynamoDB (p. 369).

```
<?php

// If necessary, reference the sdk.class.php file.

// For example, the following line assumes the sdk.class.php file is

// in an sdk sub-directory relative to this file

require_once dirname(__FILE__) . '/sdk/sdk.class.php';



// Instantiate the class

$dynamodb = new AmazonDynamoDB();
```

```
$table_name = 'ExampleTable';

######################################################################
# Create a new DynamoDB table

$response = $dynamodb->create_table(array(
    'TableName' => $table_name,
    'KeySchema' => array(
        'HashKeyElement' => array(
            'AttributeName' => 'Id',
            'AttributeType' => AmazonDynamoDB::TYPE_NUMBER
        ),
        'RangeKeyElement' => array(
            'AttributeName' => 'Date',
            'AttributeType' => AmazonDynamoDB::TYPE_NUMBER
        )
    ),
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits'  => 5,
        'WriteCapacityUnits' => 5
    )
));

// Check for success...
if ($response->isOK())
{
    echo '# Kicked off the creation of the DynamoDB table...' . PHP_EOL;
}
else
```

```
{
    print_r($response);
}


####################################################################
# Sleep and poll until the table has been created


$count = 0;
do {
    sleep(1);
    $count++;


    $response = $dynamodb->describe_table(array(
        'TableName' => $table_name
    ));
}
while ((string) $response->body->Table->TableStatus !== 'ACTIVE');


echo "The table \"${table_name}\" has been created (slept ${count} seconds)."
. PHP_EOL;


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


echo PHP_EOL . PHP_EOL;
echo '# Collecting a complete list of tables in the account...' . PHP_EOL;


$response = $dynamodb->list_tables();
var_dump($response->body->TableNames()->map_string());
```

```
####################################################################

# Updating the table


echo PHP_EOL . PHP_EOL;

echo "# Updating the \"${table_name}\" table..." . PHP_EOL;


$dynamodb->update_table(array(

    'TableName' => $table_name,

    'ProvisionedThroughput'  => array(

        'ReadCapacityUnits'  => 10,

        'WriteCapacityUnits' => 5

    )

));


$table_status = $dynamodb->describe_table(array(

    'TableName' => $table_name

));


// Check for success...

if ($table_status->isOK())

{

   print_r($table_status->body->Table->ProvisionedThroughput->to_array()->get
ArrayCopy());

}

else

{

    print_r($table_status);

}
```

```
####################################################################

# Sleep and poll until the table has been updated.


$count = 0;

do {

    sleep(5);

    $count += 5;


    $response = $dynamodb->describe_table(array(

        'TableName' => $table_name

    ));

}

while ((string) $response->body->Table->TableStatus !== 'ACTIVE');


echo "The table \"${table_name}\" has been updated (slept ${count} seconds)."
. PHP_EOL;




####################################################################

# Deleting the table

 /* The following demonstrates how to delete the table, but is commented out
so you can see the data

 * until you're ready to delete it.


echo PHP_EOL . PHP_EOL;

echo "# Deleting the \"${table_name}\" table..." . PHP_EOL;


$response = $dynamodb->delete_table(array(

    'TableName' => $table_name

));
```

```
// Check for success...

if ($response->isOK())

{

    echo 'The table is in the process of deleting...' . PHP_EOL;

}

else

{

    print_r($response);

}


####################################################################

# Sleep and poll until the table has been deleted.


$count = 0;

do {

    sleep(1);

    $count++;


    $response = $dynamodb->describe_table(array(

        'TableName' => $table_name

    ));

}

while ((integer) $response->status !== 400);


echo "The table \"${table_name}\" has been deleted (slept ${count} seconds)."
. PHP_EOL;

*/

?>
```

Working with Items in Amazon DynamoDB

Topics

In Amazon DynamoDB, an item is a collection of attributes. Each attribute has a name and a value. An attribute value can be a number, a string, a number set, or a string set. When you add an item, the primary key attribute(s) are the only required attributes. For more information, see Amazon DynamoDB Data Model (p. 3). In addition to the primary key, an item can have any number of attributes, although there is a limit of 64 KB on the item size. An item size is the sum of lengths of its attribute names and values (binary and UTF-8 lengths). So it helps if you keep attribute names short.

Except for the primary key, there is no predefined schema for the items in a table. For example, to store various 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 ProductCatalog table. Note that the example uses JSON-like syntax to show the attribute value.

Id (Primary Key)	Other Attributes

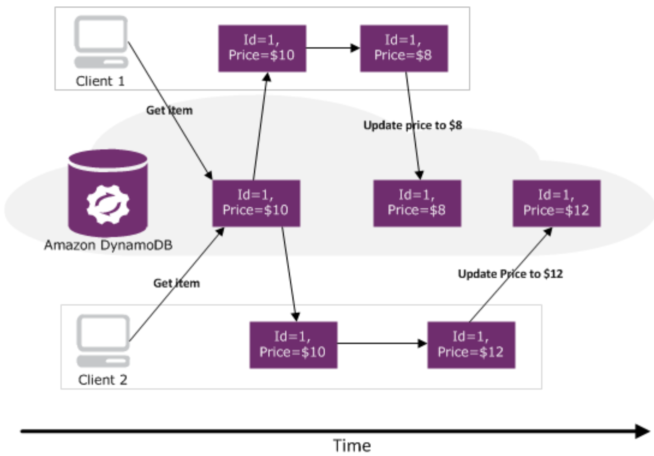
101	`{`
	`   Title = "Book 101 Title"`
	`   ISBN = "111-1111111111"`
	`   Authors = "Author 1"`
	`   Price = -2`
	`   Dimensions = "8.5 x 11.0 x 0.5"`
	`   PageCount = 500`
	`   InPublication = 1`
	`   ProductCategory = "Book"`
	`}`

Id (Primary Key)	Other Attributes
201	{ Title = "18-Bicycle 201" Description = "201 description" BicycleType = "Road" Brand = "Brand-Company A" Price = 100 Gender = "M" Color = ["Red", "Black"] ProductCategory = "Bike" }

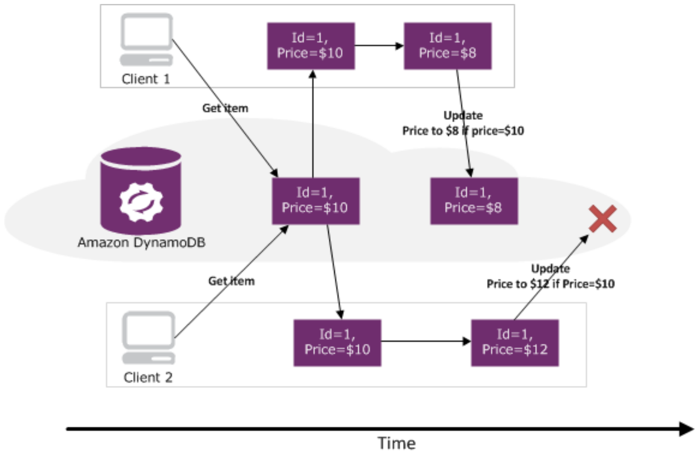
Amazon DynamoDB provides API to add, retrieve, update, and delete items. In addition, it also provides an API to retrieve a batch of items from one or more tables. The create, update, and delete APIs support conditional update. For example, when you put an item and if the item exists, Amazon DynamoDB replaces that item, by default. However, you can use the "exists" condition to determine whether you want the item replaced. Similarly, when you delete an item, you might want Amazon DynamoDB to display certain attribute values before deleting an item.

Conditional Writes

In a multi-user environment it is possible that multiple clients access the same item and attempt to update its attiribute values at the same time not realizing that the item has been updated by some other client. This is shown in the following illustration in which client 1 and client 2 get a copy of an item (Id=1). Client 1 updates the price from $10 to $8. Later client 2 updates the same item price to $12 and the update made by client 1 is lost.



To help clients coordinate updates, Amazon DynamoDB supports conditional writes in which case it applies the update only if the current item attribute(s) meet the specified condition(s). When you send a conditional write request, Amazon DynamoDB performs the operation only if the specified condition is met, otherwise it returns an error. For example, the following illustration shows a conditional update where Client 1 and Client 2 both get a copy of an item (Id=1). Client 1 first updates the item price to $8 with a condition that the existing item price on server must be $10 and the operation succeeds. Client 2 then attempts to update price to $12 with a condition that the existing item price on the server be $10 and the operation fails.



Note that the conditional update is an idempotent operation, you can send the same request multiple times and it has no further effect on the item after the first time Amazon DynamoDB performs the specified update. For example, suppose you send a conditional update request to update the price of a book item by 10% only if the current price is $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 Amazon DynamoDB updates the price only if current price is still $20.

Atomic Counters

Amazon DynamoDB also supports atomic counters that allow you to increment or decrement the value of an existing attribute without interfering with another simultaneous write requests. You can use the Amazon DynamoDB update API to increment/decrement an attribute value, instead of replacing the value. For example, a web application might want to maintain a counter per visitor to their site. In this case, the client wants to increment a value regardless of what the current value is. Note that, unlike the conditional update operation described in the preceding section, this is not an idempotent operation. That is, you might retry this operation in the event you don't know if your previous request was successful or not; however, you risk applying the same update twice. So this feature is great 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 conditional updates.

Consistent vs Eventually Consistent Read

Amazon DynamoDB maintains multiple copies of each item to ensure durability. For each successful write request, Amazon 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 your read request immediately after a write might not show the change. You can optionally request the most up to date version of the data, which takes additional read capacity units for Amazon DynamoDB to get data for the latest item. An eventually consistent get request consumes half the read capacity units as the consistent request. So it is good to design applications to take advantage of eventual consistent read where possible. Consistency across all copies of the data is usually reached within a second.

Capacity Units Consumed by Various Item Operations

When you create a table you specify your read and write capacity unit requirements. When you send requests such as get, update or delete an item, you consume the capacity units set for the table. For more information about how Amazon DynamoDB computes the capacity units consumed by your operation, see Capacity Units Calculations for Various Operations (p. 67).

You can use the Amazon DynamoDB API to work with items or use the AWS SDK libraries for Java, .NET, PHP, and Ruby. Click one of the links provided at the beginning of this section to learn more about how the specific AWS SDK library APIs support the item operations. All these sections provide working samples.

To learn more about Amazon DynamoDB items and other data model concepts, see Amazon DynamoDB Data Model (p. 3).

Working with Items Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

Topics

You can use AWS SDK for Java low-level API (protocol-level API) to perform typical create, read, update, and delete (CRUD) operations on an item in a table. The Java API for item operations maps to the underlying Amazon DynamoDB API. For more information, see API Reference for Amazon DynamoDB (p. 371).

The following are the common steps to perform data create, read, update, and delete (CRUD) operations using the Java low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
2. Provide the required operation specific information by creating a corresponding request object, for example, create a `PutItemRequest` object to upload an item and the `GetItemRequest` object to retrieve an existing item.

   You can use the request object to also provide any optional parameters supported by the operation.
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 `AmazonDynamoDBClient.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. 111).

The following are the commons steps to upload an item using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` by providing your credentials.
2. Create an instance of the `PutItemRequest` class by providing the name of the table to which you want to add the item and the item that you wish to upload.

3. Execute the `putItem` method by providing the `PutItemRequest` object that you created in the preceding step.

The following Java code snippet demonstrates the preceding tasks. The snippet stores an item in the ProductCatalog table.

```
client = new AmazonDynamoDBClient(credentials);
String tableName = "ProductCatalog";

Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
item.put("Id", new AttributeValue().withN("104"));
item.put("Title", new AttributeValue().withS("Book 104 Title"));
item.put("ISBN", new AttributeValue().withS("111-1111111111"));
item.put("Price", new AttributeValue().withS("25.00"));
item.put("Authors", new AttributeValue()
    .withSS(Arrays.asList("Author1", "Author2")));

PutItemRequest putItemRequest = new PutItemRequest()
  .withTableName(tableName)
  .withItem(item);
PutItemResult result = client.putItem(putItemRequest);
```

In the preceding example, you upload a book item that has the Id, Title, ISBN, and Authors attributes. Note that the Authors attribute is a multi-valued string attribute.

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 an `ConditionalCheckFailedException`. The code snippet specifies the following optional parameters in the `PutItemRequest`:

- A list of `ExpectedAttributeValue` objects that define 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.
- One of the `ReturnValue` enumeration values that defines what type of data the `putItem` request returns.

```
Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
item.put("Id", new AttributeValue().withN("104"));
item.put("Title", new AttributeValue().withS("Book 104 Title"));
item.put("ISBN", new AttributeValue().withS("222-2222222222"));
item.put("Price", new AttributeValue().withS("20.00"));
item.put("Authors", new AttributeValue()
    .withSS(Arrays.asList("Author1", "Author2")));

// Optional parameters Expected and ReturnValue.
Map<String, ExpectedAttributeValue> expected = new HashMap<String, ExpectedAt
tributeValue>();
expected.put("ISBN", new ExpectedAttributeValue()
    .withValue( new AttributeValue().withS("111-1111111111")));
ReturnValue retVal = ReturnValue.ALL_OLD;
```

```
PutItemRequest putItemRequest = new PutItemRequest()
    .withTableName(tableName)
    .withItem(item)
    .withExpected(expected)
    .withReturnValues(retVal);
PutItemResult result = client.putItem(putItemRequest);
```

For more information about the parameters and the API, see PutItem (p. 413).

Getting an Item

The `AmazonDynamoDBClient.getItem` method retrieves an item. To retrieve multiple items, you can use the `batchGetItem` method.

The following are the commons steps that you follow to retrieve an item using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` by providing your credentials.
2. Create an instance of the `GetItemRequest` class by providing the name of the table from which you want to retrieve an item and the primary key of the item that you want to retrieve.
3. Execute the `getItem` method by providing the `GetItemRequest` object that you created in the preceding step.

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

```
GetItemRequest getItemRequest = new GetItemRequest()
  .withTableName(tableName)
  .withKey(new Key()
  .withHashKeyElement(new AttributeValue().withN("101")));

GetItemResult result = client.getItem(getItemRequest);
Map<String, AttributeValue> map = result.getItem();
```

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. The code example specifies the following optional parameters in the `GetItemRequest`:

- A list of names that defines the attributes to retrieve.
- A Boolean value that specifies whether to request only the latest data, that is, get data that is consistent. To learn more about consistent reads, see Data Read and Consistency Considerations (p. 7).

```
List<String> attributesToGet = new ArrayList<String>(
        Arrays.asList("Id", "ISBN", "Title", "Authors"));

GetItemRequest getItemRequest = new GetItemRequest()
    .withTableName(tableName)
    .withKey(new Key().withHashKeyElement(new AttributeValue().withN("201")))
    .withAttributesToGet(attributesToGet)
    .withConsistentRead(true);
```

```
GetItemResult result = client.getItem(getItemRequest);
Map<String, AttributeValue> map = result.getItem();
```

For more information about the parameters and the API, see GetItem (p. 409).

Batch Write: Putting and Deleting Multiple Items

Batch write refers to putting and deleting multiple items in a batch. The `AmazonDynamoDBClient.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 low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class by providing your credentials.
2. Create an instance of the `BatchWriteItemRequest` class that describes all the put and delete operations.
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, Amazon DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, Amazon DynamoDB rejects the request. For more information, see BatchWriteItem (p. 389).

The following Java code snippet demonstrates the preceding steps. The example creates a `BatchWriteItemRequest` to perform the following write operations:

- Put an item in the Forum table
- Put and delete an item from Thread table

The code then executes the `batchWriteItem` to perform a batch operation.

```
// Create a map for the requests in the batch
Map<String, List<WriteRequest>> requestItems = new HashMap<String, List<Write
Request>>();

// Create a PutRequest for a new Forum item
Map<String, AttributeValue> forumItem = new HashMap<String, AttributeValue>();
forumItem.put("Name", new AttributeValue().withS("Amazon ElastiCache"));
forumItem.put("Threads", new AttributeValue().withN("0"));

List<WriteRequest> forumList = new ArrayList<WriteRequest>();
forumList.add(new WriteRequest().withPutRequest(new PutRequest().withItem(foru
mItem)));
requestItems.put("Forum", forumList);

// Create a PutRequest for a new Thread item
Map<String, AttributeValue> threadItem = new HashMap<String, AttributeValue>();
threadItem.put("ForumName", new AttributeValue().withS("Amazon ElastiCache"));
threadItem.put("Subject", new AttributeValue().withS("ElastiCache Thread 1"));

List<WriteRequest> threadList = new ArrayList<WriteRequest>();
threadList.add(new WriteRequest().withPutRequest(new PutRequest().withItem(thread
```

```
Item)));

// Create a DeleteRequest for a Thread item
Key threadDeleteKey = new Key()
   .withHashKeyElement(new AttributeValue().withS("Some hash attribute value"))

   .withRangeKeyElement(new AttributeValue().withS("Some range attribute
value"));

threadList.add(new WriteRequest().withDeleteRequest(new Delete
Request().withKey(threadDeleteKey)));
requestItems.put("Thread", threadList);

// Code for checking unprocessed items is omitted in this example

BatchWriteItemResult result;
BatchWriteItemRequest batchWriteItemRequest = new BatchWriteItemRequest();

batchWriteItemRequest.withRequestItems(requestItems);
result = client.batchWriteItem(batchWriteItemRequest);
```

For a working example, see Example: Batch Write Operation Using the AWS SDK for Java Low-Level API for Amazon DynamoDB (p. 124).

Batch Get: Getting Multiple Items

The `AmazonDynamoDBClient.batchGetItem` method 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 commons steps that you follow to get multiple items using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class by providing your credentials.
2. Create an instance of the `BatchGetItemRequest` class that describes the table name, and a list of primary key values to retrieve. For the items that you are retrieving, you can optionally specify a list of attributes to retrieve.
3. Execute the `batchGetItem` method by providing the `BatchGetItemRequest` object 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. The `BatchGetItemRequest` class has a `withRequestItems` method, which takes a HashMap of table names and primary keys to retrieve.

```
Map<String, KeysAndAttributes> requestItems = new HashMap<String, KeysAndAttrib
utes>();
Key table1key1 = new Key().withHashKeyElement(new AttributeValue().withS("Amazon
 S3"));
Key table1key2 = new Key().withHashKeyElement(new AttributeValue().withS("Amazon
 DynamoDB"));
requestItems.put(table1Name,
        new KeysAndAttributes()
            .withKeys(table1key1, table1key2));

Key table2key1 = new Key()
  .withHashKeyElement(new AttributeValue().withS("Amazon DynamoDB"))
```

```
  .withRangeKeyElement(new AttributeValue().withS("DynamoDB Thread 1"));
Key table2key2 = new Key()
  .withHashKeyElement(new AttributeValue().withS("Amazon DynamoDB"))
  .withRangeKeyElement(new AttributeValue().withS("DynamoDB Thread 2"));
Key table2key3 = new Key()
  .withHashKeyElement(new AttributeValue().withS("Amazon S3"))
  .withRangeKeyElement(new AttributeValue().withS("S3 Thread 1"));

requestItems.put(table2Name,
        new KeysAndAttributes()
            .withKeys(table2key1, table2key2, table2key3));

BatchGetItemRequest batchGetItemRequest = new BatchGetItemRequest()
    .withRequestItems(requestItems);

BatchGetItemResult result = client.batchGetItem(batchGet2ItemRequest);

BatchResponse table1Results = result.getResponses().get(table1Name);
System.out.println("Items in table " + table1Name);
for (Map<String,AttributeValue> item : table1Results.getItems() ) {
    System.out.println(item);
}

BatchResponse table2Results = result.getResponses().get(table2Name);
System.out.println("Items in table " + table2Name);
for (Map<String,AttributeValue> item : table2Results.getItems() ) {
    System.out.println(item);
}
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `batchGetItem` method. For example, you can optionally specify a list of attributes to retrieve as shown in the following Java code snippet. The code snippet retrieves two items from the Forum table. It specifies a list of attributes to retrieve by using the `withAttributesToGet` method.

```
Map<String, KeysAndAttributes> requestItems = new HashMap<String, KeysAndAttrib
utes>();
Key table1key1 = new Key().withHashKeyElement(new AttributeValue().withS("Amazon
 S3"));
Key table1key2 = new Key().withHashKeyElement(new AttributeValue().withS("Amazon
 DynamoDB"));
requestItems.put(table1Name,
        new KeysAndAttributes()
            .withKeys(table1key1, table1key2)
            .withAttributesToGet("Threads"));

BatchGetItemRequest batchGetItemRequest = new BatchGetItemRequest()
    .withRequestItems(requestItems);

BatchGetItemResult result = client.batchGetItem(batchGetItemRequest);
```

For more information about the parameters and the API, see BatchGetItem (p. 385).

Updating an Item

You can use the `AmazonDynamoDBClient.updateItem` method 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 an `UpdateItemRequest` instance that describes the updates that you want to perform.

The `updateItem` method 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 `updateItems` 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 `AttributeAction.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 (Putting an Item (p. 105)) can also can perform an update. 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 so that any other existing attributes of that item remain unchanged.

The following are the commons steps that you follow to update an existing item using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` client by providing your security credentials.
2. Create an `UpdateItemRequest` instance by providing all the updates that you wish to perform.
   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 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.

```
Map<String, AttributeValueUpdate> updateItems = new HashMap<String, Attribute
ValueUpdate>();
Key key = new Key().withHashKeyElement(new AttributeValue().withN("101"));

// Add two new authors to the list.
updateItems.put("Authors",
  new AttributeValueUpdate()
    .withAction(AttributeAction.ADD)
    .withValue(new AttributeValue().withSS("AuthorYY", "AuthorZZ")));

// Reduce the price. To add or subtract a value,
// use ADD with a positive or negative number.
updateItems.put("Price",
```

```
  new AttributeValueUpdate()
    .withAction(AttributeAction.ADD)
    .withValue(new AttributeValue().withN("-1")));

// Delete the ISBN attribute.
updateItems.put("ISBN",
  new AttributeValueUpdate()
    .withAction(AttributeAction.DELETE));

UpdateItemRequest updateItemRequest = new UpdateItemRequest()
  .withTableName(tableName)
  .withKey(key).withReturnValues(ReturnValue.UPDATED_NEW)
  .withAttributeUpdates(updateItems);

UpdateItemResult result = client.updateItem(updateItemRequest);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `updateItem` method 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 Java SDK throws an `ConditionalCheckFailedException`. For example, the following Java code snippet conditionally updates a book item price to 25. It specifies the following optional parameters:

- A hash table of keys and `ExpectedAttributeValue` objects that set a condition that the price should be updated only if the existing price is 20.00.
- A `ReturnValue` enumeration value that specifies that the `updateItem` operation should return the updated item.

```
Map<String, AttributeValueUpdate> updateItems = new HashMap<String, Attribute
ValueUpdate>();
Map<String, ExpectedAttributeValue> expectedValues = new HashMap<String, Expec
tedAttributeValue>();

Key key = new Key().withHashKeyElement(new AttributeValue().withN("101"));
// Add two new authors to the list.
updateItems.put("Price",
        new AttributeValueUpdate()
            .withAction(AttributeAction.PUT)
            .withValue(new AttributeValue().withN("25.00")));

expectedValues.put("Price",
        new ExpectedAttributeValue()
            .withValue(new AttributeValue().withN("20.00")));

ReturnValue returnValue = ReturnValue.ALL_NEW;

UpdateItemRequest updateItemRequest = new UpdateItemRequest()
    .withTableName(tableName)
    .withKey(key)
    .withAttributeUpdates(updateItems)
    .withExpected(expectedValues)
    .withReturnValues(returnValues);

UpdateItemResult result = client.updateItem(updateItemRequest);
```

For more information about the parameters and the API, see UpdateItem (p. 433).

Deleting an Item

The `AmazonDynamoDBClient.deleteItem` method deletes an item from a table.

The following are the commons steps that you follow to delete an item using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` client by providing your security credentials.
2. Create a `DeleteItemRequest` instance by providing the name of the table from which you want to delete the item and the primary key of the item that you want to delete.
3. Execute the `deleteItem` method by providing the `DeleteItemRequest` object that you created in the preceding step.

```
DeleteItemRequest deleteItemRequest = new DeleteItemRequest()
    .withTableName(tableName)
    .withKey(new Key()
        .withHashKeyElement(new AttributeValue().withN("101")));

DeleteItemResult result = client.deleteItem(deleteItemRequest);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `DeleteItem` method. For example, the following Java code snippet specifies the following optional parameters:

- A hash table of keys and `ExpectedAttributeValues` objects that specify that the Book item in the ProductCatalog table be deleted only if the book is no longer in publication, that is, the InPublication attribute value is false. Boolean values are stored as numeric 0 and 1.
- A `ReturnValue` enumeration value to request that the `DeleteItem` method return the item that was deleted.

```
Map<String, ExpectedAttributeValue> expectedValues = new HashMap<String, Expec
tedAttributeValue>();
Key key = new Key().withHashKeyElement(new AttributeValue().withN("103"));

expectedValues.put("InPublication",
        new ExpectedAttributeValue()
            .withValue(new AttributeValue().withN("0"))); // Boolean stored as
 0 or 1.

ReturnValue returnValues = ReturnValue.ALL_OLD;

DeleteItemRequest deleteItemRequest = new DeleteItemRequest()
    .withTableName(tableName)
    .withKey(key)
    .withExpected(expectedValues)
    .withReturnValues(returnValues);

DeleteItemResult result = client.deleteItem(deleteItemRequest);
```

For more information about the parameters and the API, see DeleteItem (p. 399).

Additional AWS SDK for Java APIs

The examples in this section use AWS SDK for Java low-level API. Additionally, the SDK provides Object Persistence Model API that enable you to map your client-side classes to your Amazon DynamoDB tables. For more information, see Using the AWS SDK for Java with Amazon DynamoDB (p. 259).

Example: Put, Get, Update, and Delete an Item Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

The following Java 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 code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.io.IOException;

import java.util.Arrays;

import java.util.HashMap;

import java.util.Map;



import com.amazonaws.AmazonServiceException;

import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.model.AttributeAction;

import com.amazonaws.services.dynamodb.model.AttributeValue;

import com.amazonaws.services.dynamodb.model.AttributeValueUpdate;

import com.amazonaws.services.dynamodb.model.ConditionalCheckFailedException;

import com.amazonaws.services.dynamodb.model.DeleteItemRequest;

import com.amazonaws.services.dynamodb.model.DeleteItemResult;
```

```
import com.amazonaws.services.dynamodb.model.ExpectedAttributeValue;
import com.amazonaws.services.dynamodb.model.GetItemRequest;
import com.amazonaws.services.dynamodb.model.GetItemResult;
import com.amazonaws.services.dynamodb.model.Key;
import com.amazonaws.services.dynamodb.model.PutItemRequest;
import com.amazonaws.services.dynamodb.model.PutItemResult;
import com.amazonaws.services.dynamodb.model.ReturnValue;
import com.amazonaws.services.dynamodb.model.UpdateItemRequest;
import com.amazonaws.services.dynamodb.model.UpdateItemResult;


public class LowLevelItemCRUDExample {

    static AmazonDynamoDBClient client;
    static String tableName = "ProductCatalog";

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

        createClient();

        createItems();

        retrieveItem();

        // Perform various updates.
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

```

```
        // Delete the item.

        deleteItem();

    }


    private static void createClient() throws IOException {

        AWSCredentials credentials = new PropertiesCredentials(

                LowLevelItemCRUDExample.class.getResourceAsStream("AwsCreden
tials.properties"));


        client = new AmazonDynamoDBClient(credentials);


    }


    private static void createItems() {

        try {

            Map<String, AttributeValue> item1 = new HashMap<String, Attribute
Value>();

            item1.put("Id", new AttributeValue().withN("120"));

            item1.put("Title", new AttributeValue().withS("Book 120 Title"));

            item1.put("ISBN", new AttributeValue().withS("120-1111111111"));

            item1.put("Authors", new AttributeValue()

                .withSS(Arrays.asList("Author12", "Author22")));

            item1.put("Price", new AttributeValue().withN("20.00"));

            item1.put("Category", new AttributeValue().withS("Book"));

          item1.put("Dimensions", new AttributeValue().withS("8.5x11.0x.75"));

            item1.put("InPublication", new AttributeValue().withN("0")); //
false


            PutItemRequest putItemRequest1 = new PutItemRequest()

            .withTableName(tableName)
```

```
            .withItem(item1);
            PutItemResult result1 = client.putItem(putItemRequest1);


            Map<String, AttributeValue> item2 = new HashMap<String, Attribute
Value>();
            item2.put("Id", new AttributeValue().withN("121"));
            item2.put("Title", new AttributeValue().withS("Book 121 Title"));
            item2.put("ISBN", new AttributeValue().withS("121-1111111111"));
            item2.put("Price", new AttributeValue().withN("20.00"));
            item2.put("ProductCategory", new AttributeValue().withS("Book"));
            item2.put("Authors", new AttributeValue()
                .withSS(Arrays.asList("Author21", "Author22")));
          item1.put("Dimensions", new AttributeValue().withS("8.5x11.0x.75"));

            item1.put("InPublication", new AttributeValue().withN("1"));


            PutItemRequest putItemRequest2 = new PutItemRequest()
                .withTableName(tableName)
                .withItem(item2);
            PutItemResult result2 = client.putItem(putItemRequest2);
        } catch (AmazonServiceException ase) {
            System.err.println("Create items failed.");
        }
    }


    private static void retrieveItem() {
        try {


            GetItemRequest getItemRequest = new GetItemRequest()
                .withTableName(tableName)
```

```
                .withKey(new Key()
                    .withHashKeyElement(new AttributeValue().withN("120")))
                .withAttributesToGet(Arrays.asList("Id", "ISBN", "Title", "Au
thors"));

            GetItemResult result = client.getItem(getItemRequest);

            // Check the response.
            System.out.println("Printing item after retrieving it....");
            printItem(result.getItem());

        }  catch (AmazonServiceException ase) {
                    System.err.println("Failed to retrieve item in " + table
Name);
        }

    }

    private static void updateAddNewAttribute() {
        try {
            Map<String, AttributeValueUpdate> updateItems =
                new HashMap<String, AttributeValueUpdate>();

            Key key = new Key().withHashKeyElement(new Attribute
Value().withN("121"));
            updateItems.put("NewAttribute",
                    new AttributeValueUpdate()
                        .withValue(new AttributeValue().withS("Some Value")));

            ReturnValue returnValues = ReturnValue.ALL_NEW;
```

```
            UpdateItemRequest updateItemRequest = new UpdateItemRequest()
                .withTableName(tableName)
                .withKey(key)
                .withAttributeUpdates(updateItems)
                .withReturnValues(returnValues);

            UpdateItemResult result = client.updateItem(updateItemRequest);

            // Check the response.
            System.out.println("Printing item after adding new attribute...");

            printItem(result.getAttributes());

        }   catch (AmazonServiceException ase) {
                    System.err.println("Failed to add new attribute in " +
tableName);
        }
    }

    private static void updateMultipleAttributes() {
        try {
            Map<String, AttributeValueUpdate> updateItems =
                new HashMap<String, AttributeValueUpdate>();

            Key key = new Key().withHashKeyElement(new Attribute
Value().withN("120"));
            // Add two new authors to the list.
            updateItems.put("Authors",
                    new AttributeValueUpdate()
                        .withAction(AttributeAction.ADD)
```

```
                        .withValue(new AttributeValue().withSS("Author YY",
"Author ZZ")));

            // Add a new attribute.

            updateItems.put("NewAttribute",

                    new AttributeValueUpdate()

                        .withValue(new AttributeValue().withS("someValue")));


            ReturnValue returnValues = ReturnValue.ALL_NEW;


            UpdateItemRequest updateItemRequest = new UpdateItemRequest()

                .withTableName(tableName)

                .withKey(key)

                .withAttributeUpdates(updateItems)

                .withReturnValues(returnValues);


            UpdateItemResult result = client.updateItem(updateItemRequest);


            // Check the response.

            System.out.println("Printing item after multiple attribute up
date...");

            printItem(result.getAttributes());


        }   catch (AmazonServiceException ase) {

                    System.err.println("Failed to update multiple attributes
in " + tableName);

        }

    }


    private static void updateExistingAttributeConditionally() {

        try {
```

```
            Map<String, AttributeValueUpdate> updateItems = new HashMap<String,
 AttributeValueUpdate>();

            Map<String, ExpectedAttributeValue> expectedValues = new
HashMap<String, ExpectedAttributeValue>();



            Key key = new Key().withHashKeyElement(new Attribute
Value().withN("120"));

            // Add two new authors to the list.

            updateItems.put("Price",

                    new AttributeValueUpdate()

                        .withAction(AttributeAction.PUT)

                        .withValue(new AttributeValue().withN("25.00")));


           // This updates the price only if current price is 20.00.

            expectedValues.put("Price",

                    new ExpectedAttributeValue()

                        .withValue(new AttributeValue().withN("20.00")));


            ReturnValue returnValues = ReturnValue.ALL_NEW;


            UpdateItemRequest updateItemRequest = new UpdateItemRequest()

                .withTableName(tableName)

                .withKey(key)

                .withAttributeUpdates(updateItems)

                .withExpected(expectedValues)

                .withReturnValues(returnValues);



            UpdateItemResult result = client.updateItem(updateItemRequest);
```

```
            // Check the response.
            System.out.println("Printing item after conditional update to new
attribute...");
            printItem(result.getAttributes());
        } catch (ConditionalCheckFailedException cse) {
            // Reload object and retry code.
            System.err.println("Conditional check failed in " + tableName);
        } catch (AmazonServiceException ase) {
            System.err.println("Error updating item in " + tableName);
        }
    }



    private static void deleteItem() {
        try {

            Map<String, ExpectedAttributeValue> expectedValues = new
HashMap<String, ExpectedAttributeValue>();
            Key key = new Key().withHashKeyElement(new Attribute
Value().withN("120"));

            expectedValues.put("InPublication",
                    new ExpectedAttributeValue()
                      .withValue(new AttributeValue().withN("0"))); // Boolean
 stored as 0 or 1.

            ReturnValue returnValues = ReturnValue.ALL_OLD;

            DeleteItemRequest deleteItemRequest = new DeleteItemRequest()
                .withTableName(tableName)
                .withKey(key)
```

```
                .withExpected(expectedValues)
                .withReturnValues(returnValues);

            DeleteItemResult result = client.deleteItem(deleteItemRequest);

            // Check the response.
            System.out.println("Printing item that was deleted...");
            printItem(result.getAttributes());

        }  catch (AmazonServiceException ase) {
                                System.err.println("Failed to get item after
deletion " + tableName);
        }

    }

    private static void printItem(Map<String, AttributeValue> attributeList) {

        for (Map.Entry<String, AttributeValue> item : attributeList.entrySet())
 {
            String attributeName = item.getKey();
            AttributeValue value = item.getValue();
            System.out.println(attributeName + " " +
                    (value.getS() == null ? "" : "S=[" + value.getS() + "]") +

                    (value.getN() == null ? "" : "N=[" + value.getN() + "]") +

                   (value.getSS() == null ? "" : "SS=[" + value.getSS() + "]")
 +
                    (value.getNS() == null ? "" : "NS=[" + value.getNS() + "]
\n"));
```

```
        }

    }

}
```

Example: Batch Operations Using AWS SDK for Java Low-Level API for Amazon DynamoDB

Topics

Example: Batch Write Operation Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

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 Amazon 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 (p. 389). 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 Java Low-Level API (p. 450).

For step-by-step instructions to test the following sample, see Using the AWS SDK for Java with Amazon DynamoDB (p. 259).

```
import java.io.IOException;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashMap;

import java.util.List;
```

```
import java.util.Map;


import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodb.model.AttributeValue;
import com.amazonaws.services.dynamodb.model.BatchWriteItemRequest;
import com.amazonaws.services.dynamodb.model.BatchWriteItemResult;
import com.amazonaws.services.dynamodb.model.BatchWriteResponse;
import com.amazonaws.services.dynamodb.model.DeleteRequest;
import com.amazonaws.services.dynamodb.model.Key;
import com.amazonaws.services.dynamodb.model.PutRequest;
import com.amazonaws.services.dynamodb.model.WriteRequest;


public class LowLevelBatchWrite {


    static AmazonDynamoDBClient client;
    static String table1Name = "Forum";
    static String table2Name = "Thread";


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


        createClient();


        writeMultipleItemsBatchWrite();


    }
```

```
   private static void createClient() throws IOException {

        AWSCredentials credentials = new PropertiesCredentials(

              LowLevelBatchWrite.class.getResourceAsStream("AwsCredentials.prop
erties"));


        client = new AmazonDynamoDBClient(credentials);

    }



    private static void writeMultipleItemsBatchWrite() {

        try {


           // Create a map for the requests in the batch

           Map<String, List<WriteRequest>> requestItems = new HashMap<String,
List<WriteRequest>>();


           // Create a PutRequest for a new Forum item

           Map<String, AttributeValue> forumItem = new HashMap<String, Attrib
uteValue>();

           forumItem.put("Name", new AttributeValue().withS("Amazon ElasticC
ache"));

           forumItem.put("Threads", new AttributeValue().withN("0"));


           List<WriteRequest> forumList = new ArrayList<WriteRequest>();

          forumList.add(new WriteRequest().withPutRequest(new PutRequest().with
Item(forumItem)));

           requestItems.put(table1Name, forumList);


           // Create a PutRequest for a new Thread item

          Map<String, AttributeValue> threadItem = new HashMap<String, Attrib
uteValue>();

           threadItem.put("ForumName", new AttributeValue().withS("Amazon
ElastiCache"));
```

```
            threadItem.put("Subject", new AttributeValue().withS("ElasticCache
Thread 1"));

            threadItem.put("Message", new AttributeValue().withS("ElasticCache
Thread 1 message"));

            threadItem.put("KeywordTags", new AttributeValue().withSS(Arrays.as
List("cache", "in-memory")));



            List<WriteRequest> threadList = new ArrayList<WriteRequest>();

            threadList.add(new WriteRequest().withPutRequest(new PutRequest().with
Item(threadItem)));



            // Create a DeleteRequest for a Thread item

            Key threadDeleteKey = new Key()

                .withHashKeyElement(new AttributeValue().withS("Amazon S3"))

            .withRangeKeyElement(new AttributeValue().withS("S3 Thread 100"));



            threadList.add(new WriteRequest().withDeleteRequest(new Delete
Request().withKey(threadDeleteKey)));

            requestItems.put(table2Name, threadList);



            BatchWriteItemResult result;

            BatchWriteItemRequest batchWriteItemRequest = new BatchWriteItemRe
quest();

            do {

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



                batchWriteItemRequest.withRequestItems(requestItems);

                result = client.batchWriteItem(batchWriteItemRequest);



                // Print consumed capacity units

                for(Map.Entry<String, BatchWriteResponse> entry : result.getRe
sponses().entrySet()) {
```

```
                    String tableName = entry.getKey();

                  Double consumedCapacityUnits = entry.getValue().getConsumed
CapacityUnits();

                    System.out.println("Consumed capacity units for table " +
tableName + ": " + consumedCapacityUnits);

               }



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

               System.out.println("Unprocessed Put and Delete requests: \n" +
result.getUnprocessedItems());

               requestItems = result.getUnprocessedItems();

           } while (result.getUnprocessedItems().size() > 0);


        }  catch (AmazonServiceException ase) {

            System.err.println("Failed to retrieve items: " + ase);

            ase.printStackTrace(System.err);

        }



    }



}
```

Example: Batch Get Operation Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

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

import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodb.model.AttributeValue;
import com.amazonaws.services.dynamodb.model.BatchGetItemRequest;
import com.amazonaws.services.dynamodb.model.BatchGetItemResult;
import com.amazonaws.services.dynamodb.model.BatchResponse;
import com.amazonaws.services.dynamodb.model.Key;
import com.amazonaws.services.dynamodb.model.KeysAndAttributes;


public class LowLevelBatchGet {

    static AmazonDynamoDBClient client;
    static String table1Name = "Forum";
    static String table2Name = "Thread";

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

        createClient();

        retrieveMultipleItemsBatchGet();

    }
```

```
    private static void createClient() throws IOException {

        AWSCredentials credentials = new PropertiesCredentials(

              LowLevelBatchGet.class.getResourceAsStream("AwsCredentials.prop
erties"));


        client = new AmazonDynamoDBClient(credentials);


    }


    private static void retrieveMultipleItemsBatchGet() {

        try {


            Map<String, KeysAndAttributes> requestItems = new HashMap<String,
KeysAndAttributes>();

            Key table1key1 = new Key().withHashKeyElement(new Attribute
Value().withS("Amazon S3"));

            Key table1key2 = new Key().withHashKeyElement(new Attribute
Value().withS("Amazon DynamoDB"));

            requestItems.put(table1Name,

                    new KeysAndAttributes()

                        .withKeys(table1key1, table1key2));


            Key table2key1 = new Key()

                .withHashKeyElement(new AttributeValue().withS("Amazon Dy
namoDB"))

              .withRangeKeyElement(new AttributeValue().withS("DynamoDB Thread
 1"));

            Key table2key2 = new Key()

                .withHashKeyElement(new AttributeValue().withS("Amazon Dy
namoDB"))

              .withRangeKeyElement(new AttributeValue().withS("DynamoDB Thread
 2"));
```

```
            Key table2key3 = new Key()
                .withHashKeyElement(new AttributeValue().withS("Amazon S3"))
              .withRangeKeyElement(new AttributeValue().withS("S3 Thread 1"));

            requestItems.put(table2Name,
                    new KeysAndAttributes()
                        .withKeys(table2key1, table2key2, table2key3));


            BatchGetItemResult result;
          BatchGetItemRequest batchGetItemRequest = new BatchGetItemRequest();

            do {
                System.out.println("Making the request.");


                batchGetItemRequest.withRequestItems(requestItems);
                result = client.batchGetItem(batchGetItemRequest);


                BatchResponse table1Results = result.getRe
sponses().get(table1Name);
                if (table1Results != null){
                    System.out.println("Items in table " + table1Name);
                    for (Map<String,AttributeValue> item : table1Res
ults.getItems() ) {
                        printItem(item);
                    }
                }


                BatchResponse table2Results = result.getRe
sponses().get(table2Name);
                if (table2Results != null){
                    System.out.println("\nItems in table " + table2Name);
                    for (Map<String,AttributeValue> item : table2Res
```

```
ults.getItems() ) {
                        printItem(item);
                    }
                }

                // Check for unprocessed keys which could happen if you exceed
 provisioned
                // throughput or reach the limit on response size.
              for (Map.Entry<String,KeysAndAttributes> pair : result.getUnpro
cessedKeys().entrySet()) {
                   System.out.println("Unprocessed key pair: " + pair.getKey()
 + ", " + pair.getValue());
                }
                requestItems = result.getUnprocessedKeys();
            } while (result.getUnprocessedKeys().size() > 0);



        }  catch (AmazonServiceException ase) {
            System.err.println("Failed to retrieve items.");
        }

    }
    private static void printItem(Map<String, AttributeValue> attributeList) {

       for (Map.Entry<String, AttributeValue> item : attributeList.entrySet())
 {
            String attributeName = item.getKey();
            AttributeValue value = item.getValue();
            System.out.println(attributeName + " " +
                    (value.getS() == null ? "" : "S=[" + value.getS() + "]") +

                    (value.getN() == null ? "" : "N=[" + value.getN() + "]") +
```

```
                        (value.getSS() == null ? "" : "SS=[" + value.getSS() + "]")
 +
                         (value.getNS() == null ? "" : "NS=[" + value.getNS() + "]
\n"));
          }
     }
}
```

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

Topics

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 Amazon DynamoDB API (see API Reference for Amazon DynamoDB (p. 371)).

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`, `GeItem`, `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 by providing your credentials.
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.

```
client = new AmazonDynamoDBClient(credentials);
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:

- Expected parameter specifies a condition that the item 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 = "202" }},
                   { "Title", new AttributeValue { S = "Book 202  Title" }},
                  { "ISBN", new AttributeValue { S = "2222-2222-2222-2222" }},

                   { "Authors",
                     new AttributeValue { SS = new List<string>{"Author3"}}}
               },
   // Optional parameters.
   Expected = new Dictionary<string, ExpectedAttributeValue>()
   {
      { "ISBN",  new ExpectedAttributeValue { Value = new AttributeValue { S =
 "111-1111111111" }}}
   },
    ReturnValues = "ALL_OLD"
 };
var response = client.PutItem(request);
```

For more information about the parameters and the API, see PutItem (p. 413)..

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` by providing your credentials.
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.

```
client = new AmazonDynamoDBClient(credentials);
string tableName = "ProductCatalog";

var request = new GetItemRequest
 {
   TableName = tableName,
   Key = new Key { HashKeyElement = 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:

- `AttributesToGet` parameter to specify a list of attributes to retrieve.
- `ConsistentRead` parameter to request the latest item data. To learn more about consistent read, see Data Read and Consistency Considerations (p. 7).

```
client = new AmazonDynamoDBClient(credentials);
string tableName = "ProductCatalog";

var request = new GetItemRequest
 {
   TableName = tableName,
   Key = new Key { HashKeyElement = new AttributeValue { N = "202" } },
   // Optional parameters.
   AttributesToGet = new List<string>() { "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 (p. 409).

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. 134). 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` client by providing your security credentials.
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.

```
client = new AmazonDynamoDBClient(credentials);
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
   TableName = tableName,
   Key = new Key { HashKeyElement = new AttributeValue { N = "201" } },
   AttributeUpdates = new Dictionary<string, AttributeValueUpdate>()
   {
     // Add two new authors to the list.
     { "Authors",
       new AttributeValueUpdate
          {
              Action="ADD",
              Value = new AttributeValue{SS = { "Author YY", "Author ZZ" }}
           }
      },
      // Reduce the price. To add or subtract a value,
      // use ADD with a positive or negative number.
     { "Price",
       new AttributeValueUpdate
          {
              Action="ADD",
              Value = new AttributeValue{N = "-1"}
           }
      },
      // Add a new attribute.
     { "NewAttribute",
       new AttributeValueUpdate  { Value = new AttributeValue{S = "someValue"
} } },
      // Delete the existing ISBN attribute.
     { "ISBN", new AttributeValueUpdate  { Action="DELETE" } }
   }
```

```
};
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:

- `Expected` parameter to set a condition that the price be updated only if the existing price is 20.00.
- `ReturnValues` parameter to request the updated item in the response.

```
client = new AmazonDynamoDBClient(credentials);
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
  Key = new Key { HashKeyElement = new AttributeValue { N = "201" } },
  AttributeUpdates = new Dictionary<string, AttributeValueUpdate>()
  {
    { "Price",
      new AttributeValueUpdate
      { Action = "PUT", Value = new AttributeValue{ N = "22.00"} }
     } // PUT = replace existing.
  },
  // Update price only if the current price is 20.00.
  Expected = new Dictionary<string, ExpectedAttributeValue>()
  {
     {
       "Price",
       new ExpectedAttributeValue
       { Value = new AttributeValue{ N = "20.00"} }
     }
  },
  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 (p. 433).

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` client by providing your security credentials.
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.

```
client = new AmazonDynamoDBClient(credentials);
string tableName = "ProductCatalog";

var request = new DeleteItemRequest
 {
   TableName = tableName,
   Key = new Key { HashKeyElement = 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:

- `Expected` parameter sets a condition that the book item be deleted only if it is no longer in publication, that is, the InPublication attribute value is false (Boolean values are stored as numeric 0 and 1).
- `ReturnValues` parameter to request the deleted item in the response.

```
var request = new DeleteItemRequest
{
  TableName = tableName,
  Key = new Key { HashKeyElement = new AttributeValue { N = "201" } },
  // Optional parameters.
  ReturnValues = "ALL_OLD",
  Expected = new Dictionary<string, ExpectedAttributeValue>()
  {
     {
       "InPublication",
       new ExpectedAttributeValue
       { Value = new AttributeValue{ N = "0"} } // boolean stored as 0 and 1.
     }
  }
};
var response = client.DeleteItem(request);
```

For more information about the parameters and the API, see DeleteItem (p. 399).

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 by providing your credentials.
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, Amazon DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, Amazon DynamoDB rejects the request. For more information, see BatchWriteItem (p. 389).

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 the `BatchWriteItem` to perform a batch operation.

```
client = new AmazonDynamoDBClient(credentials);

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 = "S3 forum" } },
                  { "Threads", new AttributeValue { N = "0" }}
                }
             }
          }
        }
      } ,
      {
        table2Name, new List<WriteRequest>
        {
          new WriteRequest
          {
            PutRequest = new PutRequest
            {
               Item = new Dictionary<string,AttributeValue>
               {
                 { "ForumName", new AttributeValue { S = "S3 forum" } },
                 { "Subject", new AttributeValue { S = "My sample question" }
},
                 { "Message", new AttributeValue { S = "Message Text." } },
                 { "KeywordTags", new AttributeValue { SS = new List<string> {
 "S3", "Bucket" }  } }
               }
            }
          },
          new WriteRequest
          {
             DeleteRequest = new DeleteRequest
             {
                Key = new Key
                {
```

```
                   HashKeyElement = new AttributeValue { S = "Some hash attr 
value" },
                   RangeKeyElement = new AttributeValue { S = "Some range attr
 value" }
                }
             }
          }
        }
      }
    }
 };
response = client.BatchWriteItem(request);
```

For a working example, see Example: Batch Operations Using AWS SDK for .NET Low-Level API for Amazon DynamoDB (p. 150).

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 by providing your credentials.
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.

```
client = new AmazonDynamoDBClient(credentials);

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchGetItemRequest
{
  RequestItems = new Dictionary<string, KeysAndAttributes>()
  {
    { table1Name,
      new KeysAndAttributes
      {
        Keys = new List<Key>()
        {
```

```
          new Key { HashKeyElement = new AttributeValue { N = "101" } },
          new Key { HashKeyElement = new AttributeValue { N = "102" } }
        }
      }
    },
    {
      table2Name,
      new KeysAndAttributes
      {
        Keys = new List<Key>()
        {
          new Key { HashKeyElement = new AttributeValue
                                 { S = "DynamoDB Thread 1"} },
          new Key { HashKeyElement = new AttributeValue
                                 { S = "DynamoDB Thread 2"} },
          new Key { HashKeyElement = new AttributeValue
                                 { S = "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:

- `AttributesToGet` parameter to specify a list of attributes to retrieve.

```
client = new AmazonDynamoDBClient(credentials);

string table1Name = "Forum";

var request = new BatchGetItemRequest
{
  RequestItems = new Dictionary<string, KeysAndAttributes>()
  {
    { table1Name,
      new KeysAndAttributes
      {
        Keys = new List<Key>()
        {
          new Key { HashKeyElement = new AttributeValue { N = "101" } },
          new Key { HashKeyElement = new AttributeValue { N = "102" } }
        }
      },
      // Optional list of attributes.
      AttributesToGet = new List<string>{"Title"}
    }
  }
};

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

For more information about the parameters and the API, see BatchGetItem (p. 385).

Additional AWS SDK for .NET APIs

The examples in this section use AWS SDK for .NET low-level API. The SDK also supports helper API that further simplify your application development. Additionally, the SDK provides Object Persistence Model API that enable you to map your client-side classes to your Amazon DynamoDB tables. For more information, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

Example: Put, Get, Update, and Delete an Item Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB

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 Low-Level API (p. 468).

For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.Model;
```

```
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace Amazon.DynamoDB.Documentation
{
  class Program
  {
    private static string tableName = "ProductCatalog";
    private static AmazonDynamoDBClient client;

    static void Main(string[] args)
    {
      client = new AmazonDynamoDBClient();

      try
      {
        CreateItem();
        RetrieveItem();

        // Perform various updates.
        UpdateMultipleAttributes();
        UpdateExistingAttributeConditionally();

        // Delete item.
        DeleteItem();
        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()
    {
      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 { N = "0" } }// 0 = false.

          }
      };
      client.PutItem(request);
    }

    private static void RetrieveItem()
    {
```

```
      var request = new GetItemRequest
      {
        TableName = tableName,
        Key = new Key { HashKeyElement = new AttributeValue { N = "1000" } },
       AttributesToGet = new List<string>() { "Id", "ISBN", "Title", "Authors"
},
        ConsistentRead = true
      };
      var response = client.GetItem(request);

      // Check the response.
      var result = response.GetItemResult;
      var attributeList = result.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 Key { HashKeyElement = new AttributeValue { N = "1000" } },
        AttributeUpdates = new Dictionary<string, AttributeValueUpdate>()
        {
          // Adding two new authors to the list.
          { "Authors",
            new AttributeValueUpdate
               {
                   Action="ADD",
```

```
                   Value = new AttributeValue{SS = { "Author YY", "Author ZZ"
}}

                }

           },

           // Adding a new attribute.

          { "NewAttribute",

            new AttributeValueUpdate  { Value = new AttributeValue{S = "New
Value" } } },

           // Deleting ISBN attribute.

          { "ISBN", new AttributeValueUpdate  { Action="DELETE" } }

        },

        TableName = tableName,

       ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.

      };

      var response = client.UpdateItem(request);


      // Check the response.

      var result = response.UpdateItemResult;

     var attributeList = result.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 Key { HashKeyElement = new AttributeValue { N = "1000" } },
        AttributeUpdates = new Dictionary<string, AttributeValueUpdate>()
        {
          { "Price",
            new AttributeValueUpdate
            { Action = "PUT", Value = new AttributeValue{ N = "22.00"} }
           } // PUT = replace existing.
        },
        Expected = new Dictionary<string, ExpectedAttributeValue>()
        {
          // This updates price only if current price is 20.00.
           {
             "Price",
             new ExpectedAttributeValue
             { Value = new AttributeValue{ N = "20.00"} }
           }
        }
         ,
        TableName = tableName,
       ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.

      };
      var response = client.UpdateItem(request);


      // Check the response.
      var result = response.UpdateItemResult;
     var attributeList = result.Attributes; // attribute list in the response.

    Console.WriteLine("\nPrinting item after updating price value conditionally
............");
```

```
      PrintItem(attributeList);
    }

    private static void DeleteItem()
    {
      var request = new DeleteItemRequest
      {
        TableName = tableName,
        Key = new Key { HashKeyElement = new AttributeValue { N = "1000" } },
        // Optional parameters.
        ReturnValues = "ALL_OLD",
        Expected = new Dictionary<string, ExpectedAttributeValue>()
        {
           {
             "InPublication",
             new ExpectedAttributeValue
             { Value = new AttributeValue{ N = "0"} } // boolean stored as 0
and 1.
           }
        }
      };
      var response = client.DeleteItem(request);

      // Check the response.
      var result = response.DeleteItemResult;
      var attributeList = result.Attributes; // Attribute list in the response.

      // Print item.
      Console.WriteLine("\nPrinting item that is just deleted ...........");
      PrintItem(attributeList);
```

```
    }


   private static void PrintItem(Dictionary<string, AttributeValue> attributeL
ist)
    {
      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.ToAr
ray()) + "]") +
            (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToAr
ray()) + "]")
        );
      }
      Console.WriteLine("************************************************");
    }
  }
}
```

Example: Batch Operations Using AWS SDK for .NET Low-Level API for Amazon DynamoDB

Topics

Example: Batch Write Operation Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB

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 Amazon 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 (p. 389). 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 Low-Level API (p. 468).

For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.Model;

using Amazon.Runtime;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static string table1Name = "Forum";

    private static string table2Name = "Thread";

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      client = new AmazonDynamoDBClient();
```

```
    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
    {
      RequestItems = new Dictionary<string, List<WriteRequest>>
       {
         {
           table1Name, new List<WriteRequest>
           {
             new WriteRequest
             {
                PutRequest = new PutRequest
                {
                   Item = new Dictionary<string,AttributeValue>
                   {
                     { "Name", new AttributeValue { S = "S3 forum" } },
                     { "Threads", new AttributeValue { N = "0" }}
```

```
                    }
                 }
              }
            }
          } ,
          {
            table2Name, new List<WriteRequest>
            {
              new WriteRequest
              {
                 PutRequest = new PutRequest
                 {
                    Item = new Dictionary<string,AttributeValue>
                    {
                      { "ForumName", new AttributeValue { S = "S3 forum" } },

                    { "Subject", new AttributeValue { S = "My sample question"
 } },
                      { "Message", new AttributeValue { S = "Message Text." }
 },
                      { "KeywordTags", new AttributeValue { SS = new
List<string> { "S3", "Bucket" }  } }
                    }
                 }
              },
              new WriteRequest
              {
              // For the operation to delete an item, if you provide a primary
 key value
                // that does not exist in the table, there is no error, it is
 just a no-op.
                 DeleteRequest = new DeleteRequest
```

```
                  {
                     Key = new Key
                     {
                        HashKeyElement = new AttributeValue { S = "Some hash
attr value" },
                        RangeKeyElement = new AttributeValue { S = "Some range
 attr value" }
                     }
                  }
               }
             }
           }
         }
      };

      CallBatchWriteTillCompletion(request);
    }

    private static void CallBatchWriteTillCompletion(BatchWriteItemRequest re
quest)
    {
      BatchWriteItemResponse response;

      int callCount = 0;
      do
      {
        Console.WriteLine("Making request");
        response = client.BatchWriteItem(request);
        callCount++;
```

```
        // Check the response.
        var result = response.BatchWriteItemResult;
        var responses = result.Responses;
        var unprocessed = result.UnprocessedItems;

        Console.WriteLine("Resonses");
        foreach (var resp in responses)
        {
         Console.WriteLine("{0} - {1}", resp.Key, resp.Value.ConsumedCapacity
Units);
        }

        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 unprocssed items.
        request.RequestItems = unprocessed;

      } while (response.BatchWriteItemResult.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 for Amazon DynamoDB

The following C# code example uses the `BatchGetItem` method to retrieve multiple items from the Forum and the Thread tables. The `BatchGetItemRequest` specifies the table names and a list of primary keys for each table. The example processes the response by printing the items retrieved.

If you followed the 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 Low-Level API (p. 468).

For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.Model;

using Amazon.Runtime;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static string table1Name = "Forum";

    private static string table2Name = "Thread";

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      client = new AmazonDynamoDBClient();


      try

      {

        RetrieveMultipleItemsBatchGet();


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

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

    private static void RetrieveMultipleItemsBatchGet()
    {
      var request = new BatchGetItemRequest
      {
        RequestItems = new Dictionary<string, KeysAndAttributes>()
        {
          { table1Name,
            new KeysAndAttributes
            {
              Keys = new List<Key>()
              {
                new Key { HashKeyElement = new AttributeValue { S = "Amazon
DynamoDB" } },
               new Key { HashKeyElement = new AttributeValue { S = "Amazon S3"
 } }
              }
            }
          },
          {
            table2Name,
            new KeysAndAttributes
            {
              Keys = new List<Key>()
              {
```

```
                new Key { HashKeyElement = new AttributeValue { S = "Amazon
DynamoDB"},
                          RangeKeyElement = new AttributeValue { S = "DynamoDB
 Thread 1" }},
                new Key { HashKeyElement = new AttributeValue { S = "Amazon
DynamoDB"},
                          RangeKeyElement = new AttributeValue { S = "DynamoDB
 Thread 2" }},
                new Key { HashKeyElement = new AttributeValue { S = "Amazon
S3"},
                         RangeKeyElement = new AttributeValue { S = "S3 Thread
 1" }}
              }
            }
         }
       }
      };

      BatchGetItemResponse response;
      do
      {
        Console.WriteLine("Making request");
        response = client.BatchGetItem(request);

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

        foreach (var tableResponse in responses)
        {
          var tableResults = tableResponse.Value;
         Console.WriteLine("Items retrieved from table {0}", tableResponse.Key);
```

```
          foreach (var item1 in tableResults.Items)

          {

            PrintItem(item1);

          }

        }



       // Any unprocessed keys? could happen if you exceed ProvisionedThroughput
 or some other errror.

        Dictionary<string, KeysAndAttributes> unprocessedKeys = result.Unpro
cessedKeys;

        foreach (var unprocessedTableKeys in unprocessedKeys)

        {

          // Print table name.

          Console.WriteLine(unprocessedTableKeys.Key);

          // Print unprocessed primary keys.

          foreach (var keys in unprocessedTableKeys.Value.Keys)

          {

            Console.WriteLine(

              (keys.HashKeyElement.S == null ? "" : keys.HashKeyElement.S) +

              (keys.HashKeyElement.N == null ? "" : keys.HashKeyElement.N) +

              "\t" +

              (keys.RangeKeyElement.S == null ? "" : keys.RangeKeyElement.S) +


              (keys.RangeKeyElement.N == null ? "" : keys.RangeKeyElement.N));


          }

        }



        request.RequestItems = unprocessedKeys;

      } while (response.BatchGetItemResult.UnprocessedKeys.Count > 0);
```

```
    }


    private static void PrintItem(Dictionary<string, AttributeValue> attributeL
ist)
    {
      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.ToAr
ray()) + "]") +
            (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToAr
ray()) + "]")
        );
      }
      Console.WriteLine("*************************************************");
    }
  }
}
```

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

Topics

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 Amazon DynamoDB API. For more information, see API Reference for Amazon DynamoDB (p. 371).

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

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

Putting an Item

The PHP `put_item` 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 `update_item` function. For more information, see Updating an Item (p. 167).

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

1. Create an instance of the `AmazonDynamoDB` class (the client).
2. Provide the parameters for the `put_item` 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 `$put_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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$put_response = $dynamodb->put_item(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'       => array( AmazonDynamoDB::TYPE_NUMBER      => '104'      ),
```

```
 // Primary Key
        'Title'    => array( AmazonDynamoDB::TYPE_STRING      => 'Book 104
Title' ),
       'ISBN'     => array( AmazonDynamoDB::TYPE_STRING     => '111-1111111111'
 ),
        'Price'    => array( AmazonDynamoDB::TYPE_NUMBER      => '25' ),
       'Authors'  => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS => array('Au
thor1', 'Author2') )
  )
));

// status code 200 indicates success
print_r($put_response);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters to the `put_item` 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 `put_item`:

- An `ExpectedAttributeValue` 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 `RETURN_ALL_OLD` enumeration 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.

```
// instantiate the class
$dynamodb = new AmazonDynamoDB();

$put_response = $dynamodb->put_item(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'       => array( AmazonDynamoDB::TYPE_NUMBER => '103'      ), //
Primary Key
        'Title'    => array( AmazonDynamoDB::TYPE_STRING => 'Book 103 Title'
),
        'ISBN'     => array( AmazonDynamoDB::TYPE_STRING => '333-3333333333'
),
        'Price'    => array( AmazonDynamoDB::TYPE_NUMBER => '2000' ),
       'Authors'  => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS => array('Au
thor1', 'Author2', 'Author3') )
    ),
// Optional parameters Expected and ReturnValue.
    'Expected' => array(
        'ISBN'   => array(
            'Value' => array( AmazonDynamoDB::TYPE_STRING => '333-3333333333'
 ),
  )
 ),
    'ReturnValues' => AmazonDynamoDB::RETURN_ALL_OLD
));

// status code 200 indicates success
```

```
print_r($put_response);

?>
```

For more information, see PutItem (p. 413).

Getting an Item

The AWS SDK for PHP `get_item` function retrieves a single item. To retrieve multiple items, you can use the `batch_get_item` method (see Batch Get: Getting Multiple Items (p. 165)).

The following are the steps to retrieve an item.

1. Create an instance of the `AmazonDynamoDB` class (the client).
2. Provide the parameters for the `get_item` 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 `$get_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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$get_response = $dynamodb->get_item(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'HashKeyElement' => array( AmazonDynamoDB::TYPE_NUMBER => '104' )
    )
));

// status code 200 indicates success
print_r($get_response);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `get_item` function. For example, the following PHP code snippet uses an optional method to retrieve only a specific list of attributes, and requests a strictly 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 strictly consistent read value. Read results are eventually consistent by default. You can request read results to be strictly consistent. To learn more about consistent reads, see Data Read and Consistency Considerations (p. 7).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$get_response = $dynamodb->get_item(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'HashKeyElement' => array( AmazonDynamoDB::TYPE_NUMBER => '104' ),
    'ConsistentRead' => 'true',
    'AttributesToGet' => array( 'Id', 'Authors' )
    )
));

// status code 200 indicates success
print_r($get_response);
```

For more information about the parameters and the API, see GetItem (p. 409).

Batch Write: Putting and Deleting Multiple Items

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

The following are the common steps that you follow to get multiple items.

1. Create an instance of the `AmazonDynamoDB` class (the client).
2. Execute the `batch_write_item` 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 `batch_write_item` uses syntax required by the underlying Amazon DynamoDB API. For more information, see BatchWriteItem (p. 389).

Note

This code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
$dynamodb = new AmazonDynamoDB();
```

```
$table1_name = 'Forum';
$table2_name = 'Thread';

$response = $dynamodb->batch_write_item(array(
      'RequestItems' => array(
          $table1_name => array(
              array(
                  'PutRequest' => array(
                      'Item' => $dynamodb->attributes(array(
                          'Name'    => 'S3 Forum',
                          'Threads' => 0
                      ))
                  )
              )
          ),
           $table2_name => array(
              array(
                  'PutRequest' => array(
                      'Item' => $dynamodb->attributes(array(
                          'ForumName'    => 'S3 Forum',
                          'Subject' => 'My sample question',
                          'Message'=> 'Message Text.',
                          'KeywordTags'=> array('S3', 'Bucket')
                      ))
                  )
              ),
              array(
                  'DeleteRequest' => array(
                      'Key' => $dynamodb->attributes(array(
                          'HashKeyElement' =>'Some hash value',
                          'RangeKeyElement' => 'Some range key'
                      ))
                  )
              )
           )
      )
));
```

Batch Get: Getting Multiple Items

The AWS SDK for PHP `batch_get_item` function enables you to retrieve multiple items from one or more tables. To retrieve a single item, you can use the `get_item` method.

The following are the common steps that you follow to get multiple items.

1. Create an instance of the `AmazonDynamoDB` class (the client).
2. Provide the parameters for the `batch_get_item` 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 `$batch_get_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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

// use the values for the variables $seven_days_ago and $twenty_one_days_ago
// from the loading data examples for Getting Started or from the loading data
// example for PHP in the Appendix

$batch_get_response = $dynamodb->batch_get_item(array(
    'RequestItems' => array(
        'Forum' => array(
            'Keys' => array(
                array( // Key #2
                    'HashKeyElement'  => array( AmazonDynamoDB::TYPE_STRING =>
 'Amazon DynamoDB' )
                )
            )
        ),
        'Reply' => array(
            'Keys' => array(
                array( // Key #1
                    'HashKeyElement'  => array( AmazonDynamoDB::TYPE_STRING =>
 'Amazon DynamoDB#DynamoDB Thread 2'),
                    'RangeKeyElement' => array( AmazonDynamoDB::TYPE_STRING =>
 $seven_days_ago                    ),
                ),
                array( // Key #2
                    'HashKeyElement'  => array( AmazonDynamoDB::TYPE_STRING =>
 'Amazon DynamoDB#DynamoDB Thread 2'),
                    'RangeKeyElement' => array( AmazonDynamoDB::TYPE_STRING =>
 $twenty_one_days_ago               ),
                ),
            )
        )
    )
));

// status code 200 indicates success
print_r($batch_get_response);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `batch_get_item` 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 `AttributesToGet` parameter to retrieve the count of threads in each table :

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$batch_get_response = $dynamodb->batch_get_item(array(
    'RequestItems' => array(
        'Forum' => array(
        'Keys' => array(
            array( // Key #1
              'HashKeyElement' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon
 S3'      )
            ),
            array( // Key #2
              'HashKeyElement' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon
 DynamoDB' )
            )
        ),
        'AttributesToGet' =>  array ('Threads')
        ),
    )
));

// Success?
print_r($batch_get_response);
```

For more information about the parameters and the API, see BatchGetItem (p. 385).

Updating an Item

Use the `update_item` function to update existing attribute values, add new attributes to the existing collection, or delete attributes from the existing collection.

The `update_item` function uses the following guidelines:

- If an item does not exist, the `update_item` function adds a new item using the primary key that is specified in the input.
- If an item exists, the `update_item` 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 `ACTION_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 `put_item` function (Putting an Item (p. 161)) also updates items. For example, if you use `put_item` 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 `put_item` 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 `AmazonDynamoDB` class (the client).

2. Provide the parameters for the `update_item` operation to the client instance as an *`AttributeUpdates`* array.

   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 `$update_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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$update_response = $dynamodb->update_item(array(
    'TableName' => 'ProductCatalog',
        'Key' => array(
            'HashKeyElement' => array(
                AmazonDynamoDB::TYPE_NUMBER => '201'
            )
        ),
        'AttributeUpdates' => array(
            'Authors' => array(
                'Action' => AmazonDynamoDB::ACTION_PUT,
                'Value' => array(
                   AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS => array('Author YY',
 'Author ZZ')
                )
            ),
            // Reduce the price. To add or subtract a value,
            // use ADD with a positive or negative number.
            'Price' => array(
                'Action' => AmazonDynamoDB::ACTION_ADD,
                'Value' => array(
                    AmazonDynamoDB::TYPE_NUMBER => '-1'
                )
            ),
            'ISBN' => array(
                'Action' => AmazonDynamoDB::ACTION_DELETE
            )
        )
));

// status code 200 indicates success
print_r($update_response);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `update_item` 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 an `ConditionalCheckFailedException`. For example, the following PHP code snippet conditionally updates a book item price to 25. It specifies the following optional parameters:

- An *`Expected`* parameter that sets the condition that the price should be updated only if the existing price is 20.00.
- A `RETURN_ALL_NEW` enumeration value for the *`ReturnValues`* parameter that specifies the response should include all of the item's current attribute values after the update.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$update_response = $dynamodb->update_item(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'HashKeyElement' => array(
            AmazonDynamoDB::TYPE_NUMBER => '201'
        )
    ),
    'Expected' => array(
        'Price' => array( 'Value' => array (AmazonDynamoDB::TYPE_NUMBER =>
'20.00' ) )
        ),
    'AttributeUpdates' => array(
        'Price' => array(
            'Action' => AmazonDynamoDB::ACTION_PUT,
            'Value' => array(
                AmazonDynamoDB::TYPE_STRING => '22.00'
            )
        )
    ),
    'ReturnValues' => AmazonDynamoDB::RETURN_ALL_NEW
));

// status code 200 indicates success
print_r($update_response);
```

For more information about the parameters and the API, see UpdateItem (p. 433).

Deleting an Item

The `delete_item` 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 `AmazonDynamoDB` class (the client).
2. Provide the parameters for the `delete_item` 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 `$delete_response` to use in your application.

Note

This code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$delete_response = $dynamodb->delete_item(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'HashKeyElement' => array(
            AmazonDynamoDB::TYPE_NUMBER => '101'
        )
    )
));

// status code 200 indicates success
print_r($delete_response);
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `delete_item` 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 value is "false". Boolean values are stored as numeric 0 and 1.
- 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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$delete_response = $dynamodb->delete_item(array(
    'TableName' => 'ProductCatalog',
    'Key' => array(
        'HashKeyElement' => array(
            AmazonDynamoDB::TYPE_NUMBER => '103'
        )
    ),
    'Expected' => array(
       'InPublication' => array( 'Value' => array (AmazonDynamoDB::TYPE_NUMBER
 => '0' ) )
        ),
    'ReturnValues' => AmazonDynamoDB::RETURN_ALL_OLD
));
```

```
// status code 200 indicates success
print_r($delete_response);
```

For more information about the parameters and the API, see DeleteItem (p. 399).

Example: Put, Get, Update, and Delete an Item Using the AWS SDK for PHP Low-Level API for Amazon DynamoDB

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 for Amazon DynamoDB (p. 369).

```
<?php

// If necessary, reference the sdk.class.php file.

// For example, the following line assumes the sdk.class.php file is

// in an sdk sub-directory relative to this file

require_once dirname(__FILE__) . '/sdk/sdk.class.php';


// Instantiate the class

$dynamodb = new AmazonDynamoDB();


$table_name = 'ExampleTable2';

$current_time = '1326864974';


####################################################################

# Create a new DynamoDB table


$response = $dynamodb->create_table(array(

    'TableName' => $table_name,
```

```
    'KeySchema' => array(
        'HashKeyElement' => array(
            'AttributeName' => 'id',
            'AttributeType' => AmazonDynamoDB::TYPE_NUMBER
        ),
        'RangeKeyElement' => array(
            'AttributeName' => 'date',
            'AttributeType' => AmazonDynamoDB::TYPE_NUMBER
        )
    ),
    'ProvisionedThroughput' => array(
        'ReadCapacityUnits' => 8,
        'WriteCapacityUnits' => 5
    )
));

// Check for success...
if ($response->isOK())
{
    echo '# Kicked off the creation of the DynamoDB table...' . PHP_EOL;
}
    else
{
    print_r($response);
}

####################################################################
# Sleep and poll until the table has been created
```

```
$count = 0;

do {

    sleep(1);

    $count++;



    $response = $dynamodb->describe_table(array(

        'TableName' => $table_name

    ));

}

while ((string) $response->body->Table->TableStatus !== 'ACTIVE');



echo "The table \"${table_name}\" has been created (slept ${count} seconds)."
. PHP_EOL;



####################################################################

# Adding data to the table



echo PHP_EOL . PHP_EOL;

echo "# Adding data to the table..." . PHP_EOL;



// Set up batch requests

$queue = new CFBatchRequest();

$queue->use_credentials($dynamodb->credentials);



// Add items to the batch

$dynamodb->batch($queue)->put_item(array(

    'TableName' => $table_name,

    'Item' => array(

        'id' => array( AmazonDynamoDB::TYPE_NUMBER => '1' ), // Primary (Hash)
 Key
```

```
        'date' => array( AmazonDynamoDB::TYPE_NUMBER => $current_time ), //
Range Key
        'val1' => array( AmazonDynamoDB::TYPE_STRING => 'value1' ),
        'val2' => array( AmazonDynamoDB::TYPE_STRING => 'value2' ),
        'val3' => array( AmazonDynamoDB::TYPE_STRING => 'value3' )
    )
));


$dynamodb->batch($queue)->put_item(array(
    'TableName' => $table_name,
    'Item' => array(
        'id' => array( AmazonDynamoDB::TYPE_NUMBER => '2' ), // Primary (Hash)
 Key
        'date' => array( AmazonDynamoDB::TYPE_NUMBER => $current_time ), //
Range Key
        'val1' => array( AmazonDynamoDB::TYPE_STRING => 'value1' ),
        'val2' => array( AmazonDynamoDB::TYPE_STRING => 'value2' ),
        'val3' => array( AmazonDynamoDB::TYPE_STRING => 'value3' )
    )
));


$dynamodb->batch($queue)->put_item(array(
    'TableName' => $table_name,
    'Item' => array(
        'id' => array( AmazonDynamoDB::TYPE_NUMBER => '3' ), // Primary (Hash)
 Key
        'date' => array( AmazonDynamoDB::TYPE_NUMBER => $current_time ), //
Range Key
        'val1' => array( AmazonDynamoDB::TYPE_STRING => 'value1' ),
        'val2' => array( AmazonDynamoDB::TYPE_STRING => 'value2' ),
        'val3' => array( AmazonDynamoDB::TYPE_STRING => 'value3' )
    )
```

```
));


// Execute the batch of requests in parallel

$responses = $dynamodb->batch($queue)->send();


// Check for success...

if ($responses->areOK())

{

    echo "The data has been added to the table." . PHP_EOL;

}

    else

{

    print_r($responses);

}


####################################################################

# Getting an item


echo PHP_EOL . PHP_EOL;

echo "# Getting an item from the table..." . PHP_EOL;


// Get an item

$response = $dynamodb->get_item(array(

    'TableName' => $table_name,

    'Key' => array(

        'HashKeyElement' => array( // "id" column

        AmazonDynamoDB::TYPE_NUMBER => '1'

        ),

        'RangeKeyElement' => array( // "date" column
```

```
        AmazonDynamoDB::TYPE_NUMBER => $current_time
        )
    ),
    'AttributesToGet' => 'val3'
));

// Check for success...
if ($response->isOK())
{
    print_r($response);
}

####################################################################
# Updating an item

echo PHP_EOL . PHP_EOL;
echo "# Updating an item from the table..." . PHP_EOL;

// Updating an item
$response = $dynamodb->update_item(array(
    'TableName' => $table_name,
    'Key' => array(
        'HashKeyElement' => array( // "id" column
        AmazonDynamoDB::TYPE_NUMBER => '1'
        ),
        'RangeKeyElement' => array( // "date" column
        AmazonDynamoDB::TYPE_NUMBER => $current_time
        )
    ),
```

```
    'AttributeUpdates' => array(
        'val1' => array(
            'Action' => AmazonDynamoDB::ACTION_PUT,
            'Value' => array(AmazonDynamoDB::TYPE_STRING => 'updated-value1')
        ),
        'val22' => array(
            'Action' => AmazonDynamoDB::ACTION_DELETE
        )
    ),
    'Expected' => array(
        'val1' => array(
            'Value' => array( AmazonDynamoDB::TYPE_STRING => 'value1' )
        )
    )
));

// Check for success...
if ($response->isOK())
{
    echo 'Updated the item...' . PHP_EOL;
}
else
{
    print_r($response);
}

#####################################################################
# Deleting an item
```

```
echo PHP_EOL . PHP_EOL;
echo "# Deleting an item from the table..." . PHP_EOL;


// Deleting an item
$response = $dynamodb->delete_item(array(
    'TableName' => $table_name,
    'Key' => array(
        'HashKeyElement' => array( // "id" column
            AmazonDynamoDB::TYPE_NUMBER => '1'
        ),
        'RangeKeyElement' => array( // "date" column
            AmazonDynamoDB::TYPE_NUMBER => $current_time
        )
    )
));


// Check for success...
if ($response->isOK())
{
    echo 'Deleting the item...' . PHP_EOL;
}
else
{
    print_r($response);
}



####################################################################
# Deleting the table - COMMENTED OUT
```

```
/* The following demonstrates how to delete the table, but is commented out so
 you can see the data

* until you're ready to delete it.

echo PHP_EOL . PHP_EOL;

echo "# Deleting the \"${table_name}\" table..." . PHP_EOL;


$response = $dynamodb->delete_table(array(

    'TableName' => $table_name

));


// Check for success...

if ($response->isOK())

{

    echo 'The table is in the process of deleting...' . PHP_EOL;

}

else

{

    print_r($response);

}


####################################################################

# Sleep and poll until the table has been deleted.


$count = 0;

do {

    sleep(1);

    $count++;


    $response = $dynamodb->describe_table(array(
```

```
    'TableName' => $table_name

    ));

}

while ((integer) $response->status !== 400);



echo "The table \"${table_name}\" has been deleted (slept ${count} seconds)."
. PHP_EOL;

*/

?>
```

Example: Batch Operations Using AWS SDK for PHP

Example: Batch Write Operation Using the AWS SDK for PHP for Amazon DynamoDB

The following PHP code example uses batch write API to peform the following tasks:

- Put an iten 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. 164).

This code example assumes that you have followed the Getting Started (Getting Started with Amazon DynamoDB (p. 11)) 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. 485) topic.

Note

For step-by-step instructions to test the following code example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
<?php
require_once dirname(__FILE__) . '/SDKBeta/sdk.class.php';

  $table1_name = 'Forum';
  $table2_name = 'Thread';

  $dynamodb = new AmazonDynamoDB();

  $response = $dynamodb->batch_write_item(array(
      'RequestItems' => array(
          $table1_name => array(
```

```
              array(
                  'PutRequest' => array(
                      'Item' => $dynamodb->attributes(array(
                          'Name'   => 'S3 Forum',
                          'Threads' => 0
                      ))
                  )
              )
          ),
           $table2_name => array(
              array(
                  'PutRequest' => array(
                      'Item' => $dynamodb->attributes(array(
                          'ForumName'   => 'S3 Forum',
                          'Subject' => 'My sample question',
                          'Message'=> 'Message Text.',
                          'KeywordTags'=> array('S3', 'Bucket')
                      ))
                  )
              ),
              array(
                  'DeleteRequest' => array(
                      'Key' => $dynamodb->attributes(array(
                          'HashKeyElement' =>'Some hash value',
                          'RangeKeyElement' => 'Some range key'
                      ))
                  )
              )
           )
       )
  ));

  print_r($response);
?>
```

Query and Scan in Amazon DynamoDB

Topics

Overview

Once data is in an Amazon DynamoDB table, you have two APIs for searching the data: Query and Scan.

- **Query**
  A query operation searches only primary key attribute values and supports a subset of comparison operators on key attribute values to refine the search process. A query returns all of the item data for the matching primary keys (all of each item's attributes) up to 1MB of data per query operation. A query operation always returns results, but can return empty results.

  Query results are always sorted by the range key, based on ASCII character code values. By default, the sort order is ascending. To reverse the order use the `ScanIndexForward` parameter set to `false`.

  Query supports a specific set of comparison operators. For information about each comparison operator available for query operations, see the API entry for Query (p. 417).

- **Scan**
  A scan operation scans the entire table. You can specify filters to apply to the results to refine the values returned to you, after the complete scan. Amazon DynamoDB puts a 1MB limit on the scan (the limit applies before the results are filtered). A scan can result in no table data meeting the filter criteria.

  Scan supports a specific set of comparison operators. For information about each comparison operator available for scan operations, see the API entry for Scan (p. 425).

Scan and Query Performance

Generally, a query operation is more efficient than a scan operation.

A scan operation always scans the entire table, 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 with a filter that removes many results, if possible. Also, as a table 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 in a single operation. For quicker response times, design your tables in a way that can use the Query, Get, or BatchGetItem APIs, instead. Or, design your application to use scan operations in a way that minimizes the impact on your table's request rate. For more information, see Provisioned Throughput Guidelines in Amazon DynamoDB (p. 68).

A query operation only searches for a specific range of keys that satisfy a given set of key conditions and does not have the added step of filtering out results. 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 1MB limit.

Query operation performance depends on the amount of data retrieved, rather than the overall number of primary keys in a table. The parameters for a query operation (and consequently the number of matching keys) determine the performance of the query. For example, a query operation on one table that contains a large set of range key elements for a single hash key element can be more efficient than a query operation on a 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.

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.

Pagination, LastEvaluatedKey, and ExclusiveStartKey

Amazon DynamoDB uses pagination (divides the scan or query process into distinct pieces) to respond to queries quickly. A single scan or query operation is limited to 1MB. If you scan a table that has more than 1MB of data, you'll need to perform another scan operation to continue to the next 1MB of data in the table. If you query for specific attributes that match values that amount to more than 1MB of data, you'll need to perform another query request for the next 1MB of data. The second query request uses a starting point (*ExclusiveStartKey*) based on the key of the last returned value (*LastEvaluatedKey*) so you can progressively query or scan for new data in 1MB increments. The *LastEvaluatedKey* is *null* when the entire query or scan result set is complete (i.e. the operation processed the “last page”).

Count and ScannedCount

The Amazon DynamoDB Scan and Query APIs use *Count* values for two distinct purposes.

In a request, set the *Count* parameter to `true` if you want Amazon DynamoDB to provide the total number of items that match the scan filter or query condition, instead of a list of the matching items.

In a response, Amazon DynamoDB returns a *Count* value for the number of matching items in a request. If the matching items for a scan filter or query condition is over 1MB, *Count* contains a partial count of the total number of items that match the request. To get the full count of items that match a request, use the *LastEvaluatedKey* in a subsequent request. Repeat the request until Amazon DynamoDB no longer returns a *LastEvaluatedKey*.

For a scan operation, Amazon DynamoDB also returns a *ScannedCount* value. The *ScannedCount* value is the total number of items scanned before any filter is applied to the results.

Limit

The Amazon DynamoDB Scan and Query 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 Amazon DynamoDB to process before returning results.

In a response, Amazon DynamoDB returns all the matching results within the scope of the *Limit* value. For example, if you provide a *Limit* value of 6 for a scan request, the scan operation returns the items within the first six items in the table that match the scan filter requirements (if provided). If no filter is provided, the scan operation returns the first six items. If you provide a *Limit* value of 6 for a query request, the query operation processes six items in the table that match the query parameters.

For either a scan or query operation, Amazon 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 a request in a table, use the *LastEvaluatedKey* in a subsequent request. Repeat the request until Amazon DynamoDB no longer returns a *LastEvaluatedKey*.

Consistency for Scan and Query

A scan result is not a consistent read, meaning that changes to data immediately before the scan takes place might not be included in the scan results. A query result is an eventually consistent read, with an option to set the query request for a consistent read. An eventually consistent read might not reflect the results of a recently completed put or update operation. For more information, see Data Read and Consistency Considerations (p. 7).

Capacity Units Consumed by Query and Scan Operation

When you create a table you specify your read and write capacity unit requirements. When you send a query or a scan request, you consume the capacity units set for the table. For more information about how Amazon DynamoDB computes the capacity units consumed by your operation, see Capacity Units Calculations for Various Operations (p. 67).

Querying Tables in Amazon DynamoDB

Topics

This section shows basic queries and their results.

Querying Tables Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

The `query` method enables you to query a table. You can query a table only if it has a composite primary key, that is, a primary that is composed of both a hash and range attribute.

The following are the steps to retrieve an item using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class and provide your credentials.
2. Create an instance of the `QueryRequest` class and provide query operation parameters.

   You can provide both the required and optional parameters using this object.
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 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 in Amazon DynamoDB (p. 445).

```
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.

```
QueryRequest queryRequest = new QueryRequest()
    .withTableName("Reply")
    .withHashKeyValue(new AttributeValue().withS("Amazon DynamoDB#DynamoDB 
Thread 1"));

QueryResult result = client.query(queryRequest);
for (Map<String, AttributeValue> item : result.getItems()){
    printItem(item);
}
```

Specifying Optional Parameters

The `query` method supports several optional parameters. For example, you can optionally filter the query result in the preceding query to return replies in the past two weeks by specifying a condition. The condition is called 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 (p. 417).

The following Java code snippet retrieves forum thread replies posted in the past 15 days. The snippet specifies optional parameters using:

- A `Condition` instance 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 `withAttributesToGet` method to specify a list of attributes to retrieve for items in the query results.
- The `withConsistentRead` method as true to request the latest item data. To learn more about consistent read, see Amazon DynamoDB Data Model (p. 3).

```
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);

Condition rangeKeyCondition = new Condition()
    .withComparisonOperator(ComparisonOperator.GT.toString())
    .withAttributeValueList(new AttributeValue().withS(twoWeeksAgoStr));

QueryRequest queryRequest = new QueryRequest()
    .withTableName("Reply")
    .withHashKeyValue(new AttributeValue().withS("Amazon DynamoDB#DynamoDB
Thread 1"))
    .withAttributesToGet(Arrays.asList("Subject", "ReplyDateTime"))
    .withRangeKeyCondition(rangeKeyCondition)
    .withConsistentRead(true);

QueryResult result = client.query(queryRequest);
for (Map<String, AttributeValue> item : result.getItems()){
    printItem(item);
}
```

You can also optionally limit the page size, that is, the number of items per page, by using the `withLimit` method of the request. 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 using the `withExclusiveStartKey` method. Initially, the parameter for 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 queries the Reply table. In the request, the `withLimit` and `withExclusiveStartKey` methods are used. The `do/while` loop continues to scan one page at time until the `getLastEvaluatedKey` method of the result returns a null value.

```
Key lastKeyEvaluated = null;
do {
    QueryRequest queryRequest = new QueryRequest()
        .withTableName("Reply")
        .withHashKeyValue(new AttributeValue().withS("DynamoDB Thread 1 -
Reply1"))
        .withLimit(10)
        .withExclusiveStartKey(lastKeyEvaluated);

    QueryResult result = client.query(queryRequest);
    for (Map<String, AttributeValue> item : result.getItems()){
        printItem(item);
    }
```

```
    lastKeyEvaluated = result.getLastEvaluatedKey();
} while (lastKeyEvaluated != null);
```

Additional AWS SDK for Java APIs

The examples in this section use the AWS SDK for Java low-level API. Additionally, the SDK provides Object Persistence Model API that enable you to map your client-side classes to your Amazon DynamoDB tables and perform query operations on them. For more information, see Using the AWS SDK for Java with Amazon DynamoDB (p. 259).

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 in Amazon DynamoDB (p. 445).

```
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 'Amazon DynamoDB'.

- Find replies for a thread.
- Find replies for a thread. Specify the Limit query parameter to set page size.

  This function illustrates the use of pagination to process multipage results. 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 the preceding two queries shows how you can specify range key conditions to filter query results and use other optional query parameters.

Note

This code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.io.IOException;

import java.text.SimpleDateFormat;

import java.util.Arrays;

import java.util.Date;

import java.util.Map;
```

```
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodb.model.AttributeValue;
import com.amazonaws.services.dynamodb.model.ComparisonOperator;
import com.amazonaws.services.dynamodb.model.Condition;
import com.amazonaws.services.dynamodb.model.Key;
import com.amazonaws.services.dynamodb.model.QueryRequest;
import com.amazonaws.services.dynamodb.model.QueryResult;

public class LowLevelQuery {

    static AmazonDynamoDBClient client;
    static String tableName = "Reply";

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

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

        createClient();

        findRepliesForAThread(forumName, threadSubject);
        findRepliesForAThreadSpecifyOptionalLimit(forumName, threadSubject);
        findRepliesInLast15DaysWithConfig(forumName, threadSubject);
        findRepliesPostedWithinTimePeriod(forumName, threadSubject);

    }
```

```
    private static void createClient() throws IOException {

        AWSCredentials credentials = new PropertiesCredentials(
                LowLevelQuery.class.getResourceAsStream("AwsCredentials.proper
ties"));

        client = new AmazonDynamoDBClient(credentials);

    }

    private static void findRepliesForAThread(String forumName, String thread
Subject) {

        String replyId = forumName + "#" + threadSubject;
        QueryRequest queryRequest = new QueryRequest().withTableName(tableName)

                .withHashKeyValue(new AttributeValue().withS(replyId));

        QueryResult result = client.query(queryRequest);
        for (Map<String, AttributeValue> item : result.getItems()) {
            printItem(item);
        }
    }

    private static void findRepliesForAThreadSpecifyOptionalLimit(String forum
Name, String threadSubject) {

        String replyId = forumName + "#" + threadSubject;
        Key lastKeyEvaluated = null;
        do {
```

```
            QueryRequest queryRequest = new QueryRequest()
                    .withTableName(tableName)
                    .withHashKeyValue(
                            new AttributeValue().withS(replyId))
                    .withLimit(1).withExclusiveStartKey(lastKeyEvaluated);

            QueryResult result = client.query(queryRequest);
            for (Map<String, AttributeValue> item : result.getItems()) {
                printItem(item);
            }
            lastKeyEvaluated = result.getLastEvaluatedKey();
        } while (lastKeyEvaluated != null);
    }

    private static void findRepliesInLast15DaysWithConfig(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);

        Condition rangeKeyCondition = new Condition()
        .withComparisonOperator(ComparisonOperator.GT.toString())
        .withAttributeValueList(new AttributeValue().withS(twoWeeksAgoStr));
```

```
        QueryRequest queryRequest = new QueryRequest().withTableName(tableName)

            .withHashKeyValue(new AttributeValue().withS(replyId))

            .withRangeKeyCondition(rangeKeyCondition)

            .withAttributesToGet(Arrays.asList("Message", "ReplyDateTime",
"PostedBy"));


        QueryResult result = client.query(queryRequest);

        for (Map<String, AttributeValue> item : result.getItems()) {

            printItem(item);

        }


    }


    private static void findRepliesPostedWithinTimePeriod(String forumName,
String threadSubject) {


        String replyId = forumName + "#" + threadSubject;

        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);


        Condition rangeKeyCondition = new Condition()

        .withComparisonOperator(ComparisonOperator.BETWEEN.toString())

        .withAttributeValueList(new AttributeValue().withS(startDate),

                                new AttributeValue().withS(endDate));
```

```
        QueryRequest queryRequest = new QueryRequest().withTableName(tableName)

            .withHashKeyValue(new AttributeValue().withS(replyId))
            .withRangeKeyCondition(rangeKeyCondition)
            .withAttributesToGet(Arrays.asList("Message", "ReplyDateTime",
"PostedBy"));


        QueryResult result = client.query(queryRequest);
        for (Map<String, AttributeValue> item : result.getItems()) {
            printItem(item);
        }
    }


    private static void printItem(Map<String, AttributeValue> attributeList) {

        for (Map.Entry<String, AttributeValue> item : attributeList.entrySet())
 {
            String attributeName = item.getKey();
            AttributeValue value = item.getValue();
            System.out.println(attributeName
                    + " "
                    + (value.getS() == null ? "" : "S=[" + value.getS() + "]")

                    + (value.getN() == null ? "" : "N=[" + value.getN() + "]")

                    + (value.getSS() == null ? "" : "SS=[" + value.getSS() +
"]")
                    + (value.getNS() == null ? "" : "NS=[" + value.getNS() +
"] \n"));
        }
    }
```

```
}
```

Querying Tables Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB

The `Query` method enables you to query a table. You can query only if the table has a composite primary key, that is the primary is composed of both the hash and range attributes.

The following are the steps to query a table using low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class and provide your credentials.
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 in Amazon DynamoDB (p. 445).

```
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.

```
var request = new QueryRequest
{
  TableName = "Reply",
  HashKeyValue = new AttributeValue { S = "Amazon 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 filter the query result in the preceding query to return replies in the past two weeks by specifying a condition. The condition is called 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 (p. 417).

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.
- An `AttributesToGet` parameter to specify a list of attributes to retrieve for items in the query result.
- A `ConsistentRead` parameter to request the latest item data. To learn more about consistent read, see Amazon DynamoDB Data Model (p. 3).

```
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
string twoWeeksAgoString = twoWeeksAgoDate.ToString(AWSSDKUtils.ISO8601Date
Format);

var request = new QueryRequest
{
  TableName = "Reply",
  HashKeyValue = new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 2"
},
  // Optional parameters.
  RangeKeyCondition = new Condition
  {
    ComparisonOperator = "GT",
    AttributeValueList = new List<AttributeValue>()
    {
       new AttributeValue { S = twoWeeksAgoString }
    }
  },
  AttributesToGet = new List<string> { "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, that is, 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.

```
Key lastKeyEvaluated = null;
do
{
  var request = new QueryRequest
  {
    TableName = "Reply",
    HashKeyValue = new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 
1" },
    // 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);
```

Additional AWS SDK for .NET APIs

The examples in this section use AWS SDK for .NET low-level API. The SDK also supports helper API that further simplify your application development. Additionally, the SDK provides Object Persistence Model API that enable you to map your client-side classes to your Amazon DynamoDB tables and perform query operations on them. For more information, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

Example - Querying Amazon DynamoDB 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 in Amazon DynamoDB (p. 445).

```
Forum ( Name, ... )
Thread ( ForumName, Subject, Message, LastPostedBy, LastPostDateTime, ...)
Reply ( Id, ReplyDateTime, Message, PostedBy, ...)
```

In this C# code example, you execute variations of "Find replies for a thread "DynamoDB Thread 1" in forum "Amazon 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 filter query results and use other optional query parameters.

For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.Model;

using Amazon.Runtime;

using Amazon.Util;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      try

      {

        client = new AmazonDynamoDBClient();


        // Query a specific forum and thread.

        string forumName = "Amazon DynamoDB";

        string threadSubject = "DynamoDB Thread 1";


        FindRepliesForAThread(forumName, threadSubject);

        FindRepliesForAThreadSpecifyOptionalLimit(forumName, threadSubject);
```

```
        FindRepliesInLast15DaysWithConfig(forumName, threadSubject);

        FindRepliesPostedWithinTimePeriod(forumName, threadSubject);


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

        Console.ReadLine();

      }

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

      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",

        HashKeyValue = new AttributeValue { S = replyId },

        RangeKeyCondition = new Condition
```

```
        {
          ComparisonOperator = "BETWEEN",
          AttributeValueList = new List<AttributeValue>()
          {
             new AttributeValue { S = start },
             new AttributeValue { S = end }
          }
        }
      };

      var response = client.Query(request);
      var result = response.QueryResult;

     Console.WriteLine("\nNo. of reads used (by query in FindRepliesPostedWith
inTimePeriod) {0}",
                        response.QueryResult.ConsumedCapacityUnits);
      foreach (Dictionary<string, AttributeValue> item
        in response.QueryResult.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",

        HashKeyValue = new AttributeValue { S = replyId },

        RangeKeyCondition = new Condition

        {

          ComparisonOperator = "GT",

          AttributeValueList = new List<AttributeValue>()

          {

             new AttributeValue { S = twoWeeksAgoString }

          }

        },

        // Optional parameter.

        AttributesToGet = new List<string> { "Id", "ReplyDateTime", "PostedBy"
 },

        // Optional parameter.

        ConsistentRead = true

      };


      var response = client.Query(request);

      var result = response.QueryResult;


      Console.WriteLine("No. of reads used (by query in FindRepliesIn
Last15DaysWithConfig) {0}",
```

```
                   response.QueryResult.ConsumedCapacityUnits);

      foreach (Dictionary<string, AttributeValue> item

        in response.QueryResult.Items)

      {

        PrintItem(item);

      }

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

      Console.ReadLine();

    }



    private static void FindRepliesForAThreadSpecifyOptionalLimit(string forum
Name, string threadSubject)

    {

      Console.WriteLine("*** Executing FindRepliesForAThreadSpecifyOptionalLim
it() ***");

      string replyId = forumName + "#" + threadSubject;



      Key lastKeyEvaluated = null;

      do

      {

        var request = new QueryRequest

        {

          TableName = "Reply",

          HashKeyValue = 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 FindRepliesForAThread
SpecifyLimit) {0}\n",

                          response.QueryResult.ConsumedCapacityUnits);

        foreach (Dictionary<string, AttributeValue> item

          in response.QueryResult.Items)

        {

          PrintItem(item);

        }

        lastKeyEvaluated = response.QueryResult.LastEvaluatedKey;


      } while (lastKeyEvaluated != null);

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




      Console.ReadLine();

    }


    private static void FindRepliesForAThread(string forumName, string thread
Subject)

    {

      Console.WriteLine("*** Executing FindRepliesForAThread() ***");

      string replyId = forumName + "#" + threadSubject;


      var request = new QueryRequest

      {


        TableName = "Reply",

        HashKeyValue = new AttributeValue { S = replyId }

      };
```

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

      var result = response.QueryResult;

      Console.WriteLine("No. of reads used (by query in FindRepliesForAThread)
 {0}\n",
                        response.QueryResult.ConsumedCapacityUnits);

      foreach (Dictionary<string, AttributeValue> item
        in response.QueryResult.Items)
      {
        PrintItem(item);
      }
      Console.WriteLine("To continue, press Enter");
      Console.ReadLine();
    }


    private static void PrintItem(
     Dictionary<string, AttributeValue> attributeList)
    {
      foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
      {
        string attributeName = kvp.Key;
        AttributeValue value = kvp.Value;


        Console.WriteLine(
            attributeName + " " +
            (value.S == null ? "" : "S=[" + value.S + "]") +
            (value.N == null ? "" : "N=[" + value.N + "]") +
            (value.SS == null ? "" : "SS=[" + string.Join(",", value.SS.ToAr
ray()) + "]") +
            (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToAr
ray()) + "]")
```

```
        );
      }
      Console.WriteLine("************************************************");
    }


  }
}
```

Querying Tables Using the AWS SDK for PHP Low-Level API for Amazon DynamoDB

The `query` function enables you to query a table. You can query only if the table has a composite primary key, that is the primary is made of both the hash and range attributes.

The following steps guide you through querying using the AWS SDK for PHP.

1. Create an instance of the `AmazonDynamoDB` class (the client).
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. You can request read results to be strictly consistent.
3. Load the response into a local variable, such as `$query_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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$response = $dynamodb->query(array(
    'TableName' => 'Reply',
        'HashKeyValue' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 1' ),
));

// 200 response indicates Success
print_r($response);
```

Specifying Optional Parameters

The `query` function supports several optional parameters. For example, you can optionally filter 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 Amazon 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 (p. 417).

The following PHP example retrieves forum thread replies posted in the past 7 days. The sample specifies the following optional parameters:

- `RangeKeyCondition` to retrieve only the replies within the last 7 days.

  The condition specifies ReplyDateTime value and a comparison operator to use for comparing dates.
- `AttributesToGet` to specify a list of attributes to retrieve for items in the query results
- `ConsistentRead` parameter to request the latest item data. By default read operations are eventually consistent. You can set `ConsistentRead` to `true` if you want a strictly consistent read result. To learn more about consistency, see Data Read and Consistency Considerations (p. 7).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$seven_days_ago = strtotime("-7 days");

$response = $dynamodb->query(array(
    'TableName' => 'Reply',
   'HashKeyValue' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon DynamoDB#Dy
namoDB Thread 2' ),
// optional parameters
    'AttributesToGet' => array( 'Subject', 'ReplyDateTime', 'PostedBy' ),
    'ConsistentRead' => true,
    'RangeKeyCondition' => array(
        'ComparisonOperator' => AmazonDynamoDB::CONDITION_LESS_THAN_OR_EQUAL,
        'AttributeValueList' => array(
            array( AmazonDynamoDB::TYPE_NUMBER => $seven_days_ago )
        )
    )
));

// 200 response indicates Success
print_r($response);
```

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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

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

$query_response = $dynamodb->query(array(
    'TableName' => 'Reply',
    'Limit' => 2,
    'HashKeyValue' => array(
        AmazonDynamoDB::TYPE_STRING => 'Amazon DynamoDB#DynamoDB Thread 2',
    ),
    'RangeKeyCondition' => array(
       'ComparisonOperator' => AmazonDynamoDB::CONDITION_GREATER_THAN_OR_EQUAL,

        'AttributeValueList' => array(
            array( AmazonDynamoDB::TYPE_STRING => $fourteen_days_ago )
        )
    )
));

 print_r($query_response);

// Do we have more data? Fetch it!
if (isset($query_response->body->LastEvaluatedKey))
{
$query2_response = $dynamodb->query(array(
    'TableName' => 'Reply',
    'Limit' => 2,
   'ExclusiveStartKey' => $query_response->body->LastEvaluatedKey->to_array()-
>getArrayCopy(),
   'HashKeyValue' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon DynamoDB#Dy
namoDB Thread 2' ),
    'RangeKeyCondition' => array(
       'ComparisonOperator' => AmazonDynamoDB::CONDITION_GREATER_THAN_OR_EQUAL,

        'AttributeValueList' => array(
            array( AmazonDynamoDB::TYPE_STRING => $fourteen_days_ago )
        )
    )
));
// 200 response indicates success
 print_r($query2_response);
}
```

Scanning Tables in Amazon DynamoDB

Topics

This section shows basic scans and their results.

Scanning Tables Using the AWS SDK for Java Low-Level API for Amazon DynamoDB

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 in Amazon DynamoDB (p. 182).

The following are the steps to scan a table using the low-level Java SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class and provide your credentials.
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 `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 Java code snippet scans the entire table. The `ScanRequest` instance specifies the name of the table to scan.

```
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 one or more scan filters 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 Query (p. 425).

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 scan filter condition that specifies to retrieve only the items priced less than 0 (error condition).
- A list of attributes to retrieve for items in the query results.

The following Java code snippet scans the ProductCatalog table to find all items priced less than 0.

```
Condition scanFilterCondition = new Condition()
    .withComparisonOperator(ComparisonOperator.LT.toString())
    .withAttributeValueList(new AttributeValue().withN("0"));
Map<String, Condition> conditions = new HashMap<String, Condition>();
conditions.put("Price", scanFilterCondition);

ScanRequest scanRequest = new ScanRequest()
    .withTableName("ProductCatalog")
    .withScanFilter(conditions)
    .withAttributesToGet(Arrays.asList("Id"));

ScanResult result = client.scan(scanRequest);
for (Map<String, AttributeValue> item : result.getItems()) {
    printItem(item);
}
```

You can also optionally limit the page size, that is, the number of items per page, by using the `withLimit` method of the scan request. Each time you execute the `scan` method, you get one page of results that has the specified number of items. To fetch the next page, you execute the `scan` method again by providing the primary key value of the last item in the previous page so that the `scan` method can return the next set of items. You provide this information in the request by using the `withExclusiveStartKey` method. Initially, the parameter of this method can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following Java code snippet scans the ProductCatalog table. In the request, the `withLimit` and `withExclusiveStartKey` methods are used. The `do/while` loop continues to scan one page at time until the `getLastEvaluatedKey` method of the result returns a value of null.

```
Key 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);
```

Additional AWS SDK for Java APIs

The examples in this section use the AWS SDK for Java low-level API. Additionally, the SDK provides Object Persistence Model API that enable you to map your client-side classes to your Amazon DynamoDB tables and perform scan operations on them. For more information, see Using the AWS SDK for Java with Amazon DynamoDB (p. 259).

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 0.

Note

This code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

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

import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodb.model.AttributeValue;
import com.amazonaws.services.dynamodb.model.ComparisonOperator;
import com.amazonaws.services.dynamodb.model.Condition;
import com.amazonaws.services.dynamodb.model.ScanRequest;
import com.amazonaws.services.dynamodb.model.ScanResult;

public class LowLevelScan {

    static AmazonDynamoDBClient client;
    static String tableName = "ProductCatalog";

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

```
        createClient();


        findProductsForPriceLessThanZero();

    }


    private static void createClient() throws IOException {


        AWSCredentials credentials = new PropertiesCredentials(

                LowLevelScan.class.getResourceAsStream("AwsCredentials.proper
ties"));


        client = new AmazonDynamoDBClient(credentials);


    }


    private static void findProductsForPriceLessThanZero() {


        Condition scanFilterCondition = new Condition()

            .withComparisonOperator(ComparisonOperator.LT.toString())

            .withAttributeValueList(new AttributeValue().withN("0"));

        Map<String, Condition> conditions = new HashMap<String, Condition>();

        conditions.put("Price", scanFilterCondition);


        ScanRequest scanRequest = new ScanRequest()

            .withTableName(tableName)

            .withScanFilter(conditions)

            .withAttributesToGet(Arrays.asList("Id", "Title", "ProductCat
egory"));


        ScanResult result = client.scan(scanRequest);
```

```
        System.out.println("Scan of " + tableName + " for items with a price 
less than zero.");

        for (Map<String, AttributeValue> item : result.getItems()) {

            printItem(item);

        }

    }



    private static void printItem(Map<String, AttributeValue> attributeList) {

        for (Map.Entry<String, AttributeValue> item : attributeList.entrySet())
 {

            String attributeName = item.getKey();

            AttributeValue value = item.getValue();

            System.out.println(attributeName

                    + " "

                    + (value.getS() == null ? "" : "S=[" + value.getS() + "]")

                    + (value.getN() == null ? "" : "N=[" + value.getN() + "]")

                    + (value.getSS() == null ? "" : "SS=[" + value.getSS() + 
"]")

                    + (value.getNS() == null ? "" : "NS=[" + value.getNS() + 
"] \n"));

        }

    }

}
```

Scanning Tables Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB

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 in Amazon DynamoDB (p. 182).

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 and provide your credentials.
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.

```
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 Query (p. 425).

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 `ScanFilter` parameter to retrieve only the items priced less than 0 (error condition).
- An `AttributesToGet` parameter to specify a list of 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.
   ScanFilter = new Dictionary<string, Condition>()
           {
               { "Price", new Condition
```

```
                    {
                       ComparisonOperator = "LT",
                        AttributeValueList = new List<AttributeValue>()
                        {
                            new AttributeValue { N = "0" }
                        }
                    }
                }
            },
    AttributesToGet = new List<string> { "Id" }
  };
```

You can also optionally limit the page size, that is, the number of items per page, by adding the optional `Limit` parameter. Each time you execute the `Scan` method, you get one page of results that has the specified number of items. To fetch the next page, you execute the `Scan` method again by providing the primary key value of the last item in the previous page so that the `Scan` method can return the next set of items. You provide this information in the request by setting the `ExclusiveStartKey` property. Initially, this property can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following C# code snippet scans the ProductCatalog table. In the request, it specifies the `Limit` and `ExclusiveStartKey` optional parameters. The `do/while` loop continues to scan one page at time until the `LastEvaluatedKey` returns a null value.

```
Key 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.ScanResult.Items)
  {
    PrintItem(item);
  }
  lastKeyEvaluated = response.ScanResult.LastEvaluatedKey;

} while (lastKeyEvaluated != null);
```

Additional AWS SDK for .NET APIs

The examples in this section use AWS SDK for .NET low-level API. The SDK also supports helper API that further simplify your application development. Additionally, the SDK provides Object Persistence Model API that enable you to map your client-side classes to your Amazon DynamoDB tables and perform scan operations on them. For more information, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

Example - Scan Using .NET

The following C# code example provides a working sample that scans the ProductCatalog table to find items priced less than 0.

For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.Model;

using Amazon.Runtime;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      try

      {

        client = new AmazonDynamoDBClient();

        FindProductsForPriceLessThanZero();


        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 FindProductsForPriceLessThanZero()
{
  Key lastKeyEvaluated = null;
  do
  {
    var request = new ScanRequest
    {
      TableName = "ProductCatalog",
      Limit = 2,
      ExclusiveStartKey = lastKeyEvaluated,
      ScanFilter = new Dictionary<string, Condition>()
            {
                { "Price", new Condition
                    {
                       ComparisonOperator = "LT",
                        AttributeValueList = new List<AttributeValue>()
                        {
                            new AttributeValue {  N = "0" }
                        }
                    }
                }
            },
      AttributesToGet = new List<string> { "Id", "Title", "Price" }
    };

    var response = client.Scan(request);

    foreach (Dictionary<string, AttributeValue> item
```

```
          in response.ScanResult.Items)
        {
          Console.WriteLine("\nScanThreadTableUsePaging - printing.....");
          PrintItem(item);
        }
        lastKeyEvaluated = response.ScanResult.LastEvaluatedKey;

      } while (lastKeyEvaluated != null);

      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.ToAr
ray()) + "]") +
            (value.NS == null ? "" : "NS=[" + string.Join(",", value.NS.ToAr
ray()) + "]")
        );
```

```
      }

      Console.WriteLine("************************************************");

    }

  }

}
```

Scanning Tables Using the AWS SDK for PHP Low-Level API for Amazon DynamoDB

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 in Amazon DynamoDB (p. 182).

The following tasks guide you through scanning a table using the AWS SDK for NET low-level API:

1. Create an instance of the `AmazonDynamoDB` class (the client).
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 by default. You can request read results to be strictly consistent.
3. Load the response into a local variable, such as `$scan_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 entire table.

Note

This code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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. 485) topic.

For step-by-step instructions to run the following example, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$scan_response = $dynamodb->scan(array(
'TableName' => 'Reply'
```

```
));

// 200 response indicates Success
print_r($scan_response);
```

The scan operation response is a SimpleXMLElement object. You can perform operations on the object contents once you typecast the values in the object as a string. For example, the following code snippet scans the entire ProductCatalog table, and returns the product Id and Title values.

```
$dynamodb = new AmazonDynamoDB();

$scan_response = $dynamodb->scan(array(
 'TableName' => 'ProductCatalog'
));

foreach ($scan_response->body->Items as $item)
{
 echo "<p><strong>Item Number:</strong>". (string) $item->Id->{AmazonDy
namoDB::TYPE_NUMBER};
 echo "<br><strong>Item Name: </strong>". (string) $item->Title->{AmazonDy
namoDB::TYPE_STRING} ."</p>";
}
```

Specifying Optional Parameters

The `scan` function supports several optional parameters. For example, you can optionally use a scan filter to filter the scan result. In a scan filter 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 Query (p. 425).

The following PHP code scans the ProductCatalog table to find items that are priced less than 0. The sample specifies the following optional parameters:

- `ScanFilter` to retrieve only the items priced less than 0 (error condition).
- `AttributesToGet` to specify a list of 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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$scan_response = $dynamodb->scan(array(
 'TableName' => 'ProductCatalog',
 'AttributesToGet' => array('Id'),
 'ScanFilter' => array(
  'Price' => array(
   'ComparisonOperator' => AmazonDynamoDB::CONDITION_LESS_THAN,
   'AttributeValueList' => array(
    array( AmazonDynamoDB::TYPE_NUMBER => '0' )
   )
  ),
 )
));
```

```
// 200 response indicates Success
print_r($scan_response);
```

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.

```
// Instantiate the class
$dynamodb = new AmazonDynamoDB();

$scan_response = $dynamodb->scan(array(
    'TableName' => 'ProductCatalog',
    'Limit'     => 2
));

// 200 response indicates Success
print_r($scan_response);

// Do we have more data? Fetch it!
if (isset($scan_response->body->LastEvaluatedKey))
{
$scan2_response = $dynamodb->scan(array(
    'TableName' => 'ProductCatalog',
    'Limit'     => 2,
    'ExclusiveStartKey' => $scan_response->body->LastEvaluatedKey->to_array()-
>getArrayCopy()
));

// 200 response indicates Success
print_r($scan2_response);
}
```

The Amazon DynamoDB Console

Topics

Amazon DynamoDB is available in the AWS Management Console. With the Amazon DynamoDB console you can create, update and monitor tables. To add or query data in your tables, use one of the AWS SDKs. For more information, see Getting Started with Amazon DynamoDB (p. 11).

Amazon DynamoDB Console Overview

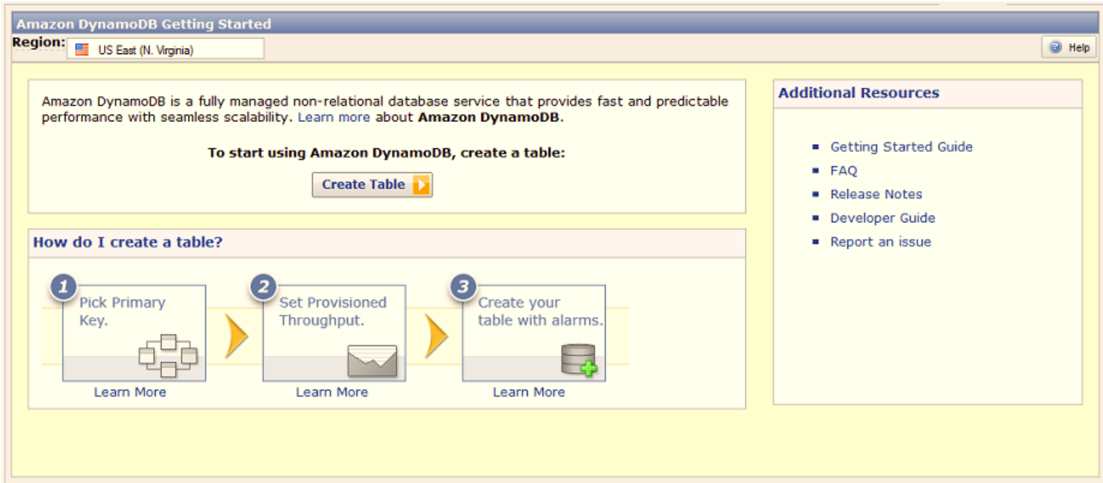
The Amazon DynamoDB console includes:

- A Create Table dialog that walks you through the steps to create a table and set up an alarm for monitoring your table's capacity.
- A provisioned throughput calculator.
- A quick view of your table's top monitoring metrics from Amazon CloudWatch.
- The ability to create custom alarms, and a quick view of all your current alarms for each table.

Your First Visit to the Amazon DynamoDB Console

Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodb/.

If your account doesn't already have tables in Amazon 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, links to relevant documentation and resources.

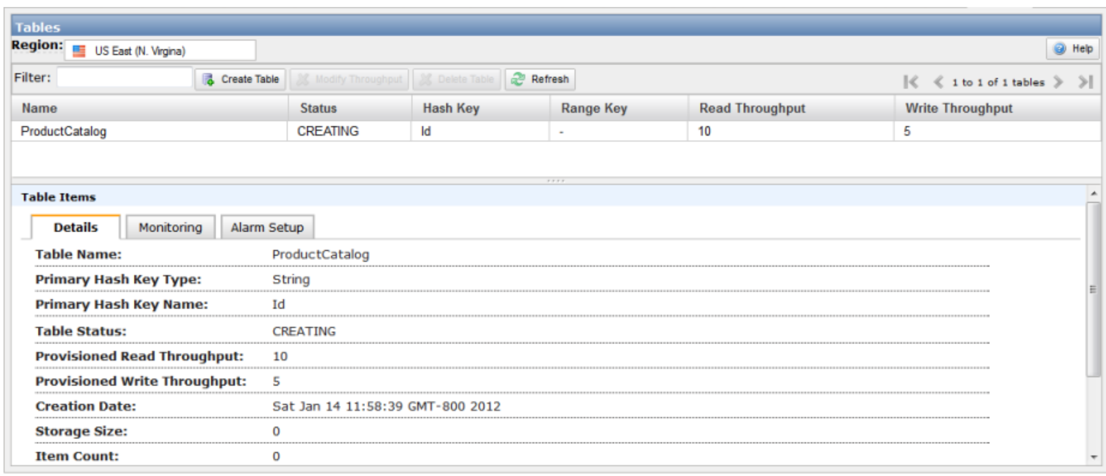


Detailed steps for creating your first table in the console are in Getting Started with Amazon DynamoDB (p. 11).

The Amazon DynamoDB Content in the Console

Once you have one or more tables, the console displays the tables as a list.

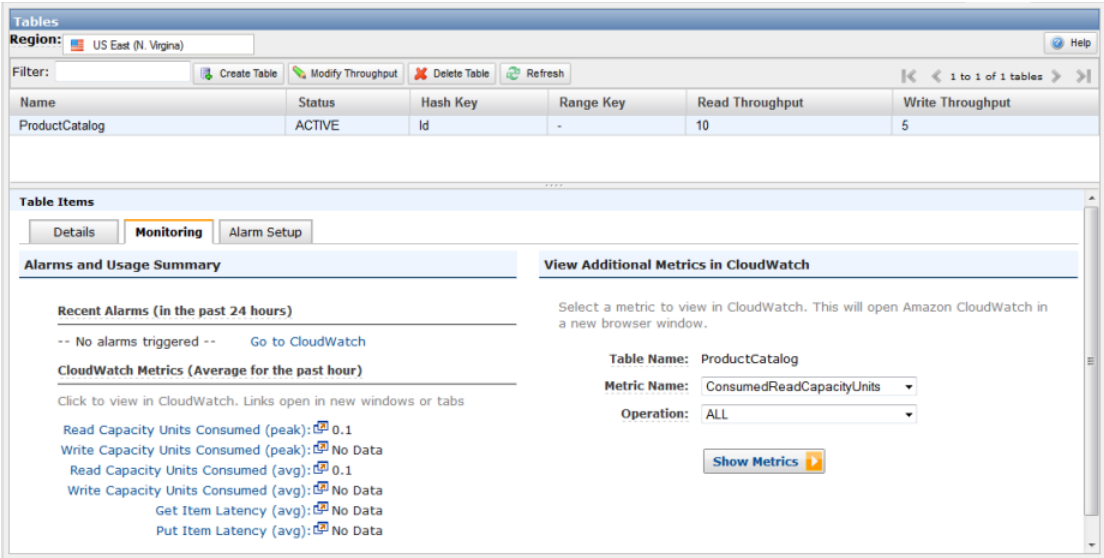
Select a table in your list to see information in the lower pane about your table. In the lower pane, you also have the option to set up alarms and view Amazon CloudWatch metrics for the table.



Monitoring Amazon DynamoDB Tables in the Console

The console for Amazon DynamoDB displays some metrics for your table in the lower pane. If you need to see other metrics, you can use the console for Amazon DynamoDB to set parameters for Amazon Cloudwatch to display information about your table.

To see the Amazon Cloudwatch metrics for your table, with your table selected, click the **Monitoring** tab.



Note

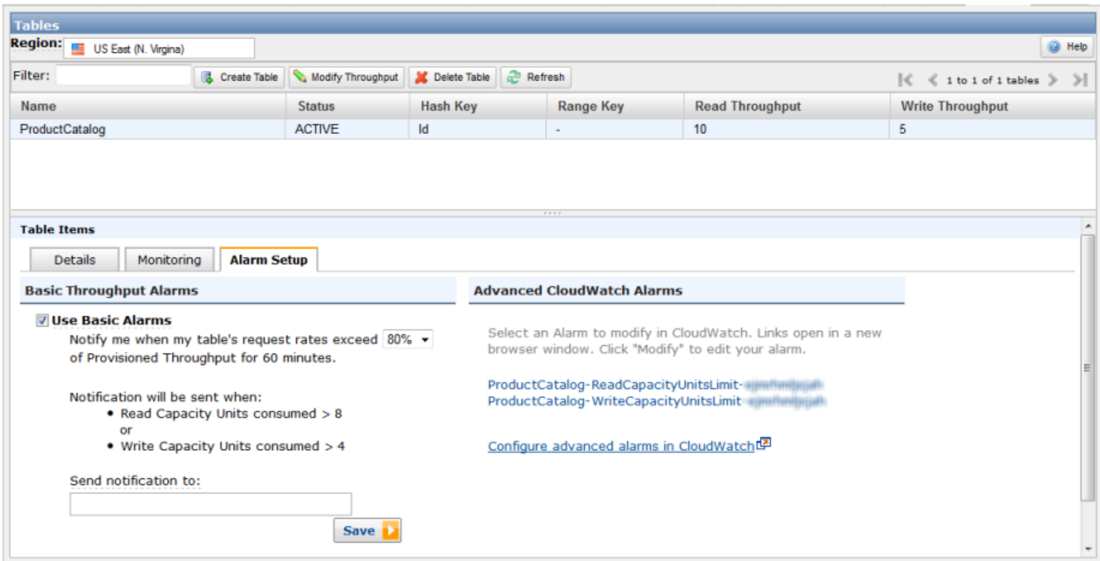
Amazon CloudWatch metrics require time to appear in the console. Wait approximately five minutes and click **Refresh** to see your table details. Also, Amazon DynamoDB updates the Storage Size value approximately every six hours. Recent changes might not be reflected in this value.

For more information about Amazon CloudWatch metrics for Amazon DynamoDB, see Monitoring Amazon DynamoDB Tables with Amazon CloudWatch (p. 223).

Setting up Amazon CloudWatch Alarms in the Console

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 Amazon 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 Amazon CloudWatch console. For more information about setting up alarms, see the Amazon CloudWatch Help in the Amazon CloudWatch console, or see the Amazon Cloudwatch Documentation.

Monitoring Amazon DynamoDB Tables with Amazon CloudWatch

The following scenarios cover operations for monitoring Amazon DynamoDB tables.

Topics

Amazon DynamoDB and Amazon CloudWatch are integrated so you can gather a variety of metrics. You can monitor these metrics using the Amazon CloudWatch console, Amazon CloudWatch's own command-line interface, or programmatically using the Amazon CloudWatch API. CloudWatch also allows you to set alarms when you reach a specified threshold for a metric.

For more information about using Amazon Cloudwatch and alarms, see the Amazon Cloudwatch Documentation.

AWS Management Console

To view Amazon DynamoDB information for your account

1. Sign in to the AWS Management Console and open the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Click **View Metrics**.

   The available Amazon DynamoDB metric options appear in the **Viewing** list.

   The metric options in the **Viewing** list serve as a filter for refining your results. The options differentiate metrics at the account level versus metrics at table level or operation level.
3. Select one of the following metric options from the **Viewing** list.

Viewing Option	Description
`All Metrics`	Account level metrics for all of your services.

Viewing Option	Description
`AWS/DynamoDB`	Account level metrics for all of your Amazon DynamoDB tables, such as `UserErrors`.
`AWS/DynamoDB, TableName`	Table level metrics, such as `ConsumedReadCapacityUnits` over a specified period of time.
`AWS/DynamoDB, TableName, Operation`	Metrics for a specified operation on the specified table, such as the `SuccessfulRequestLatency` for Scan operations over a specified period of time on a single table.
`AWS/DynamoDB, Operation`	Metrics for API calls across tables, such as `ThrottledRequests` for all BatchGetItem operations across several tables over a specified period of time. **Note** A BatchGetItem request on a single table does not show in these results. Use `AWS/DynamoDB, TableName, Operation` for a single operation on a single table.

Depending on the **Viewing** list selection, Amazon Cloudwatch displays a list of available metrics at the selected level.

4. Click a specific item to see more detail, such as a graph.

   Graphs showing the metrics for the selected item display in the bottom of the console.

Command Line Interface (CLI)

To gather disk storage statistics for a DB Instance

1. Install the Amazon Cloudwatch command line tool. For instructions and links about the tool, see Amazon Cloudwatch documentation.
2. Use the Amazon CloudWatch command line client commands to fetch information. The parameters for each command are listed in Amazon DynamoDB Metrics and Dimensions (p. 225).

   The following example uses the command **mon-get-stats** with the following parameters to determine how many requests exceeded your provisioned throughput during a specific time period.

```
PROMPT>mon-get-stats SuccessfulRequestLatency --aws-credential-file ./cre
dential-file-path.template --namespace "AWS/DynamoDB"
--statistics "Average" --start-time 2011-11-14T00:00:00Z --end-time 2011-
11-16T00:00:00Z --period 300
--dimensions "Operation=BatchGetItem"
```

API

Amazon CloudWatch also supports a Query API so you can request information programmatically.

1. Familiarize yourself with the Amazon Cloudwatch API and how to use it. For more information, see the Amazon CloudWatch Query API documentation and CloudWatch API Reference.
2. When a CloudWatch action requires a parameter that is specific to Amazon DynamoDB monitoring, such as `MetricName`, use the values listed in Amazon DynamoDB Metrics and Dimensions (p. 225).

   For example, call the Amazon CloudWatch API `GetMetricStatistics` using the following parameters:

   - *Statistics.member.1* = `Average`
   - *Dimensions.member.1* = `Operation=PutItem,TableName=TestTable`
   - *Namespace* = `AWS/DynamoDB`
   - *StartTime* = `2011-11-14T00:00:00Z`
   - *EndTime* = `2011-11-16T00:00:00Z`
   - *Period* = `300`
   - *MetricName* = `SuccessfulRequestLatency`

   Example What an API-based request looks like

   The following example shows an API-based request for Amazon CloudWatch metrics. However, the following is just to show the form of the request. This exact request won't work for you unless your data matches the specific time frame and metrics. You have to construct your own request based on your own data.

```
http://monitoring.amazonaws.com/
 ?SignatureVersion=2
 &Action=SuccessfulRequestLatency
 &Version=2010-08-01
 &StartTime=2011-11-14T00:00:00
 &EndTime=2011-11-16T00:00:00
 &Period=300
 &Statistics.member.1=Average
 &Dimensions.member.1=Operation=PutItem,TableName=TestTable
 &Namespace=AWS/DynamoDB
 &MetricName=SuccessfulRequestLatency
 &Timestamp=2011-10-15T17%3A48%3A21.746Z
 &AWSAccessKeyId=<AWS Access Key ID>
 &Signature=<Signature>
```

Amazon DynamoDB Metrics and Dimensions

Amazon DynamoDB Viewing Options

The metric options in the **Viewing** list serve as a filter for refining your results. The options differentiate metrics at the account level versus metrics at table level or operation level.

Viewing Option	Description
`All Metrics`	Account level metrics for all of your services.

Viewing Option	Description
`AWS/DynamoDB`	Account level metrics for all of your Amazon DynamoDB tables, such as `UserErrors`.
`AWS/DynamoDB, TableName`	Table level metrics, such as `ConsumedReadCapacityUnits` over a specified period of time.
`AWS/DynamoDB, TableName, Operation`	Metrics for a specified operation on the specified table, such as the `SuccessfulRequestLatency` for Scan operations over a specified period of time on a single table.
`AWS/DynamoDB, Operation`	Metrics for API calls across tables, such as `ThrottledRequests` for all BatchGetItem operations across several tables over a specified period of time. **Note** A single BatchGetItem request on a single table does not show in these results. Use `AWS/DynamoDB, TableName, Operation` for a single operation on a single table.

Amazon DynamoDB Dimensions and Metrics

The following metrics are available from the Amazon DynamoDB Service. The service only sends metrics when they have a non-zero value. For example, if no requests generating a 400 status code occur in a time period, you would see no data for the UserErrors metric that reports requests generating a 400 status code.

Note

The Statistic values available through Amazon CloudWatch, such as *`Average`* or *`Sum`*, are not always applicable to every metric. However, they are all available through the console, API, and command line client for all services. For each metric, be aware of the list of Valid Statistics for the Amazon DynamoDB metrics to track useful information. For example, Amazon CloudWatch can monitor each time an Amazon DynamoDB request is refused (the `ThrottledRequests` metric). It marks that event as one occurrence. If the request is retried and also refused, Amazon CloudWatch marks the second event as one occurrence, too. The *`Sum`* statistic is now `2`. But, the *`Average`* statistic for the `ThrottledRequests` metric is simply `1`, if a request is throttled in the specified time period, once or repeatedly. For the `ThrottledRequests` metric, use the listed Valid Statistics (either Sum or SampleCount) to see the trend of `ThrottledRequests` over a specified time period.

Metric	Description
`SuccessfulRequestLatency`	The number of successful requests in the specified time period. By default, `SuccessfulRequestLatency` provides the elapsed time for successful calls. You can see statistics for the Minimum, Maximum, or Average, over time. **Note** Cloudwatch also provides a SampleCount statistic: the total number of successful calls for a sample time period. View (namespace): `AWS/DynamoDB, TableName, Operation` Units: Milliseconds (or a count for SampleCount) Valid Statistics: Minimum, Maximum, Average, SampleCount
`UserErrors`	The number of requests generating a 400 status code (likely indicating a client error) response in the specified time period. View (namespace): `All Metrics` Units: Count Valid Statistics: Sum, SampleCount
`SystemErrors`	The number of requests generating a 500 status code (likely indicating a server error) response in the specified time period. View (namespace): `AWS/DynamoDB, TableName` Units: Count Valid Statistics: Sum, SampleCount
`ThrottledRequests`	The number of user requests that exceeded the preset provisioned throughput limits in the specified time period. View (namespace): `AWS/DynamoDB, TableName` Units: Count Valid Statistics: Sum, SampleCount

Metric	Description
`ConsumedReadCapacityUnits`	The amount of read capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used. For more information, see Provisioned Throughput in Amazon DynamoDB. View (namespace): `AWS/DynamoDB, TableName` **Note** Use the Sum value to calculate the provisioned 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 ConsumedReadCapacityUnits per second. You can compare the calculated value to the provisioned throughput value you provide Amazon DynamoDB. View (namespace): `AWS/DynamoDB, TableName` Units: Count Valid Statistics: Minimum, Maximum, Average, Sum
`ConsumedWriteCapacityUnits`	The amount of write capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used. For more information, see Provisioned Throughput in Amazon DynamoDB. **Note** Use the Sum value to calculate the provisioned 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. You can compare the calculated value to the provisioned throughput value you provide Amazon DynamoDB. View (namespace): `AWS/DynamoDB, TableName` Units: Count Valid Statistics: Minimum, Maximum, Average, Sum
`ReturnedItemCount`	The the number of items returned by a Scan or Query operation. View (namespace): `AWS/DynamoDB, TableName` Units: Count Valid Statistics: Minimum, Maximum, Average, SampleCount, Sum

Dimensions for Amazon DynamoDB Metrics

The metrics for Amazon DynamoDB are qualified by the values for the account, table name, or operation. Account level metrics display when you select **AWS/DynamoDB** as the viewing option. Otherwise, Amazon DynamoDB data can be retrieved along any of the following dimensions in the table below. Some metrics allow you to specify both a table name and operation, depending on the viewing option you specify.

Dimension	Description
`TableName`	This dimension limits the data you request to a specific table. This value can be any table name for the current account.
`Operation`	The operation corresponds to the Amazon DynamoDB service API, and can be one of the following: • `PutItem` • `DeleteItem` • `UpdateItem` • `GetItem` • `BatchGetItem` • `Scan` • `Query` For all of the operations in the current Amazon DynamoDB service API, see Operations in Amazon DynamoDB.

Exporting, Importing, Querying, and Joining Tables in Amazon DynamoDB Using Amazon EMR

Topics

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 Amazon DynamoDB, such as:

- Exporting data stored in Amazon DynamoDB to Amazon S3.
- Importing data in Amazon S3 to Amazon DynamoDB.
- Querying live Amazon DynamoDB data using SQL-like statements (HiveQL).
- Joining data stored in Amazon DynamoDB and exporting it or querying against the joined data.
- Loading Amazon 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 Amazon DynamoDB, and issue Hive commands to manipulate the data in Amazon 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 which 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 Amazon 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 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 an Amazon 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 Amazon 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 Amazon CloudWatch. If the request load is too high, you can relaunch the job flow and set the Read Percent Setting (p. 250) and Write Percent Setting (p. 250)to a lower values to throttle the Amazon EMR read and write operations. For information about Amazon DynamoDB throughput settings, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65).

Prerequisites for Integrating Amazon EMR with Amazon DynamoDB

To use Amazon Elastic MapReduce (Amazon EMR) and Hive to manipulate data in Amazon 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://aws.amazon.com, and clicking **Create an AWS Account**.
- An Amazon DynamoDB table that contains data.
- A customized version of Hive (version 0.7.1.3 or later) that includes connectivity to Amazon DynamoDB. These versions of Hive require the Amazon EMR AMI version 2.0 or later and Hadoop 0.20.205. Hive 0.7.1.3 is available by default when you launch an Amazon EMR job flow from the AWS Management Console or from a version of the Amazon EMR command line client (CLI) downloaded after 11 December 2011. If you launch a job flow using the AWS SDK or the API, you must explicitly set the AMI version to `latest` and the Hive version to `0.7.1.3`. 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 Elastic MapReduce Developer Guide*.
- Support for Amazon DynamoDB connectivity. This is loaded on 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 Amazon 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 job flow and run HiveQL queries against the Amazon 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 job flows, 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 an SSH client called PuTTY.
- (Optional) An Amazon EC2 key pair. This is only required for interactive job flows. 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 Amazon DynamoDB, you will need a key pair to connect to the Amazon EC2 instances launched by Amazon Elastic MapReduce (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.aws.amazon.com/ec2/.
2. In the Navigation pane, select a Region from the **Region** drop-down menu. This should be the same region that your Amazon 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**.

   You are prompted to download the key file.
6. Download the private key file and keep it 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 Getting an SSH Key Pair in the *Amazon EC2 User Guide*.

Step 2: Create a Job Flow

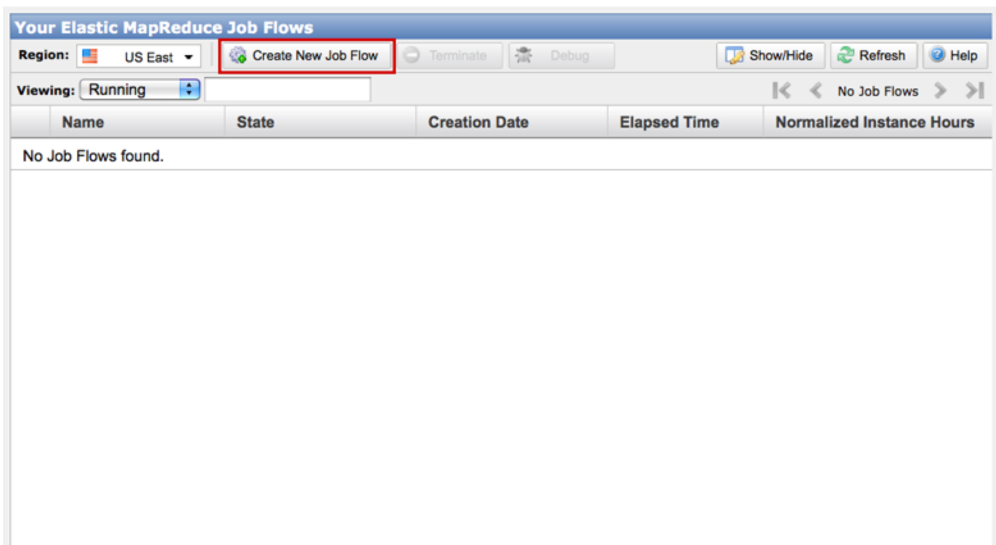
For Hive to run on Amazon Elastic MapReduce (Amazon EMR), you must create a job flow with Hive enabled. This sets up the necessary applications and infrastructure for Hive to connect to Amazon DynamoDB. The following procedures explain how to create an interactive Hive job flow from the AWS Management Console and the CLI.

Topics

To start a job flow using the AWS Management Console

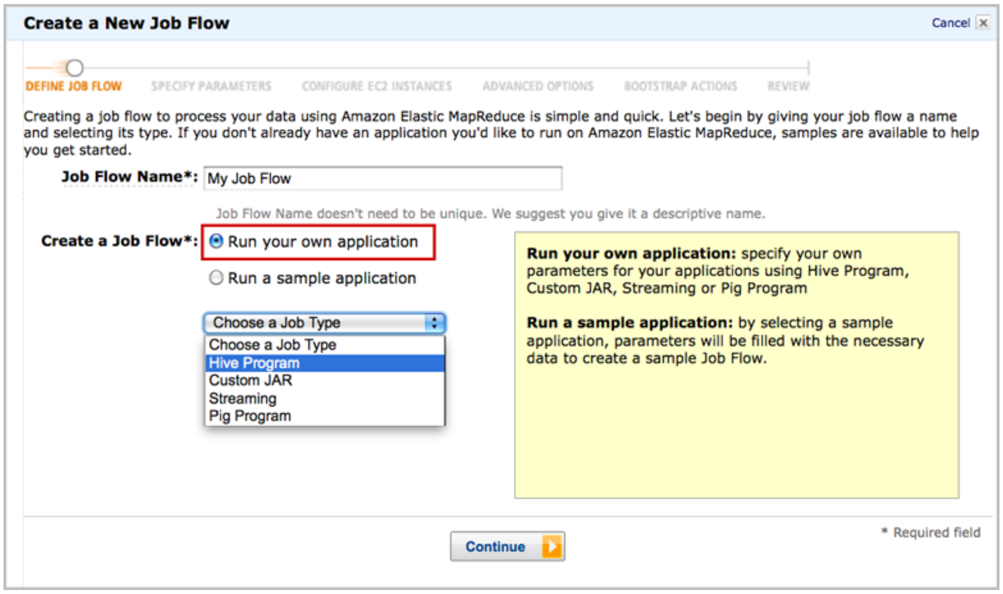
1. Open the Amazon Elastic MapReduce console at https://console.aws.amazon.com/elasticmapreduce/.

   This opens the Amazon Elastic MapReduce console which you can use to launch and manage job flows.
2. Select a region from the **Region** drop-down box. This is the region in which you'll create the Amazon EMR job flow. To avoid cross-region data transfer charges, this should be the same region that hosts your Amazon DynamoDB data. Similarly, if you are exporting data to Amazon S3, the Amazon S3 bucket should be in the same region as both the Amazon DynamoDB and the Amazon EMR job flow to avoid cross-region data transfer charges.
3. Click the **Create New Job Flow** button.



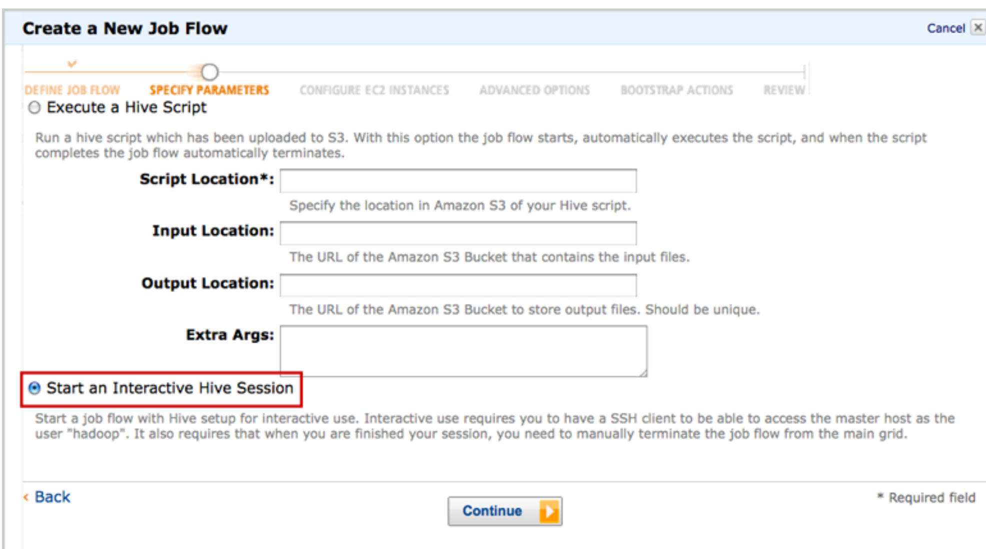
4. On the **DEFINE NEW JOB FLOW** page, do the following:

   - Give your Job Flow a name, such as "My Job Flow".
   - Select the **Run your own application** radio button.
   - In the **Choose a Job Type** drop-down menu, choose **Hive Program**.



Click **Continue**.

5. On the **SPECIFY PARAMETERS** page, select the **Start an Interactive Hive Session** radio button.



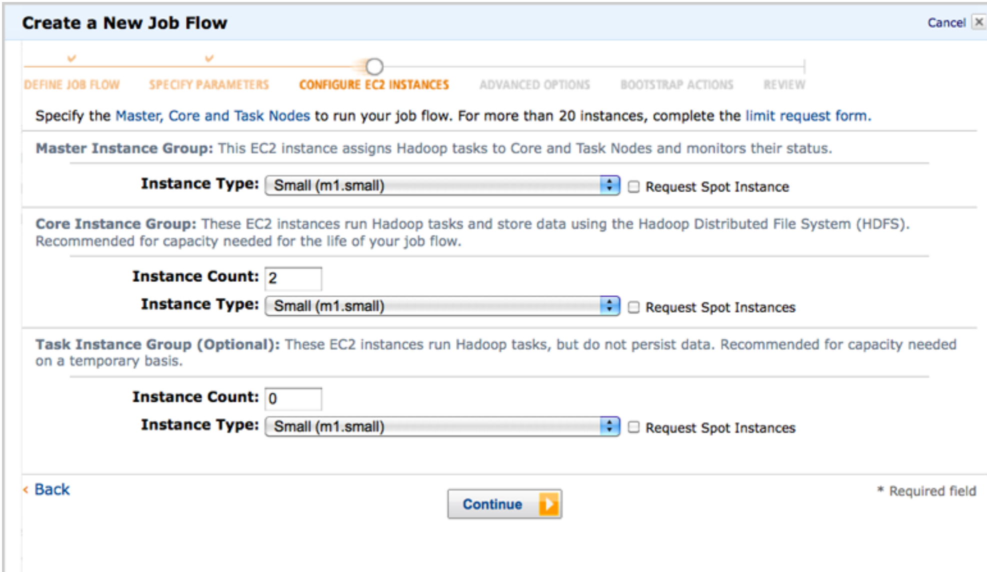
Hive is an open-source tool that runs on top of Hadoop to provide a way to query job flows using a simplified SQL syntax. Select an interactive session to issue commands from a terminal window.

Later, once you've established a set of queries that you'd like to run on a regular basis, you can save your queries as a script in an Amazon S3 bucket and have Amazon EMR run them for you without an interactive session.

Click **Continue**.

6. On the **CONFIGURE EC2 INSTANCES** page, set the number and type of instances to process the data in parallel.

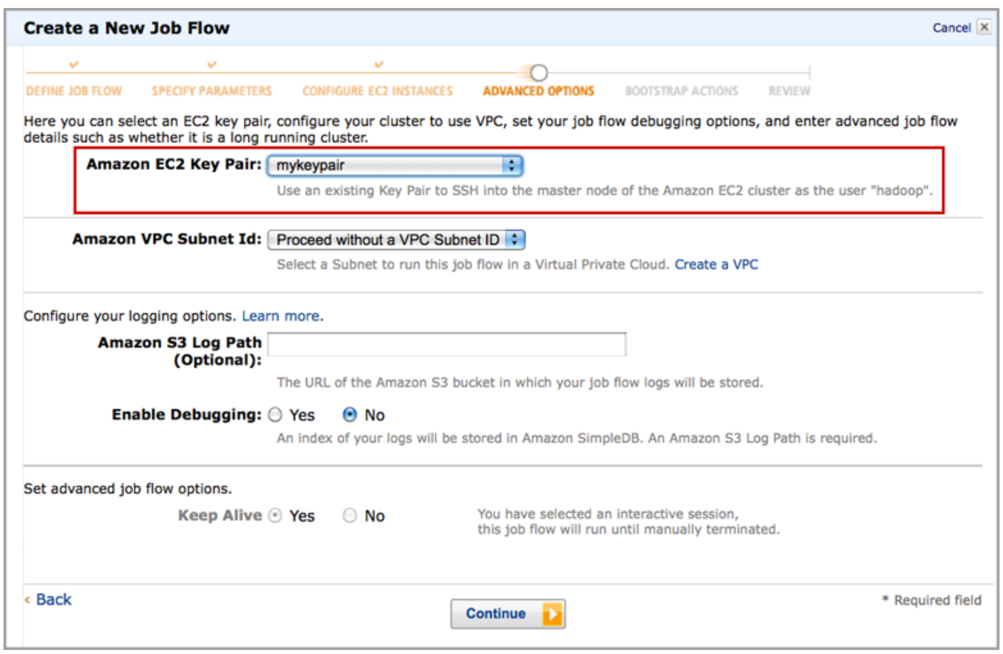
   In the **Master Instance Group**, for **Instance Type**, use an `m1.small` master node. In the **Core Instance Group**, for **Instance Count** use the default value 2 and for **Instance Type** use the default value `m1.small`. If you need more processing power, select larger options.



Click **Continue**.

7. On the **ADVANCED OPTIONS** page, select the key pair you created earlier in the **Amazon EC2 Key Pair** drop-down menu.

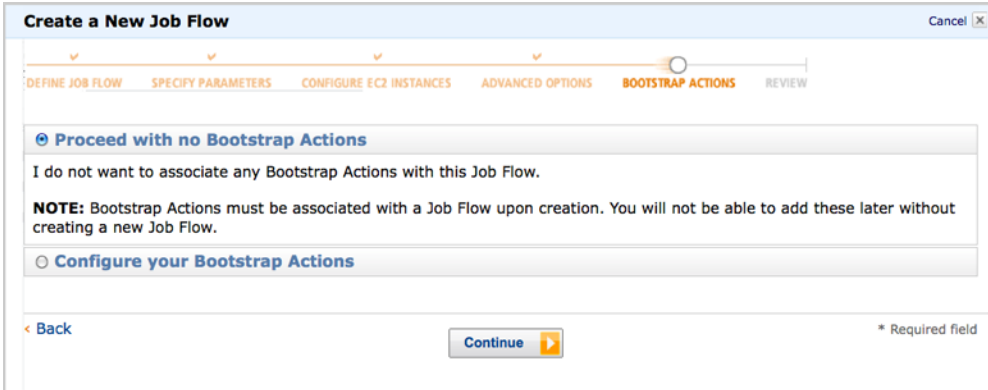
   Leave the rest of the settings on this page at the default values. For example, **Amazon VPC Subnet Id** should remain set to `Proceed without a VPC Subnet ID.`



Click **Continue**.

8. In the **Bootstrap Actions** dialog:

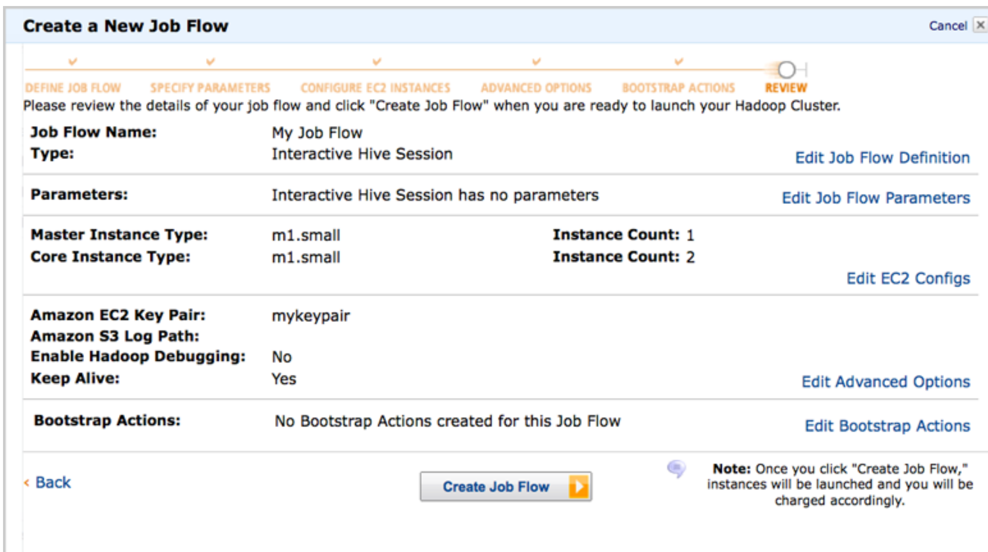
   Select the **Proceed with no Bootstrap Actions** radio button.



Click **Continue**.

9. In the **Review** dialog:

   Review the settings for your Job Flow.



Click **Create Job Flow**.

> **Note**
>
> When the confirmation window closes, your new job flow appears in the list of job flows in the Amazon Elastic MapReduce console with the status `STARTING`. If you do not see your job flow with the `STARTING` status, click **Refresh** to see the job flow. It takes a few minutes for Amazon EMR to provision the Amazon EC2 instances for your job flow. Your Job Flow is ready for use when the status is `WAITING`.

To start a Job Flow using a command line client

1. Download the Amazon EMR Ruby command line client (CLI). If you downloaded the Amazon EMR CLI before 11 December 2011, you will need to download and install the latest version to get support for AMI versioning, Amazon EMR AMI version 2.0, and Hadoop 0.20.205.

2. Install the command line client and set up your credentials. For information about how to do this, go to Sign Up and Install the Command Line Interface in the *Amazon Elastic MapReduce Developer Guide*.
3. Use the following syntax to start a new job flow, specifying your own values for the instance size and your own job flow name for `"myJobFlowName"`:

```
elastic-mapreduce --create --alive --num-instances 3 \
--instance-type m1.small \
--name "myJobFlowName" \
--hive-interactive --hive-versions 0.7.1.1 \
--ami-version latest \
--hadoop-version 0.20.205
```

   You must use the same account to create the Amazon EMR job flow that you used to store data in Amazon DynamoDB. This ensures that the credentials passed in by the CLI will match those required by Amazon DynamoDB.

Note

After you create the job flow, you should wait until its status is `WAITING` before continuing to the next step.

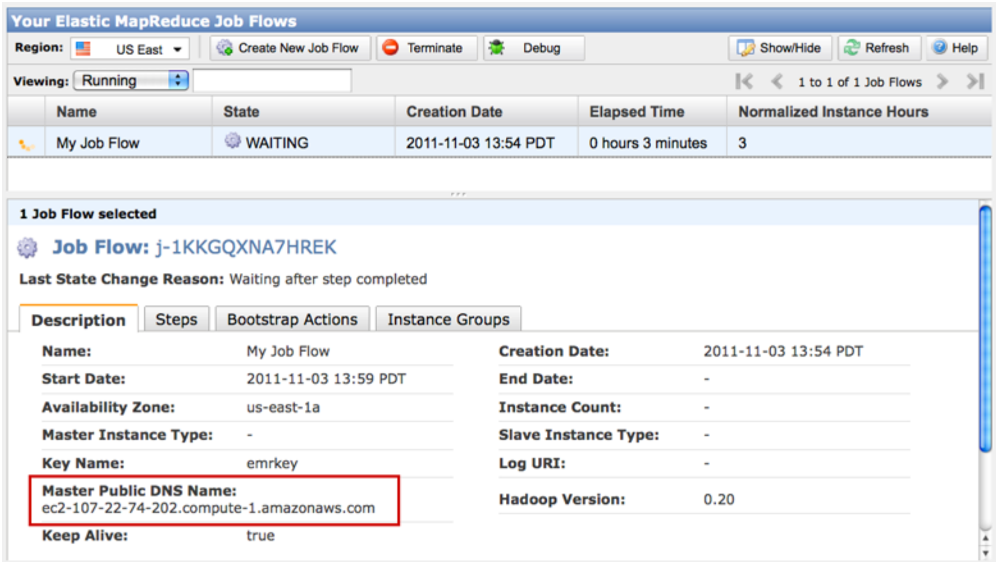
Step 3: SSH into the Master Node

When the job flow'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 SSH into the master node

1. Locate the **Master Public DNS Name**.

   In the Amazon Elastic MapReduce console, select the job from the list of running job flows in the `WAITING` state. Details about the job flow appear in the lower pane.



The DNS name you used to connect to the instance is listed on the Description tab as **Master Public DNS Name**. In the example above, the DNS name is `ec2-107-22-74-202.compute-1.amazonaws.com`. Use this name in the next step.

2. Use SSH to open up a terminal connection to the master node.

   Use the SSH application available on most Linux, Unix, and Mac OS X installations. Windows users can use an application called PuTTY to connect to the master node. The following are platform-specific instructions for opening an SSH connection.

   ### To connect to the master node using Linux/Unix/Mac OS X

   1. Open a terminal window. This is found at Applications/Utilities/Terminal on Mac OS X and at Applications/Accessories/Terminal on many Linux distributions.
   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 will return an error saying that your private key file is unprotected and will reject 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 *`ec2-107-22-74-202.compute-1.amazonaws.com`* with the Master Public DNS Name of your job flow and replace *`~/mykeypair.pem`* with the location and filename of your PEM file.

      ```
      ssh hadoop@ec2-107-22-74-202.compute-1.amazonaws.com -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.

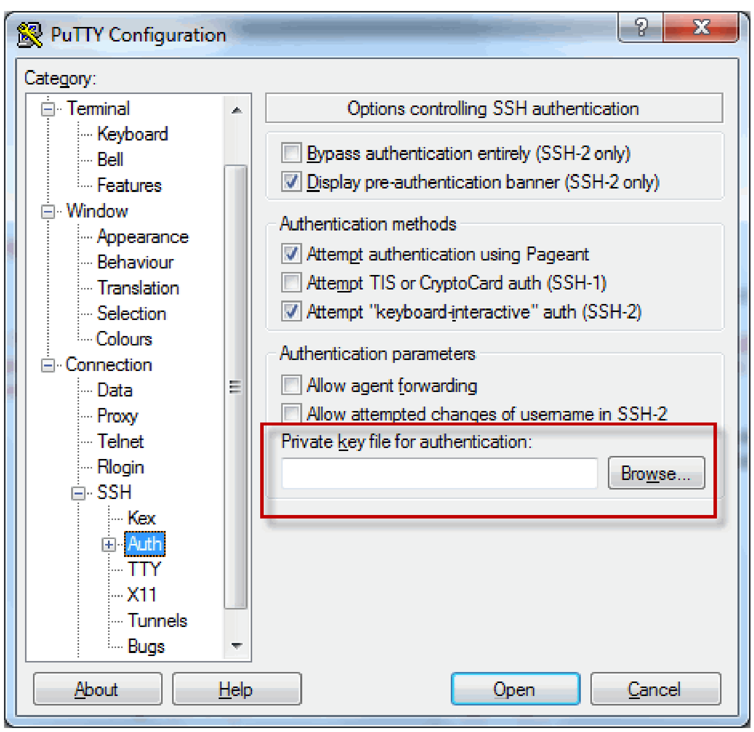
Note

If you are asked to log in, enter `hadoop`.

Now, you should see a Hadoop command prompt and you are ready to start a Hive interactive session.

To connect to the master node using 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. You only need to perform steps 1-9 the first time that you use the private key.
12. Start PuTTY.
13. 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`.
14. In the Category list, expand **Connection**, expand **SSH**, and then select **Auth**. The **Options controlling the SSH authentication** pane appears.



15. 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`.
16. Click **Open**.

    A PuTTY Security Alert pops up.
17. Click **Yes** for the PuTTY Security Alert.

    Note

    If you are asked to log in, enter `hadoop`.

Now, 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 Elastic MapReduce (Amazon EMR) job flows using a SQL-like language. Because we launched the job flow as a Hive application, Amazon EMR will install Hive on the Amazon EC2 instances it launches to process the job flow. To learn more about Hive, go to http://hive.apache.org/.

If you've followed the previous instructions to set up a job flow and SSH into the master node, you are ready to use Hive interactively.

To run Hive commands interactively

1. At the `hadoop` 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 Amazon 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 Amazon 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 an Amazon 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_attrib
ute1_name,hive_column2_name:dynamodb_attribute2_name...");
```

   When you create a table in Hive from Amazon 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 Amazon DynamoDB table named "dynamodbtable1". The Amazon DynamoDB table "dynamodbtable1" 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 hiveTableName (col1 string, col2 bigint, col3 ar
ray<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"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 Amazon 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 Amazon 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 Amazon DynamoDB. Provide `TBLPROPERTIES` with values for the *`dynamodb.table.name`* parameter and *`dynamodb.column.mapping`* parameter. These values *are* case-sensitive.

Note

All Amazon 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 Amazon DynamoDB. If you do not map the Amazon 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 will be null.

Then you can start running Hive operations on "hivetable1". Queries run against hivetable1" are internally run against the Amazon DynamoDB table "dynamodbtable1" of your Amazon DynamoDB account, consuming read or write units with each execution.

Sample HiveQL statements to perform tasks such as exporting or importing data from Amazon DynamoDB and joining tables are listed in Hive Command Examples for Exporting, Importing, and Querying Data in Amazon DynamoDB (p. 244).

You can also create a file that contains a series of commands, launch a job flow, and reference that file to perform the operations. For more information, see Interactive and Batch Modes in the *Amazon Elastic MapReduce 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, go to How to Use the Hadoop User Interface in the *Amazon Elastic MapReduce Developer Guide*.

```
hadoop job -kill job-id
```

Data Types for Hive and Amazon DynamoDB

The following table shows the available Hive data types and how they map to the corresponding Amazon DynamoDB data types.

Hive type	Amazon DynamoDB type
string	string (S)
bigint or double	number (N)
array	number set (NS) or string set (SS)

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 Amazon DynamoDB that has precision higher than is available in the Hive datatypes, using Hive to export, import, or reference the Amazon DynamoDB data could lead to a loss in precision or a failure of the Hive query.

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 job flow, these settings will have returned to the default values.

Hive Options	Description
dynamodb.throughput.read.percent	Set the rate of read operations to keep your Amazon DynamoDB provisioned throughput rate in the allocated range for your table. The value is between `0.1` and `1.5`, inclusively. 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 Amazon DynamoDB. 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.
dynamodb.throughput.write.percent	Set the rate of write operations to keep your Amazon DynamoDB provisioned throughput rate in the allocated range for your table. The value is between `0.1` and `1.5`, inclusively. 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 Amazon DynamoDB 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.
dynamodb.endpoint	Specify the endpoint in case you have tables in different regions. The default endpoint is `us-east-1`.
dynamodb.max.map.tasks	Specify the maximum number of map tasks when reading data from Amazon DynamoDB. This value must be equal to or greater than 1.
dynamodb.retry.duration	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;
```

Hive Command Examples for Exporting, Importing, and Querying Data in Amazon DynamoDB

The following examples use Hive commands to perform operations such as exporting data to Amazon S3 or HDFS, importing data to Amazon DynamoDB, joining tables, querying tables, and more.

Operations on a Hive table reference data stored in Amazon DynamoDB. Hive commands are subject to the Amazon DynamoDB table's provisioned throughput settings, and the data retrieved includes the data written to the Amazon DynamoDB table at the time the Hive operation request is processed by Amazon DynamoDB. If the data retrieval process takes a long time, some data returned by the Hive command may have been updated in Amazon 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 Amazon DynamoDB. If your Hive query references a table in Amazon DynamoDB, that table must already exist in Amazon DynamoDB, for example to write to or read from, the table must exist in Amazon DynamoDB before you run the query. For more information on creating and deleting tables in Amazon DynamoDB, go to Working with Tables in Amazon 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 Amazon DynamoDB

You can use Hive to export data from Amazon DynamoDB.

To export an Amazon DynamoDB table to an Amazon S3 bucket

- Create a Hive table that references data stored in Amazon 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 Amazon DynamoDB. You can use this to create an archive of your Amazon DynamoDB data in Amazon S3.

```
CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 ar
ray<string>)
```

```
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");


INSERT OVERWRITE DIRECTORY 's3://bucketname/path/subpath/' SELECT *
FROM hiveTableName;
```

To export an Amazon 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 Amazon DynamoDB to s3_export, the data will be 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 ar
ray<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");


CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col ar
ray<string>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

To export an Amazon 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 ar
ray<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"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 an Amazon 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 Amazon DynamoDB. This export operation is faster than exporting a Amazon 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 ar
ray<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");

SET dynamodb.throughput.read.percent=1.0;

INSERT OVERWRITE DIRECTORY 'hdfs:///directoryName' SELECT * FROM hiveTable
Name;
```

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 will need to use Hadoop to read this file. The following example shows how to export data from Amazon 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 ar
ray<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");


CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col ar
ray<string>)
STORED AS SEQUENCEFILE
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

Importing Data to Amazon DynamoDB

When you write data to Amazon DynamoDB using Hive you should ensure that the number of write capacity units is greater than the number of mappers in the job flow. For example, job flows that run on m1.xlarge EC2 instances produce 8 mappers per instance. In the case of a job flow 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 job flow, the Hive write operation may consume all of the write throughput, or attempt to consume more throughput than is provisioned. For details about the number of mappers produced by each EC2 instance type, go to Task Configuration (AMI 2.0).

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 Amazon DynamoDB table, it will be overwritten. If no item with the key exists in the target Amazon DynamoDB table, the item is inserted.

To import a table from Amazon S3 to Amazon DynamoDB

- You can use Amazon Elastic MapReduce (Amazon EMR) and Hive to write data from Amazon S3 to Amazon 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" = "dynamodbtable1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");

INSERT OVERWRITE TABLE 'hiveTableName' SELECT * FROM s3_import;
```

To import a table from HDFS to Amazon DynamoDB

- You can use Amazon EMR and Hive to write data from HDFS to Amazon DynamoDB.

```
CREATE EXTERNAL TABLE hdfs_import(a_col string, b_col bigint, c_col ar
ray<string>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 'hdfs:///directoryName';

CREATE EXTERNAL TABLE hiveTableName (col1 string, col2 bigint, col3 ar
ray<string>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "dynamodbtable1",
"dynamodb.column.mapping" = "col1:name,col2:year,col3:holidays");

INSERT OVERWRITE TABLE 'hiveTableName' SELECT * FROM hdfs_import;
```

Querying Data in Amazon DynamoDB

The following examples show the various ways you can use Amazon EMR to query data stored in Amazon 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 Amazon DynamoDB. The SELECT statement then uses that table to query data stored in Amazon 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_pur
chased: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_pur
chased:Items");

SELECT customerId, max(total_cost) from hive_purchases GROUP BY customerId
 HAVING count(*) > 3;
```

To join two Amazon DynamoDB tables

- The following example maps two Hive tables to data stored in Amazon 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 Amazon 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_pur
chased: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,custom
erAddress: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 Amazon DynamoDB. The following example joins together customer data stored as a CSV file in Amazon S3 with order data stored in Amazon 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_pur
chased: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 once, at the beginning of the Hive session.

Optimizing Performance for Amazon EMR Operations in Amazon DynamoDB

Amazon Elastic MapReduce (Amazon EMR) operations on an Amazon 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 Amazon DynamoDB table to minimize the possibility of exceeding your provisioned throughput. At the end of each Hive query, Amazon EMR returns information about the job flow used to process the query, including how many times your provisioned throughput was exceeded. You can use this information, as well as Amazon CloudFront metrics about your Amazon DynamoDB throughput to better manage the load on your Amazon DynamoDB table in subsequent requests.

The following factors influence Hive query performance when working with Amazon DynamoDB tables.

Read Percent Setting

By default, Amazon EMR manages the request load against your Amazon 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.throughput.read.percent`* parameter when you set up the Hive table. For more information about setting the read percent parameter, see Hive Options (p. 242).

Write Percent Setting

By default, Amazon EMR manages the request load against your Amazon 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.throughput.write.percent`* parameter when you set up the Hive table. For more information about setting the write percent parameter, see Hive Options (p. 242).

Retry Duration Setting

By default, Amazon EMR will re-run 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 (p. 242).

Number of Map Tasks

The mapper daemons that Hadoop launches to process your requests to export and query data stored in Amazon 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 Amazon 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 job flow *or* increase the number of mapper daemons running on each EC2 instance.

You can increase the number of EC2 instances in a job flow by stopping the current job flow 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 job flow from the Amazon Elastic MapReduce console, or with the `--num-instances` option if you're launching the job flow from the CLI.

The number of map tasks run on an instance depends on the EC2 instance type. For a list of the supported EC2 instance types and the number of mappers each one provides, go to Task Configuration.

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 job flow to run out of memory. To set `mapred.tasktracker.map.tasks.maximum`, launch the job flow 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 -s,mapred.tasktracker.map.tasks.maximum=10
```

For more information about bootstrap actions, go to Using Custom Bootstrap Actions in the *Amazon Elastic Map Reduce 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 Amazon 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 Amazon 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 Amazon 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 Amazon DynamoDB table when there is lower demand on the Amazon 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 Amazon DynamoDB database.

Time-Based Tables

If the data is organized as a series of time-based Amazon 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 Amazon 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 Amazon DynamoDB. This will conserve 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 on how to do this, go to How to Monitor Hadoop on a Master Node and How to Use the Hadoop User Interface in the *Amazon Elastic MapReduce Developer Guide.*

Controlling Access to Amazon DynamoDB Resources

Amazon DynamoDB integrates with AWS Identity and Access Management (IAM), a service that lets your organization do the following:

- Create users and groups under your organization's AWS account
- Easily share your AWS account resources between the users in the account
- Assign unique security credentials to each user
- Granularly control users access to services and resources
- Get a single AWS bill for all users under the AWS account

For general information about IAM, go to:

- Identity and Access Management (IAM)
- AWS Identity and Access Management Getting Started Guide
- Using AWS Identity and Access Management

For specific information about how you can control User access to Amazon DynamoDB, go to Integrating with Other AWS Products in *Using AWS Identity and Access Management*.

For Amazon DynamoDB, IAM lets you give other users of your AWS Account access to Amazon DynamoDB tables within the AWS Account. For example, Joe can create an Amazon DynamoDB table, and then write an IAM policy specifying which Users in his AWS Account can access that table. Joe can't give another AWS Account (or Users in another AWS Account) access to his AWS Account's Amazon DynamoDB tables.

For examples of policies that cover Amazon DynamoDB operations and resources, see Example Policies for Amazon DynamoDB (p. 254).

Amazon Resource Names (ARNs) for Amazon DynamoDB

Amazon DynamoDB supports IAM policies using ARNs per table, and require the account ID, as well. Use the following ARN Resource format.

```
"Resource": arn:aws:dynamodb:<region>:<accountID>:table/<tablename>
```

Note

The ListTables API, which lists the table names owned by the current account making the request for the current the region, is the only operation that does not support resource-level ARN policies.

Amazon DynamoDB Actions

In an IAM policy, you can specify any and all operations that Amazon DynamoDB offers. You must prefix each action name with the lowercase string `dynamodb:`. For example: `dynamodb:GetItem`, `dynamodb:Query`, `dynamodb:*` (for all Amazon DynamoDB operations). For a list of the operations, see API Reference for Amazon DynamoDB (p. 371).

Amazon DynamoDB Keys

Amazon DynamoDB implements the following policy keys, but no product-specific ones. For more information about policy keys, see Condition.

AWS-Wide Policy Keys

- `aws:CurrentTime` (for date/time conditions)
- `aws:EpochTime` (the date in epoch or UNIX time, for use with date/time conditions)
- `aws:SecureTransport` (Boolean representing whether the request was sent using SSL)
- `aws:SourceIp` (the requester's IP address, for use with IP address conditions)
- `aws:UserAgent` (information about the requester's client application, for use with string conditions)

If you use `aws:SourceIp`, and the request comes from an Amazon EC2 instance, we evaluate the instance's public IP address to determine if access is allowed.

For services that use only SSL, such as Amazon RDS and Amazon Route 53, the `aws:SecureTransport` key has no meaning.

The key names are case insensitive. For example, `aws:CurrentTime` is equivalent to `AWS:currenttime`.

Example Policies for Amazon DynamoDB

This section shows several simple policies for controlling User access to Amazon DynamoDB tables.

Example 1: Allow a group to use any Amazon DynamoDB actions on all tables

In this example, we create a policy that lets the group use any operation on any of the AWS Account's tables.

```
{
   "Statement":[{
      "Effect":"Allow",
      "Action":"dynamodb:*",
      "Resource":"*"
      }
   ]
}
```

Example 2: Allow a group to get data from items in the AWS Account's tables

In this example, we create a policy that lets the group use only the `GetItem` and `BatchGetItem` operations with any of the AWS Account's tables.

```
{
   "Statement":[{
      "Effect":"Allow",
      "Action":["dynamodb:GetItem","dynamodb:BatchGetItem"],
      "Resource":"*"
      }
   ]
}
```

Example 3: Allow one account to put, update, and delete items in one table

In this example, we create a policy that lets the account identified by ID number `123456789` use the `PutItem`, `UpdateItem` and `DeleteItem` actions with only one of the AWS Account's tables called "books_table".

```
{
   "Statement":[{
      "Effect":"Allow",
     "Action":["dynamodb:PutItem","dynamodb:UpdateItem","dynamodb:DeleteItem"],

      "Resource":"arn:aws:dynamodb:us-east-1:123456789:table/books_table"
      }
   ]
}
```

Example 4: Deny a partner operations that change data

There's no way to share a table with a different AWS Account, so the partner must work with your table as a User within your own AWS Account.

In this example, we create an IAM User for the partner, and create a policy for the User that gives access to all the operations except those that edit data; essentially, they have read-only access.

> **Note**
>
> Instead of attaching the policy to the User, you could create a group for the partner, put the User in the group, and assign the policy to the group.

```
{
   "Statement":[{
         "Effect":"Allow",
         "Action":["dynamodb:*"],
         "Resource":"*"
      },
      {
         "Effect":"Deny",
         "Action":["dynamodb:CreateTable","dynamodb:DeleteItem", "dynamodb:De
leteTable"
   "dynamodb:PutItem", "dynamodb:UpdateItem", "dynamodb:UpdateTable"],
         "Resource":"*"
      }
   ]
}
```

Limits in Amazon DynamoDB

Following is a table that describes current limits within Amazon DynamoDB (or no limit, in some cases, for your information).

Table name	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 limit in number of bytes or items.
Tables per account	By default, the number of tables per account is limited to 256. However, you can request an increase in this limit. For more information, go to Amazon DynamoDB Limit Increase Form.
Provisioned Throughput	DynamoDB is designed to scale without limits. However, if you wish to exceed throughput rates of 10,000 write capacity units or 10,000 read capacity units for an individual table, you can request an increase in these limits. For more information, go to Amazon DynamoDB Limit Increase Form. By default, there is a limit of 20,000 write capacity units or 20,000 read capacity units per account. However, you can request an increase in these limits using the same form.
Provisioned Throughput minimum per table	5 ReadCapacityUnits and 5 WriteCapacityUnits.
UpdateTable provisioned throughput change minimum (reduction or increase)	If either ReadCapacityUnits or WriteCapacityUnits is reduced or increased, it must change by at least 10%.
UpdateTable provisioned throughput reduction frequency maximum	Once per UTC calendar day. If you decrease either ReadCapacityUnits or WriteCapacityUnits, the one reduction counts as your table's allowed decrease for that day. You can decrease both values at the same time in the same UpdateTable request, but not each value, separately, in a single day.
UpdateTable provisioned throughput increase maximum	If either ReadCapacityUnits or WriteCapacityUnits is increased, it can only be increased up to twice the current value.

UpdateTable provisioned throughput increase frequency maximum	No limit.
Maximum concurrent Control Plane API requests (includes cumulative number of tables in the *CREATING* or *DELETING* state)	10. Except, you can have only one table in the *UPDATING* state at a time.
Item size	Cannot exceed 64KB 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.
Attribute name (for primary key attributes only)	Names must be between 1 and 255 characters long, inclusive. They can be any UTF-8 encodable character, and the total size of the UTF-8 string after encoding can't exceed 255 bytes.
Attribute values	Attribute values cannot be null or empty.
Attribute name-value pairs per item	The cumulative size of attributes per item must be under 64KB.
Hash primary key attribute value	2048 bytes
Range primary 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	A number can have up to 38 digits precision and can be between 10^-128 to 10^+126.
Maximum number of values in a String Set or Number Set	No limit on the quantity of values, as long as the item containing the values fits within the 64KB item limit.

Using the AWS SDKs with Amazon DynamoDB

Topics

Amazon Web Services provides SDKs for you to develop applications for Amazon DynamoDB. The AWS SDKs for Java, PHP, and .NET wrap the underlying Amazon 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 Amazon DynamoDB, including Android, iOS, and Ruby. For links to the complete set of AWS SDKs, see Sample Code & Libraries.

The AWS SDKs require an active AWS account. You should follow the steps in Getting Started with Amazon DynamoDB (p. 11) to get set up and familiar with using the AWS SDKs for Amazon DynamoDB.

Using the AWS SDK for Java with Amazon DynamoDB

Topics

The AWS SDK for Java provides an API for the Amazon DynamoDB item and table operations. The API gives you the option of using a low-level or high-level API.

Low-Level API

The low-level APIs correspond closely to the underlying Amazon DynamoDB operations. The low-level API allows you to perform the same operations that you can perform using Amazon DynamoDB operations such as create, update, and delete tables, and create, read, update, and delete items.

High-Level API

The high-level API uses Object Persistence programming techniques to map Java objects to Amazon DynamoDB tables and attributes. You cannot create tables using the high-level API, but you can create, read, update, and delete table items. You can the high-level API if you have an existing code-base that you want to leverage by mapping to Amazon DynamoDB tables.

Note

The low-level API and high-level API provide thread-safe clients for accessing Amazon DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

The Java API Organization

The following packages in the AWS SDK for Java provide the API:

- **com.amazonaws.services.dynamodb—**Provides the implementation APIs for Amazon DynamoDB item and table operations.
  For example, it provides methods to create, update, and delete tables, and create, upload, read, and delete table items.
- **com.amazonaws.services.dynamodb.model—**Provides the low-level API classes to create requests and process responses.
  For example, it includes the `GetItemRequest` class to describe your get item request, the `PutItemtRequest` class to describe your item put requests, and the `DeleteItemRequest` class to describe your item delete request.
- **com.amazonaws.services.dynamodb.datamodeling—**Provides the high-level API operations.
  The high-level API allows you to use Object Persistence programming techniques to map Java objects to Amazon DynamoDB tables and perform create, read, update, and delete actions on your table items. The high-level API provides the `DynamoDBMapper` which is an object mapper class that use to map between your Java classes and your tables in DynamoDB.

For more information about the AWS SDK for Java API, go to AWS SDK for Java API Reference.

Running Java Examples for Amazon DynamoDB

General Process of Creating Java Code Examples (Using Eclipse)

1	Create a new **AWS Java** project in Eclipse. Note that the project is pre-configured; it includes the AWS SDK for Java jar files, and also the AwsCredentials.properties file for your AWS security credentials.

2	In the **Create an AWS Java Project** dialog box, enter project name in the **Project name** field and select one of the toolkit accounts from the **Select Account** drop-down list. To add a new account to the toolkit, click **Configure AWS accounts...**.
	Eclipse adds the AWS credentials associated with the toolkit account that you select to the AwsCredentials.properties file as shown in the following example:
	`#Insert your AWS Credentials from http://aws.amazon.com/security-credentials`
	`#Wed Jan 04 19:02:08 PST 2012`
	`secretKey=*** Your Account Secret Access Key***`
	`accessKey=***Your Account Access Key ID***`

3	Copy the code from the section that you are reading to your project.
4	Run the code.

Setting the Endpoint

By default, AWS SDK for Java sets the endpoint to `https://dynamodb.us-east-1.amazonaws.com`. You can also set the endpoint explicitly as shown in the following Java code snippet.

```
client = new AmazonDynamoDBClient(credentials);
client.setEndpoint("https://dynamodb.us-east-1.amazonaws.com");
```

For a current list of supported regions and endpoints, see Regions and Endpoints.

Using the Object Persistence Model with Amazon DynamoDB

Topics

The AWS SDK for Java provides a high-level API, the Object Persistence model, that enables you to map your client-side classes to the Amazon DynamoDB tables. The individual object instances then map to items in a table. The Object Persistence model provides the `DynamoDBMapper` class that provides an entry point to Amazon DynamoDB. This class provides you with a connection to the Amazon DynamoDB database and enables you to access your tables, perform various create, read, update and delete (CRUD) operations, and execute queries.

To map your classes to tables, the Object Persistence model provides a set of annotation types.

Note

The Object Persistence model enables you to perform data operations such as save items, update items, delete items, and query the tables. However, the model does not provide API to create, update, or delete tables. Only the low-level API enables you to create, update, and delete tables. For more information, see Working with Tables Using the AWS SDK for Java Low-Level API for Amazon DynamoDB (p. 73).

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 example. The code snippet defines a `CatalogItem` class and uses the annotations that are defined by Amazon DynamoDB to establish the mapping.

```
@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 snippet, 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, the `DynamoDBContext` does not include `SomeProp` property and also it does not return this attribute when you retrieve an item from the table.

In addition to mapping properties of the Java types such as integer and string, you can use the Object Persistence model to map any arbitrary data as long as you provide an appropriate converter to map it to the Amazon DynamoDB types. To learn about mapping arbitrary types, see Mapping Arbitrary Data with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model (p. 275).

The Object Persistence model also supports optimistic locking using a version field in the class. The `@DynamoDBVersionAttribute` annotation type on a property identifies it as a version field. For more information about using versioning, see Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model (p. 273).

Amazon DynamoDB Annotations

The following table lists the annotations that Amazon DynamoDB offers for you to map your classes and properties to tables and attributes.

Note

In the following table, only the `DynamoDBTable` and the `DynamoDBHashKey` are the required tags.

<table>
<tr><th>Declarative Tag (Annotation)</th><th>Description</th></tr>
<tr><td>@DynamoDBTable</td><td>

Identifies the target table in Amazon DynamoDB. For example, the following Java code snippet defines a class `Developer` and maps it to the People table in Amazon DynamoDB.

```
@DynamoDBTable(tableName="People")
public class Developer { ...}
```

This annotation can be inherited or overridden.

- The `@DynamoDBTable` annotation can be inherited. Any new class that inherits from the `Developer` class also maps to the People table. For example, assume that you create a `Lead` class that inherits from the `Developer` class. Because you mapped the `Developer` class to the People table, the `Lead` class objects are also stored in the same table.
- The `@DynamoDBTable` can also be overridden. Any new class that inherits from the `Developer` class by default maps to the same People table. However, you can override this default mapping. For example, if you create a class that inherits from the `Developer` class, you can explicitly map it to another table by adding the `@DynamoDBTable` annotation as shown in the following Java code snippet.

```
@DynamoDBTable(tableName="Managers")
public class Manager : Developer { ...}
```

</td></tr>
<tr><td>@DynamoDBIgnore</td><td>

Indicates to the `DynamoDBMapper` instance that the associated property should be ignored. When saving data to the table, the `DynamoDBMapper` does not save this property to the table.

</td></tr>
<tr><td>@DynamoDBAttribute</td><td>

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 `DynamoDBAttribute` maps the `BookAuthors` property to the Authors attribute name in the table.

```
@DynamoDBAttribute(attributeName = "Authors")
public List<String> getBookAuthors() { return
BookAuthors; }
public void setBookAuthors(List<String> BookAu
thors) { this.BookAuthors = BookAuthors; }
```

The `DynamoDBMapper` uses Authors as the attribute name when saving the object to the table.

</td></tr>
</table>

<table>
<tr><th>Declarative Tag (Annotation)</th><th>Description</th></tr>
<tr><td><code>@DynamoDBHashKey</code></td><td>
Maps a class property to the hash attribute of the table. The property must be one of the supported String or Numeric type and cannot be a collection type.
Assume that you have a table, ProductCatalog, that has Id as the primary key. The following Java code snippet defines a <code>CatalogItem</code> class and maps its <code>Id</code> property to the primary key of the ProductCatalog table using the <code>@DynamoDBHashKey</code> tag.
<pre><code>@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.
}
</code></pre>
</td></tr>
<tr><td><code>@DynamoDBRangeKey</code></td><td>
Maps a class property to the range key attribute of the table. 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 <code>Reply</code> class and maps it to the Reply table. It uses both the <code>@DynamoDBHashKey</code> and <code>@DynamoDBRangeKey</code> tags to identify class properties that map to the primary key.
<pre><code>@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.
}
</code></pre>
</td></tr>
</table>

<table>
<tr><th>Declarative Tag (Annotation)</th><th>Description</th></tr>
<tr><td><code>@DynamoDBAutoGeneratedKey</code></td><td>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.

The following snippet demonstrates using auto-generated keys.
<pre>
@DynamoDBTable(tableName="AutoGeneratedKey
sExample")
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.pay
load };
    public String setPayload(String payload) {
this.payload = payload };

    public static void saveItem() {
        AutoGeneratedKeys obj = new AutoGener
atedKeys();
        obj.setPayload("abc123");

        // id field is null at this point
        DynamoDBMapper mapper = new DynamoDBMap
per(dynamoDBClient);
        mapper.save(obj);

        System.out.println("Object was saved with
 id " + obj.getId());
    }
}
</pre></td></tr>
<tr><td><code>@DynamoDBVersionAttribute</code></td><td>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 Amazon DynamoDB Data Types (p. 6). For more information about versioning, see Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model (p. 273).</td></tr>
</table>

DynamoDBMapper Class

The `DynamoDBMapper` class is the entry point to the Amazon DynamoDB database. It provides a connection to Amazon 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 Amazon DynamoDB.

Method	Description

`save`	Saves the specified object to the table. The object that you wish to save is the only required parameter for this method. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object.
	
	If an item that has the same primary key does not exist, this method creates a new item in the table. If an item that has the same primary key exists, it updates the existing item. String hash and range keys annotated with `@DynamoDBAutoGeneratedKey` are given a random universally unique identifier (UUID) if left uninitialized. Version fields annotated with `@DynamoDBVersionAttribute` will be incremented by one. Additionally, if a version field is updated or a key generated, the object passed in is updated as a result of the operation.
	
	By default, only attributes corresponding to mapped class properties are updated; any additional existing attributes on an item are unaffected. However, if you specify `SaveBehavior.CLOBBER`, you can force the item to be completely overwritten.
	
	`CatalogItem item = mapper.save(obj,`
	`               new DynamoDBMapperConfig(DynamoDBMapperConfig.Save`
	`Behavior.CLOBBER));`
	
	If you have versioning enabled, then the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model (p. 273).

`load`	Retrieves an item from a table. You must provide the primary key of the item that you wish to retrieve. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object. For example, you can optionally request consistent read to ensure that this method retrieves only the latest item values as shown in the following Java statement.
	
	`CatalogItem item = mapper.load(CatalogItem.class, item.getId(),`
	`                new DynamoDBMapperConfig(DynamoDBMapperConfig.Con`
	`sistentReads.CONSISTENT));`
	
	By default, Amazon DynamoDB returns the item that has values that are eventually consistent. For information about the eventual consistency model in Amazon DynamoDB, see Data Read and Consistency Considerations (p. 7).

`delete`	Deletes an item from the table. You must pass in an object instance of the mapped class. If you have versioning enabled, then the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model (p. 273).

Method	Description
query	

Method	Description
	Enables the querying of 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.

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.

```
Reply ( Id, ReplyDateTime, ... )
```

Now, assume that you created an Object Persistence model that includes a Reply class that maps to the table.

The following Java code snippet uses the `DynamoDBMapper` instance to query the table to find all replies in the past two weeks for a specific thread subject.

```
String forumName = "Amazon 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(twoWeek
sAgoStr.toString()));

DynamoDBQueryExpression queryExpression = new DynamoDBQueryExpres
sion(
        new AttributeValue().withS(hashKey));

queryExpression.setRangeKeyCondition(rangeKeyCondition);
queryExpression.setHashKeyValue(new AttributeValue().withS("Reply
DateTime"));

List<Reply> latestReplies = mapper.query(Reply.class, queryExpres
sion);
```

The query returns a collection of `Reply` objects.

Note

If your table's primary key is made of only a hash attribute, then you cannot use the `query` method. Instead, you can use the `load` method and provide the hash attribute to retrieve the item.

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td></td><td>The <code>query</code> method returns the "lazy-loaded" collection. That is, initially it returns only one page of results. It makes a service call for the next page when needed.</td></tr>
<tr><td><code>scan</code></td><td>

Scans an entire table. You can specify optional condition filters items based on one or more `Condition` instances, and you can specify a filter expression for any item attributes.

Assume that you have a table, Thread, that stores forum thread information including Subject (part of the composite primary key) and if the thread is answered.

```
Thread ( ForumName, Subject, ..., Answered )
```

If you have an Object Persistence model for this table, then you can use the `DynamoDBContext` to scan the table. For example, the following Java code snippet filters the Thread table to retrieve all the unanswered threads. The scan condition identifies the attribute and a condition.

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpres
sion();

Map<String, Condition> scanFilter = new HashMap<String, Condi
tion>();
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, scanEx
pression);
```

The `scan` method returns the "lazy-loaded" Thread collection. That is, initially it returns only one page of results. It makes a service call for the next page when needed.

</td></tr>
<tr><td><code>batchDelete</code></td><td>

Deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code snippet deletes two items (books) from the ProductCatalog table.

```
Book book1 = mapper.load(Book.class, 901);
Book book2 = mapper.load(Book.class, 902);
mapper.batchDelete(Arrays.asList(book1, book2));
```

</td></tr>
</table>

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td><code>batchSave</code></td><td>

Saves objects to one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code snippet saves two items (books) to the ProductCatalog table.

```
Book book1 = new Book();
book1.id = 901;
book1.productCategory = "Book";
book1.title = "Book 901 Title";

Book book2 = new Book();
book2.id = 902;
book2.productCategory = "Book";
book2.title = "Book 902 Title";

mapper.batchSave(Arrays.asList(book1, book2));
```

</td></tr>
<tr><td><code>batchWrite</code></td><td>

Saves objects to and deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees or support versioning (conditional puts or deletes).

The following Java code snippet writes a new item to the Forum table, writes a new item to theThread table, and deletes an item from the ProductCatalog table.

```
// Create a Forum item to save
Forum forumItem = new Forum();
forumItem.name = "Test BatchWrite Forum";

// Create a Thread item to save
Thread threadItem = new Thread();
threadItem.forumName = "AmazonDynamoDB";
threadItem.subject = "My sample question";

// Load a ProductCatalog item to delete
Book book3 = mapper.load(Book.class, 903);

List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
List<Book> objectsToDelete = Arrays.asList(book3);

mapper.batchWrite(objectsToWrite, objectsToDelete);
```

</td></tr>
<tr><td><code>count</code></td><td>Evaluates the specified scan expression and returns the count of matching items. No item data is returned.</td></tr>
<tr><td><code>marshallIntoObject</code></td><td>Utility method to transform a result from the low-level API into a domain object.</td></tr>
</table>

Specifying Optional Configuration Information to the DynamoDBMapper

The Object Persistence model provides the `DynamoDBMapper` for you to communicate with Amazon DynamoDB. When you create a mapper instance, you can specify optional configuration information using the `DynamoDBMapperConfig` class. You can specify the following arguments for an instance of `DynamoDBMapperConfig`:

- A `DynamoDBMapperConfig.SaveBehavior` enumeration value - Specifies how `DynamoDBMapper` should deal with attributes during save operations. Specifying the value `DynamoDBMapperConfig.SaveBehavior.UPDATE` will not affect unmodeled attributes on a save operation. All modeled attributes are updated. Primitive number types (byte, int, long) are set to 0. Object types are set to null. Specifying the value `DynamoDBMapperConfig.SaveBehavior.CLOBBER` will clear and replace all attributes, included unmodeled ones, (delete and recreate) on save. Versioned field constraints will also be disregarded. If you do not specify configuration information for your mapper instance, then the default `DynamoDBMapperConfig.SaveBehavior.UPDATE` is used.
- A `DynamoDBMapperConfig.ConsistentReads` enumeration value - If you specify `DynamoDBMapperConfig.ConsistentReads.CONSISTENT` , then the `DynamoDBMapper` includes a consistent read request. When retrieving data using the `load`, `query`, or `scan` operations, you can optionally add this parameter. Consistent reads have implications for performance and billing; see the product detail page for more information. Instead of consistent reads, you can specify eventual consistency with the `DynamoDBMapperConfig.ConsistentReads.EVENTUAL` enumeration value. If you do not specify configuration information for your mapper instance, then the default `DynamoDBMapperConfig.ConsistentReads.EVENTUAL` is used.
- A `DynamoDBMapperConfig.TableNameOverride` object - Instructs `DynamoDBMapper` to ignore the table name specified by a class's `DynamoDBTable` annotation and use a different one you supply. This is useful when partitioning your data into multiple tables at runtime.

You can override the default configuration object for `DynamoDBMapper` per operation, as needed.

Supported Data Types

This section describes the supported primitive Java data types, collections, and arbitrary data types.

Amazon 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`

The Amazon 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 Amazon DynamoDB types.

Java type	Amazon DynamoDB type
All number types	N (number type)
Strings	S (string type)
boolean	N (number type), 0 or 1.
Date	S (string type). The Date values are stored as ISO-8601 formatted strings.
Set collection types	SS (string set) type and NS (number set) type

In addition, Amazon DynamoDB supports arbitrary data types. For example, you can define your own complex types on the client. You use the `DynamoDBMarshaller` class and the `@DynamoDBMarhsalling` annotation type for the complex type to describe the mapping (Mapping Arbitrary Data with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model (p. 275)).

Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model

The optimistic locking support in the Object Persistence model ensures that the client-side item that you are updating (or deleting) is the same as the item on the server-side. The version mismatch can happen if after you retrieve an item some other transaction updates that item on the server making your copy a stale version. Without optimistic locking, if you send an update request using your stale version, updates made by another request might be lost.

The Object Persistence model provides the `@DynamoDBVersionAttribute` annotation type that you can use to enable optimistic locking. To enable locking, you specify one class property to store version numbers and mark it using this annotation type. When you save an object, the corresponding item on the server will have an attribute that stores the version number. The `DynamoDBMapper` assigns a version number when you first save the object and increments the version number 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.

For example, the following Java code snippet defines a `CatalogItem` class that has several properties. It also includes a property, `Version`, that is tagged with the `@DynamoDBVersionAttribute` annotation type.

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private List<String> bookAuthors;
    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 List<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(List<String> bookAuthors) { this.bookAuthors =
bookAuthors; }

    @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 the typical CRUD operations:

- `save` operation - 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` context increments the version number. You don't need to set the version number.
- `delete` operation -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 Amazon DynamoDB.

Disabling Optimistic Locking

You can configure the `DynamoDBMapper` to disable optimistic locking by specifying a `DynamoDBMapperConfig.SaveBehavior` enumeration value when using the `save` method. You can do this by creating a `DynamoDBMapperConfig` instance that skips version checking and use this instance for all your requests. For information about specifying optional `DynamoDBMapper` parameters, see Specifying Optional Configuration Information to the DynamoDBMapper (p. 342). 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 = context.load(CatalogItem.class, 101);
item.setSomeProp("value");
...
// Save the item.
mapper.save(item,
    new DynamoDBMapperConfig(
        DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

Mapping Arbitrary Data with Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model

In addition to the supported Java types (see Supported Data Types (p. 272)), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. To map these types, you must provide an implementation that converts your complex type to 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 Amazon 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 Amazon 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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.io.IOException;

import java.util.Arrays;

import java.util.HashSet;

import java.util.Set;



import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBAttribute;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBHashKey;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMapper;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMarshaller;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMarshalling;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBTable;
```

```
public class ObjectPersistenceMappingExample {

    static AmazonDynamoDBClient client;

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

        AWSCredentials credentials = new PropertiesCredentials(
              ObjectPersistenceMappingExample.class.getResourceAsStream("AwsCre
dentials.properties"));

        client = new AmazonDynamoDBClient(credentials);

        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", "Au
thor2")));
        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 in Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model

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 Amazon 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 Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.io.IOException;

import java.util.Arrays;

import java.util.HashSet;

import java.util.Set;


import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBAttribute;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBHashKey;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMapper;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMapperConfig;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBTable;


public class ObjectPersistenceCRUDExample {


    static AmazonDynamoDBClient client;


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

        AWSCredentials credentials = new PropertiesCredentials(

            ObjectPersistenceCRUDExample.class.getResourceAsStream("AwsCreden
tials.properties"));

        client = new AmazonDynamoDBClient(credentials);

        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.get
Id(), config);

        if (deletedItem == null) {

            System.out.println("Done - Sample item is deleted.");

        }

    }

}
```

Example: Batch Write Operation Using the AWS SDK for Java Object Persistence Model

The following Java code example declares Book, Forum, Thread, and Reply classes and maps them to the Amazon 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 in Amazon DynamoDB (p. 445). For step-by-step instructions to test the following sample, see Using the AWS SDK for Java with Amazon DynamoDB (p. 259).

```
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.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBAttribute;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBHashKey;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMapper;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMapperConfig;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBRangeKey;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBTable;


public class ObjectPersistenceBatchWriteExample {


    static AmazonDynamoDBClient client;

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


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

        try {

            AWSCredentials credentials = new PropertiesCredentials(

                    ObjectPersistenceBatchWriteExample.class.getResourceAs
Stream("AwsCredentials.properties"));


            client = new AmazonDynamoDBClient(credentials);

            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 = 1;
        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 = 1;
        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 = 0;
        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(DynamoDBMapper
Config.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 int 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 int getInPublication() { return inPublication; }

        public void setInPublication(int 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 in Amazon DynamoDB Using the AWS SDK for Java Object Persistence Model

The Java example in this section defines the following classes and maps them to the tables in Amazon DynamoDB. For more information about creating sample tables, see Example Tables and Data in Amazon DynamoDB (p. 445).

- `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 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 a specified value.

  For performance reason, 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 the one of the book price is set to less than 0. This examples scan the ProductCategory table to find book items (ProductCategory is book) and price is less than 0.

Note

This code example assumes that you have already loaded data into Amazon DynamoDB for your account by following the instructions in the Getting Started with Amazon DynamoDB (p. 11) 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 Low-Level API (p. 450) topic.

For step-by-step instructions to run the following example, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.List;

import java.util.Set;



import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBAttribute;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBHashKey;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBMapper;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBQueryExpression;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBRangeKey;

import com.amazonaws.services.dynamodb.datamodeling.DynamoDBScanExpression;
```

```
import com.amazonaws.services.dynamodb.datamodeling.DynamoDBTable;

import com.amazonaws.services.dynamodb.model.AttributeValue;

import com.amazonaws.services.dynamodb.model.ComparisonOperator;

import com.amazonaws.services.dynamodb.model.Condition;


public class ObjectPersistenceQueryScanExample {


    static AmazonDynamoDBClient client;

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


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

        try {

            AWSCredentials credentials = new PropertiesCredentials(

                    ObjectPersistenceQueryScanExample.class.getResourceAs
Stream("AwsCredentials.properties"));


            client = new AmazonDynamoDBClient(credentials);

            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.

            FindProductsPricedLessThanSpecifiedValue(mapper, "ProductCatalog",
 "20");

            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);

        String twoWeeksAgoStr = dateFormatter.format(twoWeeksAgo);



        Condition rangeKeyCondition = new Condition()

                .withComparisonOperator(ComparisonOperator.GT.toString())

                .withAttributeValueList(new AttributeValue().withS(twoWeek
sAgoStr.toString()));


        DynamoDBQueryExpression queryExpression = new DynamoDBQueryExpression(

                new AttributeValue().withS(hashKey));


        queryExpression.setRangeKeyCondition(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.

        String startDate = dateFormatter.format(startDateMilli);

        String endDate = dateFormatter.format(endDateMilli);


        Condition rangeKeyCondition2 = new Condition()

        .withComparisonOperator(ComparisonOperator.BETWEEN.toString())

        .withAttributeValueList(new AttributeValue().withS(startDate),

                                new AttributeValue().withS(endDate));


       DynamoDBQueryExpression queryExpression2 = new DynamoDBQueryExpression(

                new AttributeValue().withS(hashKey));


        queryExpression2.setRangeKeyCondition(rangeKeyCondition2);


       List<Reply> betweenReplies = mapper.query(Reply.class, queryExpression2);


        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 FindProductsPricedLessThanSpecifiedValue(
            DynamoDBMapper mapper,
            String tableName,
            String value) throws Exception {


        System.out.println("FindProductsPricedLessThanSpecifiedValue: Scan
ProductCatalog.");


        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);
        }
    }


    @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 int 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 int getInPublication() { return inPublication; }
        public void setInPublication(int 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; }

        @DynamoDBRangeKey(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;}

    }
}
```

Using the AWS SDK for .NET with Amazon DynamoDB

Topics

AWS SDK for .NET provides the following APIs to work with Amazon DynamoDB. All the API is 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 Amazon DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

API	Comment
Low-level API	It is the protocol level API that maps closely to the Amazon 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. For more information about the Amazon DynamoDB, see API Reference for Amazon DynamoDB (p. 371). The API is available in the `Amazon.DynamoDB.DataModel` namespace. The following sections describe the low-level API in various AWS SDKs and also provide working samples: • Working with Tables in Amazon DynamoDB (p. 64) • Working with Items in Amazon DynamoDB (p. 102) • Query and Scan in Amazon DynamoDB (p. 182)

API	Comment
Helper API	Provides wrapper classes around the low-level API to further simplify your programming task. The `Table` and `Document` are the key wrapper classes. You can use the helper API 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. The API is available in the `Amazon.DynamoDB.DocumentModel` namespace. For more information about the helper API and working samples, see Using the .NET Helper Classes in Amazon DynamoDB (p. 307).
Object Persistence Model API	The object persistence model API enables you to map your client-side classes to the Amazon DynamoDB tables. Each object instance then maps to an item in the corresponding tables. The DynamoDBContext 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. 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. The API is available in the `Amazon.DynamoDB.DataModel` namespace. For more information about the helper API and working samples, see Using the AWS SDK for .NET Object Persistence Model with Amazon DynamoDB (p. 335).

Running .NET Examples for Amazon DynamoDB

General Process of Creating .NET Code Examples (Using Visual Studio)

1	Create a new Visual Studio project using the *AWS Empty Project* template. You get this template if you installed the AWS Toolkit for Visual Studio. For more information, see Step 1: Before You Begin with Amazon DynamoDB (p. 11). The **AWS Access Credentials** dialog box opens.

2	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 security credentials.
	Visual Studio adds the AWS credentials associated with the toolkit account that you select to the app.config file as shown in the following example:
	`<?xml version="1.0"?>`
	`<configuration>`
	`  <appSettings>`
	`    <add key="AWSAccessKey" value="*** Access Key Id ***"/>`
	`    <add key="AWSSecretKey" value="*** Secret Access Key ***"/>`
	`  </appSettings>`
	`</configuration>`
	The code examples use the default client constructors that read your AWS credentials stored the app.config file.

3	Note that the *AWS Empty Project* template includes the required `AWSSDK` reference.
4	Replace the code in the project file, `Program.cs`, with the code in the section you are reading.
5	Run the code.

Setting the Endpoint

By default, AWS SDK for .NET sets endpoint to `https://dynamodb.us-east-1.amazonaws.com`. You can also set the endpoint explicitly as shown in the following C# code snippet.

```
private static void CreateClient()
{
  AmazonDynamoDBConfig config = new AmazonDynamoDBConfig();
  config.ServiceURL = "http://dynamodb.us-east-1.amazonaws.com";
  client = new AmazonDynamoDBClient(config);
}
```

For a current list of supported regions and endpoints, see Regions and Endpoints.

Using the .NET Helper Classes in Amazon DynamoDB

Topics

The AWS SDK for .NET provides helper classes that wrap some of the low-level API (see Working with Items Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB (p. 133)) functionality to further simplify your coding. Among them, the `Table` and `Document` classes are the primary helper classes. 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 Amazon DynamoDB Data Model (p. 3).

The preceding helper classes are available in the `Amazon.DynamoDB.DocumentModel` namespace.

Operations Not Supported by the Helper Classes

You cannot use these helper classes to create, update, and delete tables. The helper classes support most common data operations.

Working with Items in Amazon DynamoDB Using the AWS SDK for .NET Helper Classes

Topics

To perform data operations using the helper classes, 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 Amazon 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 Amazon 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, GetItemOpera
tionConfig 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. That is, 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 Helper Classes.

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");
// Upload a book item.
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" };

table.PutItem(book);
```

In the preceding example, the `Document` instance creates an item that has Id, Title, ISBN and Authors attributes. The Authors attribute is a multi-valued attribute.

Specifying Optional Parameters

You can configure optional parameter for the `PutItem` operation by adding the `PutItemOperationConfig` parameter. For a complete list of optional parameters, see PutItem (p. 413).

The following C# code snippet puts an item in the ProductCatalog table. It specifies the following optional parameter:

- The `Expected` parameter to make this a conditional put request. The example provides another `Document` instance that specifies an ISBN attribute that has a specific value that you expect 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" };

// Create another document for the optional conditional put operation.
Document expectedDocument = new Document();
expectedDocument["ISBN"] = "55-55-55-55";

PutItemOperationConfig config = new PutItemOperationConfig()
{
  // Optional parameter.
  Expected = expectedDocument
};

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 the eventually consistent read (see Data Read and Consistency Considerations (p. 7)) 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 (p. 409). 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 "Id" and "Title".
- 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. 7).

```
Table table = Table.LoadTable(client, "ProductCatalog");

GetItemOperationConfig config = new GetItemOperationConfig()
```

```
{
  AttributesToGet = new List<string>() { "Id", "Title" },
  ConsistentRead = true
};
Document document2 = table.GetItem(101, config);
```

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 (p. 403). The following C# code snippet specifies the two following optional parameters:

- The `Expected` 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.

Document expected = new Document();
expected["ISBN"] = "11-11-11-11";

// Specify optional parameters for Delete operation.
DeleteItemOperationConfig config = new DeleteItemOperationConfig
{
     Expected = expected,
     ReturnValues = ReturnValues.AllOldAttributes // This is the only supported
 value when using helper API.
};

// 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 th `Add` action (see UpdateItem (p. 433)) supported by the underlying API.

Note

The `PutItem` operation (Putting an Item - Table.PutItem Method (p. 309)) can also can perform an update. If you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. Note that, if there are attributes in the existing item and those attributes are not specified on the `Document` that is being put, the PutItem operation deletes those attributes. However, 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 Helper Classes.

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 (p. 433).

The following C# code snippet updates a book item price to 25. It specifies the two following optional parameters:

- The `Expected` parameter that includes the `Document` instance 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;

Document expectedDocument = new Document();
expectedDocument["Price"] = 20;

UpdateOperationConfig config = new UpdateOperationConfig()
{
  Expected = expectedDocument,
  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 helper 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 helper API, you can specify any number of operations in a batch. However, note that Amazon DynamoDB limits the number of operations in a batch and the total size of the batch in a batch operation. For more information on the specific limits, see BatchWriteItem (p. 389). If the helper 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 helper 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 Helper API";
book1["Price"] = 10;
book1["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };

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 Helper API for Amazon DynamoDB (p. 320).

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 helper API.

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"] = "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);

superBatch.Execute();
```

Example: Put, Get, Update, and Delete an Item in Amazon DynamoDB Using the AWS SDK for .NET Helper Classes

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 with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using System.Linq;

using Amazon.DynamoDB.DocumentModel;

using Amazon.Runtime;



namespace Amazon.DynamoDB.Documentation

{

  class Program

  {
```

```
private static AmazonDynamoDBClient client;
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
  {
    client = new AmazonDynamoDBClient();

    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"] = 1; // True.

      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, created another document with expected
price value.

      Document expectedDocument = new Document();

      expectedDocument["Price"] = 19.99;

      // Optional parameters.

      UpdateItemOperationConfig config = new UpdateItemOperationConfig

      {

        Expected = expectedDocument,

        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;
        else if (value is PrimitiveList)
          stringValue = string.Join(",", (from primitive
                                          in value.AsPrimitiveList().Entries

                                         select primitive.Value).ToArray());
        Console.WriteLine("{0} - {1}", attribute, stringValue);
      }
    }
  }
}
```

Example: Batch Operations Using AWS SDK for .NET Helper API for Amazon DynamoDB

Topics

Example: Batch Write Using AWS SDK for .NET Helper Classes

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 in the ProductCatalog table.
- To illustrate a multi-table batch write, it adds an item in both the Forum and Thread tables and deletes and item from the Thread table.

If you followed the Getting Started 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 Low-Level API (p. 468). For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.DocumentModel;

using Amazon.Runtime;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;

    static void Main(string[] args)

    {

      try

      {

        client = new AmazonDynamoDBClient();


        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 API";

      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";

      batchWrite.AddDocumentToPut(book1);

      // Specify delete item using overload that takes PK.

      batchWrite.AddKeyToDelete(12345);

      Console.WriteLine("Performing batch write in SingleTableBatchWrite()");

      batchWrite.Execute();

    }


    private static void MultiTableBatchWrite()

    {

      // 1. Specify item to add in the Forum table.
```

```
Table forum = Table.LoadTable(client, "Forum");
var forumBatchWrite = forum.CreateBatchWrite();

var forum1 = new Document();
forum1["Name"] = "Test BatchWrite Forum";
forum1["Threads"] = 0;
forumBatchWrite.AddDocumentToPut(forum1);



// 2a. Specify item to add in the Thread table.
Table thread = Table.LoadTable(client, "Thread");
var threadBatchWrite = thread.CreateBatchWrite();

var thread1 = new Document();
thread1["ForumName"] = "S3 forum";
thread1["Subject"] = "My sample question";
thread1["Message"] = "Message text";
thread1["KeywordTags"] = new List<string> { "S3", "Bucket" };
threadBatchWrite.AddDocumentToPut(thread1);

// 2b. Specify item to delete from the Thread table.
threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

// 3. Create multi-table batch.
var superBatch = new MultiTableDocumentBatchWrite();
superBatch.AddBatch(forumBatchWrite);
superBatch.AddBatch(threadBatchWrite);
Console.WriteLine("Performing batch write in MultiTableBatchWrite()");
superBatch.Execute();
```

```
    }

  }

}
```

Querying Tables in Amazon DynamoDB Using the AWS SDK for .NET Helper Classes

Topics

Table.Query Helper 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. 7).

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, consistent read for the latest data, page size, and the number of items returned per page. For a complete list of parameters, see Query (p. 417). 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 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 helper 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 consistent read and a list of attributes to retrieve.
  - Find forum thread replies posted in a duration.
    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.DynamoDB.DocumentModel;

using Amazon.Runtime;

using Amazon.SecurityToken;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      try

      {

        client = new AmazonDynamoDBClient();


        // Query examples.

        Table replyTable = Table.LoadTable(client, "Reply");

        string forumName = "Amazon DynamoDB";

        string threadSubject = "DynamoDB Thread 2";

        FindRepliesInLast15Days(replyTable, forumName, threadSubject);

       FindRepliesInLast15DaysWithConfig(replyTable, forumName, threadSubject);


       FindRepliesPostedWithinTimePeriod(replyTable, forumName, threadSubject);
```

```
        // Get Example.
        Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");

        int productId = 101;
        GetProduct(productCatalogTable, productId);


        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
      }
      catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
      catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
      catch (Exception e) { Console.WriteLine(e.Message); }
    }


    private static void GetProduct(Table tableName, int productId)
    {
      Console.WriteLine("*** Executing GetProduct() ***");
      Document productDocument = tableName.GetItem(productId);
      PrintDocument(productDocument);
    }


    private static void FindRepliesInLast15Days(Table table, string fourmName,
 string threadSubject)
    {
      string hashAttribute = fourmName + "#" + threadSubject;


      DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
      RangeFilter filter = new RangeFilter(QueryOperator.GreaterThan, twoWeek
sAgoDate);
```

```
      // Use Query overloads that takes the minimum required query parameters.

      Search search = table.Query(hashAttribute, 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));


      QueryOperationConfig config = new QueryOperationConfig()

      {

        HashKey = forumName + "#" + threadSubject,

        Limit = 2, // 2 items/page.

        AttributesToGet = new List<string> { "Message",

                                             "ReplyDateTime",

                                             "PostedBy" },

        ConsistentRead = true,

        Filter = new RangeFilter(QueryOperator.Between, startDate, endDate)
```

```
      };


      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);
      // You are specifying optional parameters so use QueryOperationConfig.
      QueryOperationConfig config = new QueryOperationConfig()
      {
        HashKey = forumName + "#" + threadName,
        Filter = new RangeFilter(QueryOperator.GreaterThan, twoWeeksAgoDate),
        // Optional parameters.
        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;
        else if (value is PrimitiveList)
          stringValue = string.Join(",", (from primitive
                                           in value.AsPrimitiveList().Entries
```

```
                                            select primitive.Value).ToArray());

        Console.WriteLine("{0} - {1}", attribute, stringValue);

      }

    }

  }

}
```

Table.Scan Helper 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 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 helper 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.DynamoDB.DocumentModel;



namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;



    static void Main(string[] args)

    {
```

```
      client = new AmazonDynamoDBClient();


      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 product
CatalogTable)

    {

      // 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,

        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;

        else if (value is PrimitiveList)

          stringValue = string.Join(",", (from primitive

                                          in value.AsPrimitiveList().Entries

                                         select primitive.Value).ToArray());

        Console.WriteLine("{0} - {1}", attribute, stringValue);

      }

    }

  }

}
```

Using the AWS SDK for .NET Object Persistence Model with Amazon DynamoDB

Topics

The AWS SDK for .NET provides an object persistence model that enables you to map your client-side classes to the Amazon DynamoDB tables. Each object instance then maps to an item in the corresponding tables. To save your client-side objects to the tables the object persistence model provides the `DynamoDBContext` class, an entry point to Amazon DynamoDB. This class provides you a connection to Amazon 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 for Amazon DynamoDB (p. 82).

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 Amazon DynamoDB types. To learn about mapping arbitrary types, see Mapping Arbitrary Data with Amazon DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 346).

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 Amazon DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 344).

Amazon DynamoDB Attributes

The following table lists the attributes the object persistence model offers so you can map your classes and properties to Amazon DynamoDB tables and attributes.

Note

In the following table, only `DynamoDBTable` and `DynamoDBHashKey` are required tags.

<table>
<tr><th>Declarative Tag (attribute)</th><th>Description</th></tr>
<tr><td><code>DynamoDBHashKey</code></td><td>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 ProductCatalog 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></td></tr>
<tr><td><code>DynamoDBIgnore</code></td><td>Indicates <code>DynamoDBContext</code> 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.</td></tr>
</table>

<table>
<tr><th>Declarative Tag (attribute)</th><th>Description</th></tr>
<tr><td><code>DynamoDBProperty</code></td><td>

Maps a class property to a table attribute. If the class property maps to the same name table attribute, then you don't need to specify this attribute. However, if the names are not the same, you can use this tag to provide the mapping. In the following C# statement the `DynamoDBProperty` maps the `BookAuthors` property to the Authors attribute in the table.

```
[DynamoDBProperty("Authors")]
public List<string> BookAuthors { get; set; }
```

`DynamoDBContext` uses this mapping information to create the Authors attribute when saving object data to the corresponding table.

</td></tr>
<tr><td><code>DynamoDBRangeKey</code></td><td>

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.

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.

For more information about sample tables, see Example Tables and Data in Amazon DynamoDB (p. 445).

```
[DynamoDBTable("Reply")]
public class Reply {
   [DynamoDBHashKey]
   public int ThreadId { get; set; }
   [DynamoDBRangeKey]
   public string Replenishment { get; set; }
   // Additional properties go here.
}
```

</td></tr>
</table>

<table>
<tr><th>Declarative Tag (attribute)</th><th>Description</th></tr>
<tr><td><code>DynamoDBTable</code></td><td>
Identifies the target table in Amazon DynamoDB to which the class maps. For example, the following C# code example maps the <code>Developer</code> class to the People table in Amazon DynamoDB.
<pre><code>[DynamoDBTable("People")]
public class Developer { ...}</code></pre>
This attribute can be inherited or overridden.

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 People table. That is, both the <code>Developer</code> and <code>Lead</code> objects are stored in the People 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 (Managers).
<pre><code>[DynamoDBTable("Managers")]
public class Manager : Developer { ...}</code></pre>

You can add the optional parameter, <code>LowerCamelCaseProperties</code>, to request Amazon 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.
<pre><code>[DynamoDBTable("People", LowerCamelCaseProperties=true)]
public class Developer {
   string DeveloperName;
   ...}</code></pre>
When saving instances of the <code>Developer</code> class, <code>DynamoDBContext</code> saves the <code>DeveloperName</code> property as the developerName.
</td></tr>
<tr><td><code>DynamoDBVersion</code></td><td>Identifies a class property for storing the item version number. To more information about versioning, see Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 344).</td></tr>
</table>

DynamoDBContext Class

The DynamoDBContext class is the entry point to the Amazon DynamoDB database. It provides a connection to Amazon 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
`Create BatchWrite`	Creates a `BatchWrite` object you can use to put or delete several items from a table. This method maps to the the Amazon DynamoDB `BatchWriteItem` operation (see BatchWriteItem (p. 389)). For more information, see Batch Write: Putting and Deleting Multiple Items (p. 350).
`Create BatchGet`	Returns multiple items from one or more tables. It executes the Amazon DynamoDB `BatchGetItem` operation (see BatchGetItem (p. 385). For more information, see Batch Get: Getting Multiple Items (p. 352).
`Delete`	Deletes an item from the table. The method requires the primary key of the item you want to delete. You can provide either the primary key value or a client-side object containing a primary key value as a parameter to this method. • If you specify a client-side object as a parameter and you have enabled optimistic locking, the delete succeeds only if the client-side and the server-side versions of the object match. • If you specify only the primary key value as a parameter, the delete succeeds regardless of whether you have enabled optimistic locking or not.
`Load`	Retrieves an item from a table. The methods requires only the primary key of the item you want to retrieve. By default, Amazon DynamoDB returns the item with values that are eventually consistent. For information on the eventual consistency model, see Data Read and Consistency Considerations (p. 7).

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td><code>Query</code></td><td>
Queries a table based on query parameters you provide.
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.
Suppose you have a client-side Reply class mapped to the Reply table in Amazon 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 in Amazon DynamoDB (p. 445).
<pre><code>DynamoDBContext context = new DynamoDBContext(client);

string replyId = "Amazon 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);</code></pre>
The query returns a collection of <code>Reply</code> objects. The <code>Query</code> method returns a "lazy-loaded" IEnumerable collection. That is, initially it returns only one page of results. It makes a service call for the next page when needed.
Note
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.
</td></tr>
<tr><td><code>Save</code></td><td>
Saves the specified object to the table. If the primary key specified in the input object does not exist in the table, the method adds a new item to the table. If primary key exists, the method updates the existing item.
If you have optimistic locking configured, the update succeeds only if the client and the server side versions of the item match. For more information, see Optimistic Locking Using Version Number with Amazon DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 344).
</td></tr>
</table>

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td><code>Scan</code></td><td>
Performs an entire table scan.
You can filter scan result by specifying a scan condition. The condition can be evaluated on any attributes in the table. Suppose you have a client-side class <code>Book</code> mapped to the ProductCatalog table in Amazon DynamoDB. The following C# snippet scans the table and returns only the book items priced less than 0.
<pre><code>IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(
                    new ScanCondition("Price", ScanOperat
or.LessThan, price),
                   new ScanCondition("ProductCategory", ScanOper
ator.Equal, "Book")
      );</code></pre>
The <code>Scan</code> method returns the "lazy-loaded" IEnumerable collection. That is, initially it returns only one page of results. It makes a service call for the next page when needed.
For performance reasons you should query your tables and avoid a table scan.
</td></tr>
<tr><td><code>ToDocument</code></td><td>
Returns an instance of the <code>Document</code> helper class from your class instance.
This is helpful if you want to use the helper classes along with the object persistence model to perform any data operations. For more information about the helper classes provided by the AWS SDK for .NET, see Using the .NET Helper Classes in Amazon DynamoDB (p. 307).
Suppose you have a client-side class mapped to the sample ProductCatalog table. You can then use a DynamoDBContext to get an item, as a <code>Document</code> object, from the <code>ProductCatalog</code> table as shown in the following C# code snippet.
<pre><code>DynamoDBContext context = new DynamoDBContext(client);

Forum forum101 = context.Load<Forum>(101); // Retrieve a forum by
 primary key.
Document d = context.ToDocument<Forum>(forum101);</code></pre>
</td></tr>
</table>

Specifying Optional Parameters to the DynamoDBContext

When using the object persistence model, you can specify the following optional parameters to 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 data consistency, see Amazon DynamoDB Data Model (p. 3).
- **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 Amazon DynamoDB Using the AWS SDK for .NET Object Persistence Model (p. 344).

The following C# snippet creates a new DynamoDBContext by specifying both 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 Amazon DynamoDB types.

.NET primitive type	Amazon DynamoDB type
All number types	N (number type)
All string types	S (string type)
bool	N (number type), 0 represents false and 1 represents true.
Collection types	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 Amazon DynamoDB types.

Optimistic Locking Using Version Number with Amazon 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 Amazon 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 to the DynamoDBContext (p. 342).

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 Amazon DynamoDB Using the AWS SDK for .NET Object Persistence Model

In addition to the supported .NET types (see Supported Data Types (p. 343)), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. The object persistence model supports storing data of arbitrary types as long as you provide the converter to convert data from the arbitrary type to the Amazon 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 Amazon DynamoDB types and during query and scan any data comparisons made are against the data stored in Amazon 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 Amazon 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 with Amazon DynamoDB (p. 305) in the *Amazon DynamoDB Developer Guide*.

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.DataModel;

using Amazon.DynamoDB.DocumentModel;

using Amazon.Runtime;

using Amazon.SecurityToken;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      try
```

```
    {
      client = new AmazonDynamoDBClient();
      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 || string.IsNullOrEmpty(primitive.Value))
      throw new ArgumentOutOfRangeException();
```

```
     string[] data = primitive.Value.Split(new string[] { " x " }, StringSplit
Options.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 Amazon DynamoDB limits the number of operations in a batch and the total size of the batch in a batch operation. For more information on the specific limits, see BatchWriteItem (p. 389). 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 Amazon 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 = "S3 forum",
  Subject = "My sample question",
  KeywordTags = new List<string> { "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. 356).

Note

Amazon DynamoDB batch API limits the number of writes in batch and also limits the size of the batch. For more information, see BatchWriteItem (p. 389). 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 Amazon 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(devBatch.Results.Count);
Book book1 = bookBatch.Results[0];
Book book2 = bookBatch.Results[1];
Book book3 = bookBatch.Results[2];
```

To retrieve objects from multiple tables, do the following:

- For each type, create an instance of the `CreateBatchGet` type and provide the primary key values you want to retrieve from each table.
- Create an instance of the `MultiTableBatchGet` class using one of the following methods:
  - Execute the `Combine` method on one of the `BatchGet` objects you created in the preceding step.
  - Create an instance of the `MultiBatchGet` type by providing a list of `BatchGet` objects.
  - Execute the `CreateMultiTableBatchGet` method of `DynamoDBContext` and pass in your list of `BatchGet` objects.
- Call the `Execute` method of `MultiTableBatchGet` which returns the typed results in the individual `BatchGet` objects.

The following C# code snippet retrieves multiple items from the Order and OrderDetail tables using the CreateBatchGet method.

```
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 in Amazon DynamoDB 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 Amazon 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 in Amazon DynamoDB (p. 445). For step-by-step instructions to test the following sample, go to Using the AWS SDK for .NET with Amazon DynamoDB (p. 305) in the *Amazon DynamoDB Developer Guide*.

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.DataModel;

using Amazon.Runtime;


namespace Amazon.DynamoDB.Documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      try

      {

        client = new AmazonDynamoDBClient();

        DynamoDBContext context = new DynamoDBContext(client);

        TestCRUDOperations(context);

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

        Console.ReadLine();

      }

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

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

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

    }


    private static void TestCRUDOperations(DynamoDBContext context)
```

```
    {
      int bookID = 1001; // Some unique value.
      Book myBook = new Book
      {
        Id = bookID,
        Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
        ISBN = "111-1111111001",
        BookAuthors = new List<string> { "Author 1", "Author 2" },
      };

      // Save the book.
      context.Save(myBook);
      // Retrieve the book.
      Book bookRetrieved = context.Load<Book>(bookID);

      // Update few properties.
      bookRetrieved.ISBN = "222-2222221001";
      bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author x"
}; // Replace existing authors list with this.
      context.Save(bookRetrieved);

      // Retrieve the updated book. This time add the optional ConsistentRead
parameter using DynamoDBContextConfig object.
      Book updatedBook = context.Load<Book>(bookID, new DynamoDBContextConfig
{ ConsistentRead = true });

      // Delete the book.
      context.Delete<Book>(bookID);
      // Try to retrieve deleted book. It should return null.
      Book deletedBook = context.Load<Book>(bookID, new DynamoDBContextConfig
{ ConsistentRead = true });
```

```
      if (deletedBook == null)
        Console.WriteLine("Book is deleted");

    }
  }

  [DynamoDBTable("ProductCatalog")]
  public class Book
  {
    [DynamoDBHashKey]   // 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 Amazon 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 in Amazon DynamoDB (p. 445). For step-by-step instructions to test the following sample, see Using the AWS SDK for .NET with Amazon DynamoDB (p. 305).

```
using System;
```

```
using System.Collections.Generic;
using Amazon.DynamoDB.DataModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace Amazon.DynamoDB.Documentation
{
  class ORMTest
  {
    static void Main(string[] args)
    {
      try
      {
        var client = CreateClient();
        DynamoDBContext context = new DynamoDBContext(client);
        SingleTableBatchWrite(context);
        MultiTableBatchWrite(context);
      }
      catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
      catch (Exception e) { Console.WriteLine(e.Message); }

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

    private static void SingleTableBatchWrite(DynamoDBContext context)
    {
      Book book1 = new Book
      {
```

```
        Id = 902,
        InPublication = true,
        ISBN = "902-11-11-1111",
        PageCount = "100",
        Price = 10,
        ProductCategory = "Book",
        Title = "My book3 in batch write"
      };
      Book book2 = new Book
      {
        Id = 903,
        InPublication = true,
        ISBN = "903-11-11-1111",
        PageCount = "200",
        Price = 10,
        ProductCategory = "Book",
        Title = "My book4 in batch write"
      };

      var bookBatch = context.CreateBatchWrite<Book>();
      bookBatch.AddPutItems(new List<Book> { book1, book2 });

      Console.WriteLine("Performing batch write in SingleTableBatchWrite().");
      bookBatch.Execute();
    }

    private static void MultiTableBatchWrite(DynamoDBContext context)
    {
      // 1. New Forum item.
```

```
    Forum newForum = new Forum
    {
      Name = "Test BatchWrite Forum",
      Threads = 0
    };
    var forumBatch = context.CreateBatchWrite<Forum>();
    forumBatch.AddPutItem(newForum);

    // 2. New Thread item.
    Thread newThread = new Thread
    {
      ForumName = "S3 forum",
      Subject = "My sample question",
      KeywordTags = new List<string> { "S3", "Bucket" },
      Message = "Message text"
    };

    DynamoDBOperationConfig config = new DynamoDBOperationConfig();
    config.SkipVersionCheck = true;
    var threadBatch = context.CreateBatchWrite<Thread>(config);
    threadBatch.AddPutItem(newThread);
    threadBatch.AddDeleteKey("some hash attr", "some range attr");

    var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);
    Console.WriteLine("Performing batch write in MultiTableBatchWrite().");
    superBatch.Execute();
  }

  private static AmazonDynamoDBClient CreateClient()
```

```
  {
    var userCredentials = new EnvironmentAWSCredentials();
    var stsClient = new AmazonSecurityTokenServiceClient(userCredentials);
    var sessionCredentials = new RefreshingSessionAWSCredentials(stsClient);

    var client = new AmazonDynamoDBClient(sessionCredentials);
    return client;
  }
}

[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 Amazon 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 Amazon DynamoDB. For more information about creating sample tables, see Example Tables and Data in Amazon DynamoDB (p. 445).

- `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 examples scan 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 with Amazon DynamoDB (p. 305).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB.DataModel;

using Amazon.DynamoDB.DocumentModel;

using Amazon.Runtime;



namespace Amazon.DynamoDB.Documentation

{

  class Test

  {

    private static AmazonDynamoDBClient client;
```

```
static void Main(string[] args)
{
  try
  {
    AmazonDynamoDBClient client = new AmazonDynamoDBClient();
    DynamoDBContext context = new DynamoDBContext(client);
    // Get item.
    GetBook(context, 101);

    // Sample forum and thread to test queries.
    string forumName = "Amazon DynamoDB";
    string threadSubject = "DynamoDB Thread 1";
    // Sample queries.
    FindRepliesInLast15Days(context, forumName, threadSubject);
    FindRepliesPostedWithinTimePeriod(context, forumName, threadSubject);

    // Scan table.
    FindProductsPricedLessThanZero(context);
    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
  }
  catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
  catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
  catch (Exception e) { Console.WriteLine(e.Message); }
}

private static void GetBook(DynamoDBContext context, int productId)
{
  Book bookItem = context.Load<Book>(productId);
```

```
      Console.WriteLine("\nGetBook: Printing result.....");

      Console.WriteLine("Title: {0} \n No.Of threads:{1} \n No. of messages:
{2}",

                        bookItem.Title, bookItem.ISBN, bookItem.PageCount);

    }


    private static void FindRepliesInLast15Days(DynamoDBContext context,

                                                string forumName,

                                                string threadSubject)

    {

      string replyId = forumName + "#" + threadSubject;

      DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);

      IEnumerable<Reply> latestReplies =

        context.Query<Reply>(replyId, QueryOperator.GreaterThan, twoWeeksAgoD
ate);

      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 con
text,

                                                          string forumName,

                                                         string threadSubject)


    {

      string forumId = forumName + "#" + threadSubject;

      Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: Printing res
ult.....");
```

```
      DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(30);

      DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(1);



      IEnumerable<Reply> repliesInAPeriod = context.Query<Reply>(forumId,

                                                                    QueryOperat
or.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 AWS SDK for PHP with Amazon DynamoDB

The AWS SDK for PHP currently has one level of API (called the "low-level" API) for Amazon DynamoDB that directly maps 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 upon your environment and how you want to run your application. You must read the instructions at AWS SDK for PHP and set up your environment, accordingly, before you can run the examples in this documentation.

The AWS SDK for PHP uses HTTPS, by default. Read the PHP API Reference documentation to learn how to override this setting, if necessary.

The PHP API Reference documentation is online at http://docs.amazonwebservices.com/AWSSDKforPHP/latest/.

Running PHP Examples for Amazon DynamoDB

General Process of Creating PHP Code Examples

1	Install the PHP SDK in your environment and verify your environment meets the minimum requirements according to the instructions in Getting Started with the AWS SDK for PHP.

<table>
<tr><td>2</td><td>Make sure you include a reference to the <code>sdk.class.php</code> file, and have edited your <code>config.inc.php</code> file, as instructed in Getting Started with the AWS SDK for PHP.</td></tr>
<tr><td>3</td><td>Copy the example code from the section you are reading to your project. Depending upon your environment and the instructions for the AWS SDK for PHP, you might need to add lines to the code example that reference the correct configuration and SDK files.
For example, to load a PHP example in a browser, add the following to the top of the PHP code and save it to a file with the <code>.php</code> extension in the Web application directory (such as <code>www</code> or <code>htdocs</code>):
<pre><?php
header('Content-Type: text/plain; charset=utf-8');

// If necessary, reference the sdk.class.php file. Otherwise, comment-out or delete the reference.
// This assumes the sdk.class.php file is in the same directory as this file
require_once dirname(__FILE__) . '/sdk.class.php';</pre></td></tr>
<tr><td>4</td><td>Test the example according to your setup.</td></tr>
</table>

Setting the Endpoint

By default, AWS SDK for PHP sets the endpoint to `https://dynamodb.us-east-1.amazonaws.com`. You can also set the endpoint explicitly as shown in the following PHP code snippet. Refer to the AWS SDK for PHP documentation at AWS SDK for PHP to learn more about setting the endpoint and protocol.

```
$dynamodb = new AmazonDynamoDB();
$dynamodb->set_hostname(Endpoint URL Here);
```

For a current list of supported regions and endpoints, see Regions and Endpoints.

API Reference for Amazon DynamoDB

Topics

JSON Data Format in Amazon DynamoDB

Amazon DynamoDB uses the JSON (JavaScript Object Notation) data format to send and receive formatted data. JSON presents data in a hierarchy so that both data values and data structure are conveyed simultaneously. Amazon DynamoDB uses the JSON (JavaScript Object Notation) data format to send and receive formatted data. 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":
    {"CreationDateTime":1.309988345372E9,
    "ItemCount":23,
    "KeySchema":
        {"HashKeyElement":
            {"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":
            {"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":
       {"LastIncreaseDateTime": 1.309988345384E9, "ReadCapacityUnits":10,"Write
CapacityUnits":10},
    "TableName":"users",
    "TableSizeBytes":949,
    "TableStatus":"ACTIVE"
```

```
    }
}
```

JSON Is for the Transport Protocol Only

Amazon DynamoDB uses JSON only as a transport protocol. You use JSON notation to send data, and Amazon DynamoDB responds with JSON notation, but the data is not being stored "on-disk" in the JSON data format.

Applications that use Amazon 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 Amazon 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, Amazon 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 *"01"*, *"2"*, and *"03"* in the proper sequence.

Making HTTP Requests to Amazon DynamoDB

Topics

If you don't use one of the AWS SDKs, you can perform Amazon 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.

Note

Amazon DynamoDB uses the AWS Security Token Service for session authorization. Before you can create a signed request to Amazon DynamoDB, you need to get temporary security credentials from the AWS Security Token Service. You use your temporary security credentials to make Amazon DynamoDB requests for as long as they are valid (up to 36 hours). For more information, see Requesting AWS Security Token Service Authentication for Amazon DynamoDB (p. 377).

HTTP Header Contents

Amazon DynamoDB requires the following information in the header of an HTTP request:

- *host* The Amazon DynamoDB endpoint. For more information about endpoints, see Accessing Amazon DynamoDB (p. 9).

- *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 date must be specified in one of the following three formats, as specified in the HTTP/1.1 RFC:
  - Sun, 06 Nov 1994 08:49:37 GMT (RFC 822, updated by RFC 1123)
  - Sunday, 06-Nov-94 08:49:37 GMT (RFC 850, obsoleted by RFC 1036)
  - Sun Nov 6 08:49:37 1994 (ANSI C asctime() format)
- *x-amzn-authorization* The signed request parameters in the format:

```
AWS3 AWSAccessKeyId=Temporary Access Key Value from AWS IAM Service,Al
gorithm=HmacSHA256, [,SignedHeaders=Header1;Header2;...]
Signature=S(StringToSign)
```

  AWS3 - An AWS implementation-specific tag that denotes the authentication version used to sign the request. (For Amazon DynamoDB, this value is always *AWS3*.)

  AWSAccessKeyId - The AWS Access Key ID provided by the AWS Security Token Service.

  Algorithm - The algorithm used to create the HMAC-SHA value of the string-to-sign, such as *HmacSHA256*.

  Signature - Base64(Algorithm(StringToSign, SigningKey)). For more information about the signature format, see Calculating the HMAC-SHA256 Signature for Amazon DynamoDB (p. 375)

  SignedHeaders - Optional. If present, contains a list of all the HTTP headers used in the canonicalized HttpHeaders calculation. A single semicolon character (;) (ASCII character 59) must be used as the delimiter for list values.
- *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_20111205.CreateTable*
- *x-amz-security-token* The security token value provided by the AWS Security Token Service. For more information, see Requesting AWS Security Token Service Authentication for Amazon DynamoDB (p. 377)
- *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-east-1.amazonaws.com
x-amz-date: Mon, 16 Jan 2012 17:49:52 GMT
x-amzn-authorization: AWS3 AWSAccessKeyId=TemporaryAccessKeyID,Algorithm=Hmac
SHA256,SignedHeaders=Host;x-Amz-Date;x-Amz-Target;x-amz-security-token,Signa
ture=*Encoded Signature*=Date: Mon, 31 Oct 2011 17:49:52 GMT
x-amz-target: DynamoDB_20111205.CreateTable
x-amz-security-token:*Token Value*
content-type: application/x-amz-json-1.0
content-length: 23
connection: Keep-Alive
user-agent: aws-sdk-java/1.2.10 Windows_7/6.1 Java_HotSpot(TM)_64-Bit_Serv
er_VM/20.2-b06
```

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 Amazon DynamoDB API. The Amazon DynamoDB JSON data schema defines the types of data and parameters (such as comparison operators and enumeration constants) available for each operation.

Note

Amazon DynamoDB uses JSON as a transport protocol, then it parses the data for storage. However, data is not stored natively in JSON format. For more information, see JSON Data Format in Amazon DynamoDB (p. 371).

Amazon DynamoDB does not serialize null values. If you are using a JSON parser set to serialize null values for requests, Amazon 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":
            [{"HashKeyElement":{"S":"Dave"}},
            {"HashKeyElement":{"S":"John"}},
            {"HashKeyElement":{"S":"Jane"}}],
        "AttributesToGet":
            ["score"]
        }
    }
}
```

Sample Amazon DynamoDB JSON Request and Response

The following examples show a request for the item in a table where the *HashKeyElement* is the name *"Bill & Ted's Excellent Adventure"* and the *RangeKeyElement* is the date *1989*. Then it shows the Amazon DynamoDB response, including all the attributes of that item.

HTTP POST Request:

```
POST / HTTP/1.1
Host: dynamodb.us-east-1.amazonaws.com
x-amz-date: Mon, 16 Jan 2012 17:50:52 GMT
x-amzn-authorization: AWS3 AWSAccessKeyId=TemporaryAccessKeyID,Algorithm=Hmac
SHA256,SignedHeaders=Host;x-amz-date;x-amz-target;x-amz-security-token,Signa
ture=*Signature Value*=
Date: Mon, 31 Oct 2011 17:49:52 GMT
x-amz-target: DynamoDB_20111205.GetItem
x-amz-security-token: *Token Value*
Content-Type: application/x-amz-json-1.0
```

```
Content-Length: 135
Connection: Keep-Alive
User-Agent: aws-sdk-java/1.2.10 Windows_7/6.1 Java_HotSpot(TM)_64-Bit_Serv
er_VM/20.2-b06

{"TableName":"my-table",
    "Keys":
        [{"HashKeyElement":{"S":"Bill & Ted's Excellent Adventure"},
        "RangeKeyElement":{"S":1989}}]
}
```

Amazon DynamoDB Response:

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
Content-Type: application/x-amz-json-1.0
Content-Length: 144
Date: Mon, 16 Jan 2012 17:49:52 GMT

{"Items":
    [{"date":{"S":1989},
    "fans":{"SS":["Keneau","Alexis","John"]},
    "name":{"S":"Bill & Ted's Excellent Adventure"},
    "rating":{"S":"****"}}]
}
```

Notice the protocol (*HTTP/1.1*) is followed by a status code (*200*). A code value of *200* indicates a successful operation. For information on error codes, see API Error Codes (p. 379).

Calculating the HMAC-SHA256 Signature for Amazon DynamoDB

Required Authentication Information

Every request to Amazon DynamoDB must be authenticated. The AWS SDKs automatically sign your requests and manage your AWS Security Token Service credentials as required for Amazon DynamoDB. If you want to write your own HTTP POST requests, however, you need to create an *x-amzn-authorization* value for the HTTP POST Header content as part of authenticating your request. For more information about formatting headers, see HTTP Header Contents (p. 372).

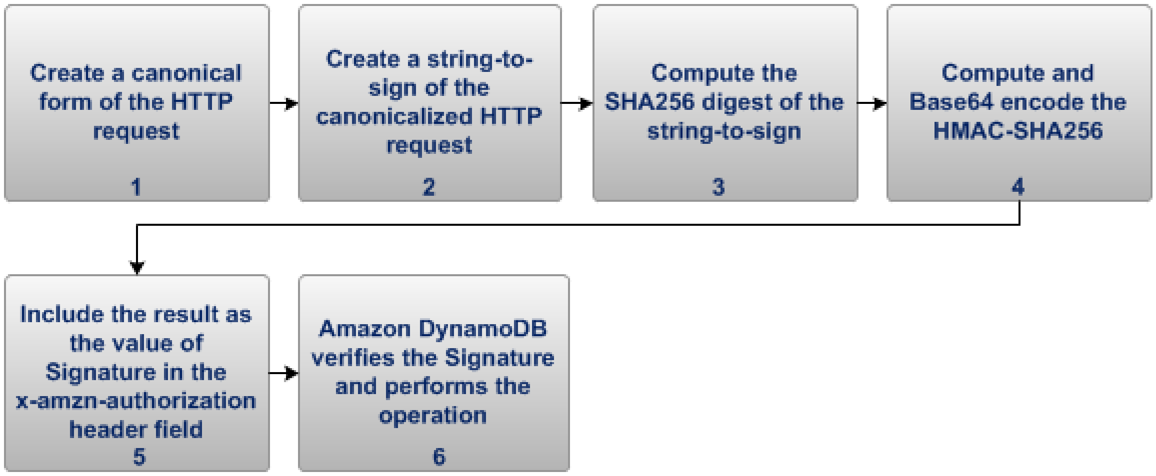
Signature Process

The following diagram shows the series of tasks required to create an HMAC-SHA256 (Hash-based Message Authentication Code-Secure Hash Algorithm) request signature. This process assumes that you have already received temporary security credentials from the AWS Security Token Service. For more information about the AWS Security Token Service, see What Is AWS Security Token Service Authentication? (p. 377).

Note

Use the SHA256 function for signing; it provides improved security over the SHA1 function. The value SHA256 must match the value for the *Algorithm* name provided in the HTTP header of the request to Amazon DynamoDB.

You perform the following tasks to sign and submit a request to Amazon DynamoDB.



Signing Process

1. Create a canonical form of the HTTP request headers. The canonical form of the HTTP header includes the following.
   - `host`, which specifies the endpoint to use.

```
host:dynamodb.us-east-1.amazonaws.com\n
```

   - Any header element starting with `x-amz-`. The date is required, and the following shows the format.

```
x-amz-date:Mon, 16 Jan 2012 17:49:52 GMT\n
```

   For more information about the included headers, see HTTP Header Contents (p. 372).

   a. For each header name-value pair, convert the header name to lower-case (not the header value).

   b. Build a map of header name to comma separated header values as prescribed by RFC 2616, section 4.2.

```
x-amz-example: value1
x-amz-example: value2  =>  x-amz-example:value1,value2
```

   c. For each header name-value pair, convert the name-value pair into a string in the format `headerName:headerValue`. Trim any whitespace from the beginning and end of both `headerName` and `headerValue`, with no space on each side of the colon.

```
x-amz-example1:value1,value2
x-amz-example2:value3
```

   d. Insert a new line (U+000A) after each converted string, including the last string.

   e. Sort the collection of converted strings by header name, alphabetically.

2. Create a *`string-to-sign`* value that includes the following.
   - Line 1: The HTTP method (*`POST`*), followed by a newline.
   - Line 2: The request URI (*`/`*), followed by a newline.

- Line 3: An empty string. Typically, a query string goes here, but Amazon DynamoDB doesn't use a query string. Follow with a newline.
- Line 4-n: The string representing the canonicalized request headers you computed in step 1, followed by a newline.
- The request body. Do not follow the request body with a newline.

3. Compute the SHA256 digest of the `string-to-sign` value. Use the same SHA method throughout the process.
4. Compute and Base64 encode the HMAC-SHA256 digest of the resulting value from the previous step using the temporary security credentials you received from the AWS Security Token Service API.

   Note

   Amazon DynamoDB expects an equals character (=) at the end of the Base64 encoded HMAC-SHA value. If your Base64 encoding routine doesn't append an equals character, manually append it.

5. Use the resulting value as the value for the *`Signature`* name in the *`x-amzn-authorization`* header field of the HTTP request to Amazon DynamoDB.
6. Amazon DynamoDB verifies the request and performs the specified operation.

For the AWS Java SDK implementation of AWS version 3 signing, see the AWSSigner.java class.

Requesting AWS Security Token Service Authentication for Amazon DynamoDB

Topics

This section explains how AWS authenticates your requests.

What Is AWS Security Token Service Authentication?

Authentication is a process for identifying and verifying who is sending a request. Amazon DynamoDB requires users to acquire credentials from the AWS Security Token Service for speed and efficiency in the authentication process. When the AWS Security Token Service creates the temporary security credentials, you can configure how long the credentials remain valid. For security reasons, the lifetime of a security token for an AWS account's root identity is restricted to one hour; however, temporary credentials for IAM users, or for federated user credentials retrieved by IAM users can be valid for up to 36 hours.

Note

The AWS SDKs manage AWS Security Token Service credentials for you. You need only enter your AWS account key pair, as explained in Using the AWS SDKs with Amazon DynamoDB (p. 259).

Getting Credentials

If you don't use one of the AWS SDKs, you need to request temporary security credentials from the AWS Security Token Service. Amazon DynamoDB uses the temporary security credentials returned by the AWS Security Token Service; do not provide a user's own AWS account private key pair directly to Amazon DynamoDB.

General Process for using AWS Security Token Service Authentication

1	Obtain AWS account credentials or IAM user credentials with permission to use Amazon DynamoDB.
2	Use your credentials to request a set of temporary security credentials from the AWS Security Token Service. The resulting session credentials contain: an access key ID, a secret key, and a security token. **Note** Cache the temporary security credentials for the duration of the session. Do not request new credentials for every request to Amazon DynamoDB. The AWS Security Token Service limits the rate of requests for credentials per account, and the latency for your operations increases significantly if you request new credentials for every transaction. For the complete AWS Security Token Service API, including sample requests, go to the AWS Security Token Service API Reference.
3	Use the temporary security credentials to sign your requests to Amazon DynamoDB. For more information about signing a request to Amazon DynamoDB, see Calculating the HMAC-SHA256 Signature for Amazon DynamoDB (p. 375).
4	Provide the signature in the header of your request to Amazon DynamoDB. For more information about forming a request to Amazon DynamoDB, see HTTP Header Contents (p. 372).

Handling Errors in Amazon DynamoDB

Topics

This section describes how to handle client and server errors. For information on specific error messages, see API Error Codes (p. 379).

Error Types

While interacting with Amazon 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 datatypes`).

Client Errors

Client errors are indicated by a 4xx HTTP response code.

Client errors indicate that Amazon 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 Amazon 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 on retrying requests, see Error Retries and Exponential Backoff (p. 383).

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 Calculating the HMAC-SHA256 Signature for Amazon DynamoDB (p. 375).	N

HTTP Status Code	Error code	Message	Cause	Retry
400	LimitExceededException	Too many operations for a given subscriber.	Example: The number of concurrent table requests (cumulative number of tables in the *CREATING*, *DELETING* or *UPDATING* state) exceeds the maximum allowed of 20. The total limit of tables (currently in the *ACTIVE* state) is 250.	N
400	MissingAuthenticationTokenException	Request must contain a valid (registered) AWS Access Key ID.	The request did not include the required x-amz-security-token. See Making HTTP Requests to Amazon DynamoDB (p. 372).	N

HTTP Status Code	Error code	Message	Cause	Retry
400	ProvisionedThroughputExceededException	You exceeded your maximum allowed provisioned throughput.	Example: Your request rate is too high or the request is too large. The AWS SDKs for Amazon DynamoDB automatically retry requests that receive this exception. So, your request is eventually successful, unless the request is too large or your retry queue is too large to finish. Reduce the frequency of requests, using Error Retries and Exponential Backoff (p. 383). Or, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65) for other strategies.	Y
400	ResourceInUseException	The resource which is being attempted to be changed is in use.	Example: You attempted to recreate an existing table, or delete a table currently in the CREATING state.	N
400	ResourceNotFoundException	The resource which is being requested does not exist.	Example: Table which is being requested does not exist, or is too early in the CREATING state.	N
400	ThrottlingException	Rate of requests exceeds the allowed throughput.	This can be returned by the control plane API (CreateTable, DescribeTable, etc) when they are requested too rapidly.	Y

HTTP Status Code	Error code	Message	Cause	Retry
400	ValidationException	One or more required parameter values were missing.	One or more required parameter values were missing.	N
413		Request Entity Too Large.	Maximum item size of 1MB 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
500	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#ProvisionedThroughputExceededExcep
tion",
"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 Amazon 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 Amazon 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 Amazon DynamoDB implements retry logic, automatically.The AWS SDK for Java automatically retries requests, and you can configure the 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 Amazon DynamoDB (p. 259).

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 2400 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 Provisioned Throughput Guidelines in Amazon DynamoDB (p. 68).

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 Amazon DynamoDB request

  IF Exception.errorCode = ProvisionedThroughputExceededException
    set retry to true
  ELSE IF Exception.httpStatusCode = 500
    set retry to true
  ELSE IF Exception.httpStatusCode = 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
```

Operations in Amazon DynamoDB

Topics

This section contains detailed descriptions of all Amazon DynamoDB operations, their request parameters, their response elements, any special errors, and examples of requests and responses.

Amazon DynamoDB uses a JSON data format to send and receive data. The examples in this API Reference show the request and response for each operation using the JSON data format. For more information about the format, see JSON Data Format in Amazon DynamoDB (p. 371).

BatchGetItem

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 the size limit. If the response size limit is exceeded or a partial result is returned due to an internal processing failure, Amazon DynamoDB returns an *`UnprocessedKeys`* value so you can retry the operation starting with the next item to get. Amazon 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 necessary, your application needs its own logic to assemble the pages of results into one set.

Note

The `BatchGetItem` operation is eventually consistent, only. For string consistency, use *`GetItem`* with *`ConsistentRead`* set to *`true`*.

`BatchGetItem` fetches items in parallel to minimize response latencies.

When designing your application, keep in mind that Amazon 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 requested items do not exist, nothing is returned in the response for those items.

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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":"Key
Value2"}},
            {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement":{"N":"Key
Value4"}},
            {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement":{"N":"Key
Value6"}}],
        "AttributesToGet":["AttributeName1", "AttributeName2", "Attribute
Name3"]},
    "Table2":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}}],
       "AttributesToGet": ["AttributeName4", "AttributeName5", "AttributeName6"]
```

```
        }
    }
}
```

Name	Description	Required
RequestItems	A container of the table name and corresponding items to get by primary key. While requesting items, each table name can be invoked only once per operation. Type: String Default: None	Yes
Table	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
Table:*Keys*	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
Table:*AttributesToGet*	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

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", "Attribute
Value"]}
        },
        {"AttributeName1": {"S": "AttributeValue"},
        "AttributeName2": {"S": "AttributeValue"},
        "AttributeName3": {"NS": ["AttributeValue", "AttributeValue", "Attrib
uteValue"]}
```

```
        }],
    "ConsumedCapacityUnits":1},
    "Table2": 
        {"Items":
        [{"AttributeName1": {"S":"AttributeValue"},
        "AttributeName2": {"N":"AttributeValue"},
       "AttributeName3": {"SS":["AttributeValue", "AttributeValue", "Attribute
Value"]}
        },
        {"AttributeName1": {"S": "AttributeValue"},
        "AttributeName2": {"S": "AttributeValue"},
       "AttributeName3": {"NS": ["AttributeValue", "AttributeValue","Attribute
Value"]}
        }],
    "ConsumedCapacityUnits":1}
    },
    "UnprocessedKeys":
        {"Table3": 
        {"Keys": 
           [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement":{"N":"Key
Value2"}},
            {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement":{"N":"Key
Value4"}},
            {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement":{"N":"Key
Value6"}}],
       "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]}

        }
}
```

Name	Description
Responses	Table names and the respective item attributes from the tables. Type: Map
Table	The name of the table containing the items. The entry is simply a string specifying the table with no label. Type: String
Items	Container for the attribute names and values meeting the operation parameters. Type: Map of attribute names to and their data types and values.
ConsumedCapacityUnits	The number of read capacity units consumed, for each table. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Number

Name	Description
UnprocessedKeys	Contains an array of tables and their respective keys that were not processed with the current response, possibly due to reaching a limit on the response size. The *UnprocessedKeys* value is in the same form as a *RequestItems* parameter (so the value can be provided directly to a subsequent `BatchGetItem` operation). For more information, see the above *RequestItems* parameter. Type: Array
UnprocessedKeys: *Table*: *Keys*	The primary key attribute values that define the items and the attributes associated with the items. For more information about primary keys, see Primary Key (p. 5) . Type: Array of attribute name-value pairs.
UnprocessedKeys: *Table*: *AttributesToGet*	Attribute names within the specified table. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array of attribute names.

Special Errors

No errors specific to this API.

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 Amazon DynamoDB (p. 102).

Sample Request

The following sample requests attributes from two different tables.

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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"},"RangeKeyEle
```

```
ment":{"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

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 (EMR) or migrate data from another database in to Amazon 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. For more information about consumed capacity units, see Working with Tables in Amazon DynamoDB (p. 64).

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.

Amazon 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.

Requests

Syntax

```
POST / HTTP/1.1
Host: dynamodb.region.amazonaws.com
Date: date
X-Amzn-Authorization: AWS3 AWSAccessKeyId=TemporaryAccessKeyID,Algorithm=Hmac
SHA256,SignedHeaders=Host;Date;X-Amz-Target;X-Amz-Security-Token,Signature=sig
nature
X-Amz-Security-Token: SecurityToken
Content-Type: application/x-amz-json-1.0
Content-Length: PayloadSizeBytes
X-Amz-Target: DynamoDB_20111205.BatchWriteItem

{
  "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| StringSet | NumericSet

Numeric ::=
{
   "N": "Number"
}

String ::=
{
   "S": "String"
}

StringSet ::=
{
   "SS": [ "String1", "String2", ... ]
}
```

```
NumberSet ::=
{
   "NS": [ "Number1", "Number2", ... ]
}
```

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 Amazon DynamoDB (p. 102).

Sample Request

```
POST / HTTP/1.1
Host: dynamodb.us-east-1.amazonaws.com
Date: Thu, 05 Apr 2012 18:22:09 GMT
X-Amzn-Authorization: AWS3 AWSAccessKeyId=YourTempAccessKeyId,Algorithm=Hmac
SHA256,SignedHeaders=Host;Date;X-Amz-Target;X-Amz-Security-Token,Signature=Sig
natureValue
X-Amz-Security-Token: Security Token
Content-Type: application/x-amz-json-1.0
Content-Length: 830
X-Amz-Target: DynamoDB_20111205.BatchWriteItem

{
  "RequestItems":{
    "Reply":[
      {
        "PutRequest":{
          "Item":{
            "ReplyDateTime":{
              "S":"2012-04-03T11:04:47.034Z"
            },
            "Id":{
              "S":"Amazon DynamoDB#DynamoDB Thread 5"
            }
          }
        }
      },
      {
        "DeleteRequest":{
          "Key":{
            "HashKeyElement":{
              "S":"Amazon DynamoDB#DynamoDB Thread 4"
            },
            "RangeKeyElement":{
              "S":"oops - accidental row"
            }
          }
        }
      }
    ],
    "Thread":[
      {
        "PutRequest":{
          "Item":{
            "ForumName":{
              "S":"Amazon 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":"Amazon DynamoDB#DynamoDB Thread 4"
                  },
                  "RangeKeyElement":{
                     "S":"oops - accidental row"
                  }
               }
            }
         }
      ]
   }
}
```

CreateTable

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-east-1.amazonaws.com). Each Amazon DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-east-1.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. Amazon 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 DescribeTable (p. 406) API to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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","Attribute
Type":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10}
}
```

Name	Description	Required
`TableName`	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
`KeySchema`	The primary key (simple or composite) structure for the table. A name-value pair for the *`HashKeyElement`* is required, and a name-value pair for the *`RangeKeyElement`* is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5). Primary key element names can be between 1 and 255 characters long with no character restrictions. Type: Map of *`HashKeyElement`*, or *`HashKeyElement`* and *`RangeKeyElement`* for a composite primary key.	Yes

Name	Description	Required
ProvisionedThroughput	New throughput for the specified table, consisting of values for *ReadCapacityUnits* and *WriteCapacityUnits*. For details, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). **Note** For current maximum/minimum values, see Limits in Amazon DynamoDB (p. 257). Type: Array	Yes
ProvisionedThroughput: *ReadCapacityUnits*	Sets the minimum number of consistent *ReadCapacityUnits* consumed per second for the specified table before Amazon 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 *ReadCapacityUnits* per second provides 100 eventually consistent *ReadCapacityUnits* per second. Type: Number	Yes
ProvisionedThroughput: *WriteCapacityUnits*	Sets the minimum number of *WriteCapacityUnits* consumed per second for the specified table before Amazon DynamoDB balances the load with other operations. Type: Number	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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","Attribute
Type":"N"}},
```

```
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"CREATING"
    }
}
```

Name	Description
TableDescription	A container for the table properties.
CreationDateTime	Date when the table was created in UNIX epoch time. Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the *HashKeyElement* is required, and a name-value pair for the *RangeKeyElement* is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5) . Type: Map of *HashKeyElement*, or *HashKeyElement* and *RangeKeyElement* for a composite primary key.
ProvisionedThroughput	Throughput for the specified table, consisting of values for *ReadCapacityUnits* and *WriteCapacityUnits*. See Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Array
ProvisionedThroughput :*ReadCapacityUnits*	The minimum number of *ReadCapacityUnits* consumed per second before Amazon DynamoDB. balances the load with other operations Type: Number
ProvisionedThroughput :*WriteCapacityUnits*	The minimum number of *ReadCapacityUnits* consumed per second before *WriteCapacityUnits*. balances the load with other operations Type: Number
TableName	The name of the deleted table. Type: String
TableStatus	The current state of the table (*CREATING*). Once the table is in the *ACTIVE* state, you can put data in it. Use the DescribeTable (p. 406) API to check the status of the table. Type: String

Special Errors

Error	Description
ResourceInUse	Attempt to recreate an already existing table.

Error	Description
`LimitExceeded`	The number of simultaneous table requests (cumulative number of tables in the *CREATING*, *DELETING* or *UPDATING* state) exceeds the maximum allowed. **Note** For current maximum/minimum values, see Limits in Amazon DynamoDB (p. 257). .

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 Amazon DynamoDB (p. 64).

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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

- DescribeTable (p. 406)
- DeleteTable (p. 403)

DeleteItem

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, Amazon 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 Amazon DynamoDB JSON Request
and Response (p. 374).
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":"At
tributeValue2"}},
    "Expected":{"AttributeName3":{"Value":{"S":"AttributeValue3"}}},
    "ReturnValues":"ALL_OLD"}
}
```

Name	Description	Required
TableName	The name of the table containing the item to delete. Type: String	Yes

Name	Description	Required
Key	The primary key that defines the item. For more information about primary keys, see Primary Key (p. 5). Type: Map of `HashKeyElement` to its value and `RangeKeyElement` to its value.	Yes
Expected	Designates an attribute for a conditional delete. The `Expected` parameter allows you to provide an attribute name, and whether or not Amazon DynamoDB should check to see if the attribute has a particular value before deleting it. Type: Map of attribute names.	No
Expected:AttributeName	The name of the attribute for the conditional put. Type: String	No

<table>
<tr><th>Name</th><th>Description</th><th>Required</th></tr>
<tr><td>Expected:AttributeName: ExpectedAttributeValue</td><td>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.
The following JSON notation deletes the item if the "Color" attribute doesn't exist for that item:
<pre>"Expected" :
 {"Color":{"Exists":false}}</pre>
The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before deleting the item:
<pre>"Expected" :
 {"Color":{"Ex
ists":true},{"Value":{"S":"Yel
low"}}}</pre>
By default, if you use the Expected parameter and provide a Value, Amazon DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify {"Exists":true}, because it is implied. You can shorten the request to:
<pre>"Expected" :
 {"Color":{"Value":{"S":"Yellow"}}}</pre>
Note
If you specify {"Exists":true} without an attribute value to check, Amazon DynamoDB returns an error.</td><td>No</td></tr>
<tr><td>ReturnValues</td><td>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.
Type: String</td><td>No</td></tr>
</table>

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
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","Attribute
Value5"]},
    "AttributeName2":{"S":"AttributeValue2"},
    "AttributeName1":{"N":"AttributeValue1"}
    },
"ConsumedCapacityUnits":1
}
```

Name	Description
Attributes	If the *ReturnValues* parameter is provided as *ALL_OLD* in the request, Amazon DynamoDB returns an array of attribute name-value pairs (essentially, the deleted item). Otherwise, the response contains an empty set. Type: Array of attribute name-value pairs.
ConsumedCapacityUnits	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Number

Special Errors

Error	Description
ConditionalCheckFailed	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 Amazon DynamoDB JSON Request
and Response (p. 374).
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"}
    },
"WritesUsed":1
}
```

Related Actions

- PutItem (p. 413)

DeleteTable

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 Amazon 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 Amazon DynamoDB returns a `ResourceInUseException` error. If the specified table does not exist, Amazon DynamoDB returns a `ResourceNotFoundException`. If table is already in the *`DELETING`* state, no error is returned.

Note

Amazon 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-east-1.amazonaws.com). Each Amazon DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-east-1.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 DescribeTable (p. 406) API to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1"}
```

Name	Description	Required
TableName	The name of the table to delete. Type: String	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Sun, 14 Aug 2011 22:56:22 GMT

{"TableDescription":
    {"CreationDateTime":1.313362508446E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"DELETING"
    }
}
```

Name	Description
TableDescription	A container for the table properties.
CreationDateTime	Date when the table was created. Type: Number

Name	Description
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the *HashKeyElement* is required, and a name-value pair for the *RangeKeyElement* is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 5). Type: Map of *HashKeyElement*, or *HashKeyElement* and *RangeKeyElement* for a composite primary key.
ProvisionedThroughput	Throughput for the specified table, consisting of values for *ReadCapacityUnits* and *WriteCapacityUnits*. See Specifying Read and Write Requirements (Provisioned Throughput) (p. 65).
ProvisionedThroughput: *ReadCapacityUnits*	The minimum number of *ReadCapacityUnits* consumed per second for the specified table before Amazon DynamoDB balances the load with other operations. Type: Number
ProvisionedThroughput: *WriteCapacityUnits*	The minimum number of *WriteCapacityUnits* consumed per second for the specified table before Amazon DynamoDB balances the load with other operations. Type: Number
TableName	The name of the deleted table. Type: String
TableStatus	The current state of the table (*DELETING*). Once the table is deleted, subsequent requests for the table return *resource not found*. Use the DescribeTable (p. 406) API to check the status of the table. Type: String

Special Errors

Error	Description
ResourceInUseException	Table is in state *CREATING* or *UPDATING* and can't be deleted.

Examples

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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. 394)
- DescribeTable (p. 406)

DescribeTable

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, Amazon DynamoDB returns a `ResourceNotFound` exception. 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 Amazon DynamoDB JSON Request
and Response (p. 374).
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1"}
```

Name	Description	Required
TableName	The name of the table to describe. Type: String	Yes

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
Content-Length: 543


{"Table":
    {"CreationDateTime":1.309988345372E9,
    ItemCount:1,
    "KeySchema":
       {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},

        "RangeKeyElement":{"AttributeName":"AttributeName2","Attribute
Type":"N"}},
    "ProvisionedThroughput":{"LastIncreaseDateTime": Date, "LastDecreaseDate
Time": Date, "ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableSizeBytes":1,
    "TableStatus":"ACTIVE"
    }
}
```

Name	Description
Table	Container for the table being described. Type: String
CreationDateTime	Date when the table was created in UNIX epoch time.
ItemCount	Number of items in the specified table. Amazon DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the *HashKeyElement* is required, and a name-value pair for the *RangeKeyElement* is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separately (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary Key (p. 5) .

Name	Description
ProvisionedThroughput	Throughput for the specified table, consisting of values for *LastIncreaseDateTime* (if applicable), *LastDecreaseDateTime* (if applicable), *ReadCapacityUnits* and *WriteCapacityUnits*. If the throughput for the table has never been increased or decreased, Amazon DynamoDB does not return values for those elements. See Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Array
TableName	The name of the requested table. Type: String
TableSizeBytes	Total size of the specified table, in bytes. Amazon DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
TableStatus	The current state of the table (*CREATING*, *ACTIVE*, *DELETING* or *UPDATING*). Once the table is in the *ACTIVE* state, you can add data.

Special Errors

No errors are specific to this 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 Amazon DynamoDB JSON Request
and Response (p. 374).
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, "ReadCa
pacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"users",
    "TableSizeBytes":949,
    "TableStatus":"ACTIVE"
    }
}
```

Related Actions

- CreateTable (p. 394)
- DeleteTable (p. 403)
- ListTables (p. 411)

GetItem

Description

The `GetItem` operation returns a set of `Attributes` for an item that matches the primary key.

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. 7).

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
  "Key":
  {"HashKeyElement": {"S":"AttributeValue1"},
  "RangeKeyElement": {"N":"AttributeValue2"}
 },
 "AttributesToGet":["AttributeName3","AttributeName4"],
 "ConsistentRead":Boolean
}
```

Name	Description	Required
TableName	The name of the table containing the requested item. Type: String	Yes
Key	The primary key values that define the item. For more information about primary keys, see Primary Key (p. 5). Type: Map of *HashKeyElement* to its value and *RangeKeyElement* to its value.	Yes
AttributesToGet	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
ConsistentRead	If set to `true`, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 144

{"Item":{
 "AttributeName3":{"S":"AttributeValue3"},
 "AttributeName4":{"N":"AttributeValue4"}
 },
"ConsumedCapacityUnits": 0.5
}
```

Name	Description
`Item`	Contains the requested attributes. Type: Map of attribute name-value pairs.
ConsumedCapacityUnits	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Number

Special Errors

No errors specific to this API.

Examples

For examples using the AWS SDK, see Working with Items in Amazon DynamoDB (p. 102).

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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

Description

Returns an array of all the tables associated with the current account and endpoint. Each Amazon DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-east-1.amazonaws.com and one in dynamodb.us-west-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 Amazon DynamoDB JSON Request
and Response (p. 374).
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.ListTables
content-type: application/x-amz-json-1.0

{"ExclusiveStartTableName":"Table1","Limit":3}
```

The ListTables operation, by default, requests all of the table names associated with the account making the request, for the endpoint that receives the request.

Name	Description	Required
Limit	A number of maximum table names to return. Type: Integer	No
ExclusiveStartTableName	The name of the table that starts the list. If you already ran a ListTables operation and received an *LastEvaluatedTableName* value in the response, use that value here to continue the list. Type: String	No

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT

{"TableNames":["Table1","Table2","Table3"], "LastEvaluatedTableName":"Table3"}
```

Name	Description
TableNames	The names of the tables associated with the current account at the current endpoint. Type: Array

Name	Description
LastEvaluatedTableName	The name of the last table in the current list, only if some tables for the account and endpoint have not been returned. This value does not exist in a response if all table names are already returned. Use this value as the *ExclusiveStartTableName* in a new request to continue the list until all the table names are returned. Type: String

Special Errors

No errors are specific to this 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 Amazon DynamoDB JSON Request
and Response (p. 374).
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

- DescribeTable (p. 406)
- CreateTable (p. 394)
- DeleteTable (p. 403)

PutItem

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.

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 Amazon DynamoDB (p. 102).

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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"},
 },
 "Expected":{"AttributeName3":{"Value": {"S":"AttributeValue"}, "Ex
ists":Boolean}},
 "ReturnValues":"ReturnValuesConstant"}
```

Name	Description	Required
TableName	The name of the table to contain the item. Type: String	Yes
Item	A map of the attributes for the item, and must include the primary key values that define the item. Other attribute name-value pairs can be provided for the item. For more information about primary keys, see Primary Key (p. 5). Type: Map of attribute names to attribute values.	Yes
Expected	Designates an attribute for a conditional put. The *Expected* parameter allows you to provide an attribute name, and whether or not Amazon 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 an attribute names to an attribute value, and whether it exists.	No

Name	Description	Required
Expected:AttributeName	The name of the attribute for the conditional put. Type: String	No

Expected:AttributeName: ExpectedAttributeValue	Use this parameter to specify whether or not a value already exists for the attribute name-value pair.
	The following JSON notation replaces the item if the "Color" attribute doesn't already exist for that item:
	`"Expected" : {"Color":{"Exists":false}}`
	The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before replacing the item:
	`"Expected" : {"Color":{"Exists":true},{"Value":{"S":"Yellow"}}}`
	By default, if you use the *Expected* parameter and provide a *Value*, Amazon DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify `{"Exists":true}`, because it is implied. You can shorten the request to:
	`"Expected" : {"Color":{"Value":{"S":"Yellow"}}}`
	Note
	If you specify `{"Exists":true}` without an attribute value to check, Amazon DynamoDB returns an error.
		No

ReturnValues	Use this parameter if you want to get the attribute name-value pairs before they were updated with the *PutItem* request. Possible parameter values are `NONE` (default) or `ALL_OLD`. If `ALL_OLD` is specified, and `PutItem` overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is `NONE`, nothing is returned. Type: String	No

Responses

Syntax

The following syntax example assumes the request specified a *ReturnValues* parameter of *ALL_OLD*; otherwise, the response has only the *ConsumedCapacityUnits* element.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 85

{"Attributes":
 {"AttributeName3":{"S":"AttributeValue3"},
 "AttributeName2":{"SS":"AttributeValue2"},
 "AttributeName1":{"SS":"AttributeValue1"},
 },
"ConsumedCapacityUnits":1
}
```

Name	Description
Attributes	Attribute values before the put operation, but only if the *ReturnValues* parameter is specified as *ALL_OLD* in the request. Type: Map of attribute name-value pairs.
ConsumedCapacityUnits	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Number

Special Errors

Error	Description
ConditionalCheckFailed	Conditional check failed. An expected attribute value was not found.
ResourceNotFound	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see Working with Items in Amazon DynamoDB (p. 102).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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. 433)
- DeleteItem (p. 399)
- GetItem (p. 409)
- BatchGetItem (p. 385)

Query

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.

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.

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 Amazon DynamoDB JSON Request
and Response (p. 374).
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"}],"Compar
isonOperator":"GT"}
 "ScanIndexForward":true,
 "ExclusiveStartKey":{
  "HashKeyElement":{"S":"AttributeName1"},
  "RangeKeyElement":{"N":"AttributeName2"}
 },
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

Name	Description	Required
TableName	The name of the table containing the requested items. Type: String	Yes
AttributesToGet	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
Limit	The maximum number of items to return (not necessarily the number of matching items). If Amazon 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 *LastEvaluatedKey* to apply in a subsequent operation to continue the query. Also, if the result set size exceeds 1MB before Amazon DynamoDB hits this limit, it stops the query and returns the matching values, and a *LastEvaluatedKey* to apply in a subsequent operation to continue the query. Type: Number	No

Name	Description	Required
ConsistentRead	If set to `true`, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No
Count	If set to `true`, Amazon DynamoDB returns a total number of items that match the query parameters, instead of a list of the matching items and their attributes. Do not set *Count* to `true` while providing a list of *AttributesToGet*, otherwise Amazon DynamoDB returns a validation error. For more information, see Count and ScannedCount (p. 183). Type: Boolean	No
HashKeyValue	Attribute value of the hash component of the composite primary key. Type: String or Number	Yes
RangeKeyCondition	A container for the attribute values and comparison operators to use for the query. A query request does not require a *RangeKeyCondition*. If you provide only the *HashKeyValue*, Amazon DynamoDB returns all items with the specified hash key element value. Type: Map	No
RangeKeyCondition: *AttributeValueList*	The attribute values to evaluate for the query parameters. The *AttributeValueList* contains one attribute value, unless a *BETWEEN* comparison is specified. For the *BETWEEN* comparison, the *AttributeValueList* contains two attribute values. Type: A map of *AttributeValue* to a *ComparisonOperator*.	No

Name	Description	Required
RangeKeyCondition:ComparisonOperator	The criteria for evaluating the provided attributes, such as equals, greater-then, etc. The following are valid comparison operators for a Query 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. Type: String.	No
	`EQ` : Equal. For *`EQ`*, *`AttributeValueList`* can contain only one *`AttributeValue`* of type String or Number (not a set). If an item contains an *`AttributeValue`* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.	
	`LE` : Less than or equal. For *`LE`*, *`AttributeValueList`* can contain only one *`AttributeValue`* of type String or Number (not a set). If an item contains an *`AttributeValue`* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	`LT` : Less than. For *`LT`*, *`AttributeValueList`* can contain only one *`AttributeValue`* of type String or Number (not a set). If an item contains an *`AttributeValue`* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	

Name	Description	Required
	`GE` : Greater than or equal. For `GE`, `AttributeValueList` can contain only one `AttributeValue` of type String or Number (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	`GT` : Greater than. For `GT`, `AttributeValueList` can contain only one `AttributeValue` of type String or Number (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	`BEGINS_WITH` : checks for a substring prefix. For `BEGINS_WITH`, `AttributeValueList` can contain only one `AttributeValue` of type String (not a Number or a set). The target attribute of the comparison must be a String (not a Number or a set).	
	`BETWEEN` : Greater than, or equal to, the first value and less than, or equal to, the second value. For `BETWEEN`, `AttributeValueList` must contain two `AttributeValue` elements of the same type, either String or Number (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not compare to `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
`ScanIndexForward`	Specifies ascending or descending traversal of the index. Amazon DynamoDB returns results reflecting the requested order determined by the range key, based on ASCII character code values. Type: Boolean Default is `true` (ascending).	No

Name	Description	Required
ExclusiveStartKey	Primary key of the item from which to continue an earlier query. An earlier query might provide this value as the *LastEvaluatedKey* if that query operation was interrupted before completing the query; either because of the result set size or the *Limit* parameter. The *LastEvaluatedKey* can be passed back in a new query request to continue the operation from that point. Type: *HashKeyElement*, or *HashKeyElement* and *RangeKeyElement* for a composite primary key.	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"},
 }],
 "LastEvaluatedKey":{"HashKeyElement":{"AttributeValue3":"S"},"RangeKeyEle
ment":{"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. 183). Type: Number

Name	Description
LastEvaluatedKey	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 *LastEvaluatedKey* is *null* when the entire query result set is complete (i.e. the operation processed the "last page"). Type: *HashKeyElement*, or *HashKeyElement* and *RangeKeyElement* 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 (Provisioned Throughput) (p. 65). Type: Number

Special Errors

Error	Description
ResourceNotFound	The specified table was not found.

Examples

For examples using the AWS SDK, see Query and Scan in Amazon DynamoDB (p. 182).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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
Amazon DynamoDB JSON Request and Response (p. 374).
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"}],"ComparisonOperat
or":"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. 425)

Scan

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 Amazon DynamoDB JSON Request
and Response (p. 374).
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"}],"Comparison
Operator":"EQ"}
 },
    "ExclusiveStartKey":{
  "HashKeyElement":{"S":"AttributeName1"},
  "RangeKeyElement":{"N":"AttributeName2"}
 },
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

Name	Description	Required
`TableName`	The name of the table containing the requested items. Type: String	Yes

Name	Description	Required
AttributesToGet	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
Limit	The maximum number of items to return (not necessarily the number of matching items). If Amazon 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 *LastEvaluatedKey* to apply in a subsequent operation to continue retrieving items. Also, if the scanned data set size exceeds 1MB before Amazon DynamoDB hits this limit, it stops the scan and returns the matching values up to the limit, and a *LastEvaluatedKey* to apply in a subsequent operation to continue the scan. For more information see Limit (p. 184). Type: Number	No
Count	If set to *true*, Amazon DynamoDB returns a total number of items for the Scan operation, even if the operation has no matching items for the assigned filter. Do not set *Count* to *true* while providing a list of *AttributesToGet*, otherwise Amazon DynamoDB returns a validation error. For more information, see Count and ScannedCount (p. 183). Type: Boolean	No
ScanFilter	Evaluates the scan results and returns only the desired values. Type: A map of attribute names to values with comparison operators.	No
ScanFilter:*AttributeValueList*	The values and conditions to evaluate the scan results for the filter. Type: A map of *AttributeValue* to a *Condition*.	No

Name	Description	Required
ScanFilter:*ComparisonOperator*	The criteria for evaluating the provided attributes, such as equals, greater-then, 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. Type: String.	No
	EQ : Equal. For *EQ*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (not a set). If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.	
	NE : Not Equal. For *NE*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (not a set). If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.	
	LE : Less than or equal. For *LE*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (not a set). If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	

Name	Description	Required
	LT : Less than. For *LT*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (not a set). If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	GE : Greater than or equal. For *GE*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (not a set). If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	GT : Greater than. For *GT*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (not a set). If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	NOT_NULL : Attribute exists.	
	NULL : Attribute does not exist.	
	CONTAINS : checks for a substring, or value in a set. For *CONTAINS*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (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 a set ("SS" or "NS"), then the operation checks for a member of the set (not as a substring).	

Name	Description	Required
	NOT_CONTAINS : checks for absence of a substring, or absence of a value in a set. For *NOT_CONTAINS*, *AttributeValueList* can contain only one *AttributeValue* of type String or Number (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 a set ("SS" or "NS"), then the operation checks for the absence of a member of the set (not as a substring).	
	BEGINS_WITH : checks for a substring prefix. For *BEGINS_WITH*, *AttributeValueList* can contain only one *AttributeValue* of type String (not a Number or a set). The target attribute of the comparison must be a String (not a Number or a set).	
	IN : checks for exact matches. For *IN*, *AttributeValueList* can contain more than one *AttributeValue* of type String or Number (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.	
	BETWEEN : Greater than, or equal to, the first value and less than, or equal to, the second value. For *BETWEEN*, *AttributeValueList* must contain two *AttributeValue* elements of the same type, either String or Number (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an *AttributeValue* of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not compare to `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
ExclusiveStartKey	Primary key of the item from which to continue an earlier scan. An earlier scan might provide this value if that scan operation was interrupted before scanning the entire table; either because of the result set size or the *Limit* parameter. The *LastEvaluatedKey* can be passed back in a new scan request to continue the operation from that point. Type: *HashKeyElement*, or *HashKeyElement* and *RangeKeyElement* for a composite primary key.	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"},
    }],
    "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. 183). Type: Number
`ScannedCount`	Number of items in the complete scan before any filters are applied. A high *`ScannedCount`* value with few, or no, `Count` results indicates an inefficient Scan operation. For more information, see Count and ScannedCount (p. 183). Type: Number
`LastEvaluatedKey`	Primary key of the item where the scan operation stopped. Provide this value in a subsequent scan operation to continue the operation from that point. The *`LastEvaluatedKey`* is *`null`* when the entire scan result set is complete (i.e. the operation processed the “last page”).
`ConsumedCapacityUnits`	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Number

Special Errors

Error	Description
ResourceNotFound	The specified table was not found.

Examples

For examples using the AWS SDK, see Query and Scan in Amazon DynamoDB (p. 182).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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":"1989"},
 "fans":{"SS":["Alexis","Keneau"]},
 "name":{"S":"Bill & Ted's Excellent Adventure"},
 "rating":{"S":"****"}
 }],
    "ConsumedCapacityUnits":0.5
 "ScannedCount":4}
```

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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":"Fredy"}
  }],
"ConsumedCapacityUnits":0.5
"ScannedCount":4
}
```

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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":"Fredy"},"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. 417)
- BatchGetItem (p. 385)

UpdateItem

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.

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. 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 type.

Note

If you use `ADD` to increment or decrement a number value for an item that doesn't exist before the update, Amazon 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) Amazon 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. Amazon 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 Amazon DynamoDB (p. 102).

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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"},"Ac
tion":"PUT"}},
    "Expected":{"AttributeName3":{"Value":{"S":"AttributeValue3_Current"}}},
    "ReturnValues":"ReturnValuesConstant"
}
```

Name	Description	Required
TableName	The name of the table containing the item to update. Type: String	Yes
Key	The primary key that defines the item. For more information about primary keys, see Primary Key (p. 5). Type: Map of *HashKeyElement* to its value and *RangeKeyElement* to its value.	Yes
AttributeUpdates	Map of attribute name to the new value and action for the update. The attribute names specify the attributes to modify, and cannot contain any primary key attributes. Type: Map of attribute name, value, and an action for the attribute update.	
AttributeUpdates:Action	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
Expected	Designates an attribute for a conditional update. The *Expected* parameter allows you to provide an attribute name, and whether or not Amazon 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
Expected:AttributeName	The name of the attribute for the conditional put. Type: String	No

<table>
<tr><th>Name</th><th>Description</th><th>Required</th></tr>
<tr>
<td>Expected:AttributeName: ExpectedAttributeValue</td>
<td>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.
The following JSON notation updates the item if the "Color" attribute doesn't already exist for that item:
<pre><code>"Expected" :
 {"Color":{"Exists":false}}</code></pre>
The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before updating the item:
<pre><code>"Expected" :
 {"Color":{"Ex
ists":true},{"Value":{"S":"Yel
low"}}}</code></pre>
By default, if you use the Expected parameter and provide a Value, Amazon DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify <code>{"Exists":true}</code>, because it is implied. You can shorten the request to:
<pre><code>"Expected" :
 {"Color":{"Value":{"S":"Yellow"}}}</code></pre>
Note
If you specify <code>{"Exists":true}</code> without an attribute value to check, Amazon DynamoDB returns an error.</td>
<td>No</td>
</tr>
</table>

Name	Description	Required
ReturnValues	Use this parameter if you want to get the attribute name-value pairs before they were updated with the *UpdateItem* request. Possible parameter values are NONE (default) or ALL_OLD, UPDATED_OLD, ALL_NEW or UPDATED_NEW. If ALL_OLD is specified, and *UpdateItem* 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. Type: String	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"},
 },
"ConsumedCapacityUnits":1
}
```

Name	Description
Attributes	A map of attribute name-value pairs, but only if the *ReturnValues* parameter is specified as something other than *NONE* in the request. Type: Map of attribute name-value pairs.
ConsumedCapacityUnits	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Number

Special Errors

Error	Description
ConditionalCheckFailed	Conditional check failed. Attribute ("+ name +") value is ("+ value +") but was expected ("+ expValue +")
ResourceNotFound	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see Working with Items in Amazon DynamoDB (p. 102).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see Sample
Amazon DynamoDB JSON Request and Response (p. 374).
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. 413)
- DeleteItem (p. 399)

UpdateTable

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 Amazon DynamoDB. For more information, see Specifying Read and Write Requirements (Provisioned Throughput) (p. 65).

The provisioned throughput values can be upgraded or downgraded based on the maximums and minimums listed in Limits in Amazon DynamoDB (p. 257).

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 Amazon DynamoDB JSON Request
and Response (p. 374).
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}
}
```

Name	Description	Required
TableName	The name of the table to update. Type: String	Yes
ProvisionedThroughput	New throughput for the specified table, consisting of values for *ReadCapacityUnits* and *WriteCapacityUnits*. See Specifying Read and Write Requirements (Provisioned Throughput) (p. 65). Type: Array	Yes

Name	Description	Required
ProvisionedThroughput:ReadCapacityUnits	Sets the minimum number of consistent *ReadCapacityUnits* consumed per second for the specified table before Amazon 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 *ReadCapacityUnits* per second provides 100 eventually consistent *ReadCapacityUnits* per second. Type: Number	Yes
ProvisionedThroughput:WriteCapacityUnits	Sets the minimum number of *WriteCapacityUnits* consumed per second for the specified table before Amazon DynamoDB balances the load with other operations. Type: Number	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
Content-Type: application/json
Content-Length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.321657838135E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeValue1","Attribute
Type":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeValue2","Attribute
Type":"N"}},
    "ProvisionedThroughput":
        {"LastDecreaseDateTime":1.321661704489E9,
        "LastIncreaseDateTime":1.321663607695E9,
        "ReadCapacityUnits":5,
        "WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"UPDATING"}}
```

Name	Description
CreationDateTime	Date when the table was created. Type: Number

Name	Description
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the *HashKeyElement* is required, and a name-value pair for the *RangeKeyElement* is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separately (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary Key (p. 5). Type: Map of *HashKeyElement*, or *HashKeyElement* and *RangeKeyElement* for a composite primary key.
ProvisionedThroughput	Current throughput settings for the specified table, including values for *LastIncreaseDateTime* (if applicable), *LastDecreaseDateTime* (if applicable), Type: Array
TableName	The name of the updated table. Type: String
TableStatus	The current state of the table (*CREATING*, *ACTIVE*, *DELETING* or *UPDATING*), which should be *UPDATING*. Use the DescribeTable (p. 406) API to check the status of the table. Type: String

Special Errors

Error	Description
ResourceNotFound	The specified table was not found.
ResourceInUse	The table is not in the *ACTIVE* state.

Examples

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see Sample Amazon DynamoDB JSON Request
and Response (p. 374).
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. 394)
- DescribeTable (p. 406)
- DeleteTable (p. 403)

Document History for Amazon DynamoDB

This Document History describes the important changes to the documentation in this release of *Amazon DynamoDB*.

Relevant Dates to this History:

- **Current product version—**2011-12-05
- **Latest product release—**January 2012
- **Last document update—**24 April 2012

Change	Description	Release Date
New endpoints	Amazon 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.	In this release
BatchWriteItem API support	Amazon 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 Amazon DynamoDB batch write API, see BatchWriteItem (p. 389). For information about working with items and using batch write feature using AWS SDKs, see Working with Items in Amazon DynamoDB (p. 102) and Using the AWS SDKs with Amazon DynamoDB (p. 259).	19 April 2012
Documented more error codes	For more information, see Handling Errors in Amazon DynamoDB (p. 378).	5 April 2012

Change	Description	Release Date
Added "Getting Started with Amazon DynamoDB" video	A video is now embedded in the Amazon DynamoDB Developer Guide to provide an overview of the service and steps for creating your first table, adding data to the table using the AWS SDK for Java, and monitoring the table in Amazon CloudWatch. For more information, see Getting Started with Amazon DynamoDB (p. 11).	23 March 2012
Updated Hive version to 0.7.1.3	Amazon Elastic MapReduce now supports a new version of Hive. For more information, see Exporting, Importing, Querying, and Joining Tables in Amazon DynamoDB Using Amazon EMR (p. 230).	13 March 2012
New endpoint	Amazon DynamoDB expands to the Asia Pacific (Tokyo) Region. For the current list of supported endpoints, see Regions and Endpoints.	29 Feb 2012
Updated Hive version to 0.7.1.2	Amazon Elastic MapReduce now supports a new version of Hive. For more information, see Exporting, Importing, Querying, and Joining Tables in Amazon DynamoDB Using Amazon EMR (p. 230).	28 Feb 2012
Clarified use of the AWS Security Token Service	For more information, see Requesting AWS Security Token Service Authentication for Amazon DynamoDB (p. 377).	24 Feb 2012
`ReturnedItemCount` metric added	A new metric, `ReturnedItemCount`, provides the number of items returned in the response of a Query or Scan operation for Amazon DynamoDB is available for monitoring through Amazon CloudWatch. For more information, see Monitoring Amazon DynamoDB Tables with Amazon CloudWatch (p. 223).	24 Feb 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 for Amazon DynamoDB (p. 216).	2 Feb 2012
Added examples for incrementing values	Amazon DynamoDB supports incrementing and decrementing existing numeric values. Examples show adding to existing values in the "Updating an Item" sections at: Working with Items Using the AWS SDK for Java Low-Level API for Amazon DynamoDB (p. 105). Working with Items Using the AWS SDK for .NET Low-Level API for Amazon DynamoDB (p. 133). Working with Items Using the AWS SDK for PHP Low-Level API for Amazon DynamoDB (p. 160).	25 Jan 2012
Initial product release	Amazon DynamoDB is introduced as a new service in Beta release.	18 Jan 2012

Appendix for Amazon DynamoDB

Example Tables and Data in Amazon DynamoDB

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 (Id, ...)	Hash	Attribute Name: Id Type: Number	-
Forum (Name, ...)	Hash	Attribute Name: Name Type: String	-
Thread (ForumName, Subject, ...)	Hash and Range	Attribute Name: ForumName Type: String	Attribute Name: Subject Type: String
Reply (Id, ReplyDateTime, ...)	Hash and Range	Attribute Name: Id Type: String	Attribute Name: ReplyDateTime Type: String

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 an Amazon 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.

Amazon 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 Amazon DynamoDB data model, see Amazon 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 in Amazon DynamoDB (p. 17).

Id (Primary Key)	Other Attributes
101	`{ Title = "Book 101 Title" ISBN = "111-1111111111" Authors = "Author 1" Price = -2 Dimensions = "8.5 x 11.0 x 0.5" PageCount = 500 InPublication = 1 ProductCategory = "Book" }`
102	`{ 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 = 1 ProductCategory = "Book" }`
103	`{ 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 = 0 ProductCategory = "Book" }`

<table>
<tr><th>Id (Primary Key)</th><th>Other Attributes</th></tr>
<tr><td>201</td><td><pre>{
   Title = "18-Bicycle 201"
   Description = "201 description"
   BicycleType = "Road"
   Brand = "Brand-Company A"
   Price = 100
   Gender = "M"
   Color = ["Red", "Black"]
   ProductCategory = "Bike"
}</pre></td></tr>
<tr><td>202</td><td><pre>{
   Title = "21-Bicycle 202"
   Description = "202 description"
   BicycleType = "Road"
   Brand = "Brand-Company A"
   Price = 200
   Gender = "M"
   Color = ["Green", "Black"]
   ProductCategory = "Bike"
}</pre></td></tr>
<tr><td>203</td><td><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></td></tr>
<tr><td>204</td><td><pre>{
   Title = "18-Bicycle 204"
   Description = "204 description"
   BicycleType = "Mountain"
   Brand = "Brand-Company B"
   Price = 400
   Gender = "W"
   Color = ["Red"]
   ProductCategory = "Bike"
}</pre></td></tr>
</table>

Id (Primary Key)	Other Attributes
205	`{ Title = "20-Bicycle 205" Description = "205 description" BicycleType = "Hybrid" Brand = "Brand-Company C" Price = 500 Gender = "B" Color = [ "Red", "Black" ] ProductCategory = "Bike" }`

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 in Amazon DynamoDB (p. 17).

Name (Primary Key)	Other Attributes
"Amazon DynamoDB"	`{ Category="Amazon Web Services" Threads=3 Messages=4 Views=1000 LastPostBy="User A" LastPostDateTime= "2012-01-03T00:40:57.165Z" }`
"Amazon S3"	`{ Category="Amazon Web Services" Threads=1 }`

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 in Amazon DynamoDB (p. 17).

Note that, the LastPostDateTime values are 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 = "Amazon DynamoDB"
Subject = "DynamoDB Thread 1"
	<pre>{
	   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"
	 }</pre>

ForumName = "Amazon DynamoDB"
Subject = "DynamoDB Thread 2"
	<pre>{
	   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"
	 }</pre>

ForumName = "Amazon S3"
Subject = "S3 Thread 1"
	<pre>{
	   Message = "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"
	 }</pre>

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 in Amazon DynamoDB (p. 17).

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 = "Amazon DynamoDB#DynamoDB Thread 1"
ReplyDateTime = "2011-12-11T00:40:57.165Z"
	<pre>{
	   Message = "DynamoDB Thread 1 Reply 1 text"
	   PostedBy = "User A"
	 }</pre>

Primary Key	Other Attributes
Id = "Amazon DynamoDB#DynamoDB Thread 1" ReplyDateTime = "2011-12-18T00:40:57.165Z"	`{ Message = "DynamoDB Thread 1 Reply 1 text" PostedBy = "User A" }`
Id = "Amazon DynamoDB#DynamoDB Thread 1" ReplyDateTime = "2011-12-25T00:40:57.165Z"	`{ Message = "DynamoDB Thread 1 Reply 3 text" PostedBy = "User B" }`
Id = "Amazon DynamoDB#DynamoDB Thread 2" ReplyDateTime = "2011-12-25T00:40:57.165Z"	`{ Message = "DynamoDB Thread 2 Reply 1 text" PostedBy = "User A" }`
Id = "Amazon 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 for Amazon DynamoDB

Topics

In the Getting Started, you first create tables using the Amazon 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 Low-Level API

The following Java code example creates tables and uploads data to the the tables. The resulting table structure and data is shown in Example Tables and Data in Amazon DynamoDB (p. 445). For step-by-step instructions to run this code using Eclipse, see Running Java Examples for Amazon DynamoDB (p. 260).

```
import java.text.SimpleDateFormat;
```

```
import java.util.Arrays;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;



import com.amazonaws.AmazonServiceException;

import com.amazonaws.auth.AWSCredentials;

import com.amazonaws.auth.PropertiesCredentials;

import com.amazonaws.services.dynamodb.AmazonDynamoDBClient;

import com.amazonaws.services.dynamodb.model.AttributeValue;

import com.amazonaws.services.dynamodb.model.CreateTableRequest;

import com.amazonaws.services.dynamodb.model.CreateTableResult;

import com.amazonaws.services.dynamodb.model.DeleteTableRequest;

import com.amazonaws.services.dynamodb.model.DeleteTableResult;

import com.amazonaws.services.dynamodb.model.DescribeTableRequest;

import com.amazonaws.services.dynamodb.model.KeySchema;

import com.amazonaws.services.dynamodb.model.KeySchemaElement;

import com.amazonaws.services.dynamodb.model.ProvisionedThroughput;

import com.amazonaws.services.dynamodb.model.PutItemRequest;

import com.amazonaws.services.dynamodb.model.TableDescription;

import com.amazonaws.services.dynamodb.model.TableStatus;



public class AmazonDynamoDBSampleData_CreateTablesUploadData {



    static AmazonDynamoDBClient client;

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

    static String productCatalogTableName = "ProductCatalog";

    static String forumTableName = "Forum";

    static String threadTableName = "Thread";
```

```
    static String replyTableName = "Reply";

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

        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");

            uploadSampleProducts(productCatalogTableName);
            uploadSampleForums(forumTableName);
            uploadSampleThreads(threadTableName);
            uploadSampleReplies(replyTableName);

```

```
        } catch (AmazonServiceException ase) {

            System.err.println("Data load script failed.");

        }

    }


    private static void createClient() throws Exception {

        AWSCredentials credentials = new PropertiesCredentials(

              AmazonDynamoDBSampleData_CreateTablesUploadData.class.getResour
ceAsStream("AwsCredentials.properties"));


        client = new AmazonDynamoDBClient(credentials);

    }


    private static void deleteTable(String tableName){

        try {


            DeleteTableRequest request = new DeleteTableRequest()

                .withTableName(tableName);


            DeleteTableResult result = client.deleteTable(request);

            waitForTableToBeDeleted(tableName);


            }   catch (AmazonServiceException ase) {

                System.err.println("Failed to delete table " + tableName);

            }


    }

    private static void createTable(String tableName, long readCapacityUnits,
long writeCapacityUnits,

            String hashKeyName, String hashKeyType) {
```

```
        createTable(tableName, readCapacityUnits, writeCapacityUnits, hashKey
Name,  hashKeyType, null, null);

    }

    private static void createTable(String tableName, long readCapacityUnits,
long writeCapacityUnits,

           String hashKeyName, String hashKeyType, String rangeKeyName, String
 rangeKeyType) {


        try {


            KeySchemaElement hashKey = new KeySchemaElement().withAttribute
Name(hashKeyName).withAttributeType(hashKeyType);

            KeySchema ks = new KeySchema().withHashKeyElement(hashKey);


            if (rangeKeyName != null){

                KeySchemaElement rangeKey = new KeySchemaElement().withAttrib
uteName(rangeKeyName).withAttributeType(rangeKeyType);

                ks.setRangeKeyElement(rangeKey);

            }


            // Provide initial provisioned throughput values as Java long data
 types

          ProvisionedThroughput provisionedthroughput = new ProvisionedThrough
put()

                .withReadCapacityUnits(readCapacityUnits)

                .withWriteCapacityUnits(writeCapacityUnits);


            CreateTableRequest request = new CreateTableRequest()

                .withTableName(tableName)

                .withKeySchema(ks)

                .withProvisionedThroughput(provisionedthroughput);
```

```
            CreateTableResult result = client.createTable(request);
            waitForTableToBecomeAvailable(tableName);

            }   catch (AmazonServiceException ase) {
                System.err.println("Failed to create table " + tableName);
            }
    }

    private static void uploadSampleProducts(String tableName) {

        try {
            // Add books.
            Map<String, AttributeValue> item = new HashMap<String, Attribute
Value>();
            item.put("Id", new AttributeValue().withN("101"));
            item.put("Title", new AttributeValue().withS("Book 101 Title"));
            item.put("ISBN", new AttributeValue().withS("111-1111111111"));
            item.put("Authors", new AttributeValue().withSS(Arrays.asList("Au
thor1")));
            item.put("Price", new AttributeValue().withN("2"));
            item.put("Dimensions", new AttributeValue().withS("8.5 x 11.0 x
0.5"));
            item.put("PageCount", new AttributeValue().withN("500"));
            item.put("InPublication", new AttributeValue().withN("1"));
            item.put("ProductCategory", new AttributeValue().withS("Book"));

          PutItemRequest itemRequest = new PutItemRequest().withTableName(table
Name).withItem(item);
            client.putItem(itemRequest);
            item.clear();
```

```
            item.put("Id", new AttributeValue().withN("102"));

            item.put("Title", new AttributeValue().withS("Book 102 Title"));

            item.put("ISBN", new AttributeValue().withS("222-2222222222"));

            item.put("Authors", new AttributeValue().withSS(Arrays.asList("Au
thor1", "Author2")));

            item.put("Price", new AttributeValue().withN("20"));

            item.put("Dimensions", new AttributeValue().withS("8.5 x 11.0 x
0.8"));

            item.put("PageCount", new AttributeValue().withN("600"));

            item.put("InPublication", new AttributeValue().withN("1"));

            item.put("ProductCategory", new AttributeValue().withS("Book"));


            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();


            item.put("Id", new AttributeValue().withN("103"));

            item.put("Title", new AttributeValue().withS("Book 103 Title"));

            item.put("ISBN", new AttributeValue().withS("333-3333333333"));

            item.put("Authors", new AttributeValue().withSS(Arrays.asList("Au
thor1", "Author2")));

            // Intentional. Later we run scan to find price error. Find items
> 1000 in price.

            item.put("Price", new AttributeValue().withN("2000"));

            item.put("Dimensions", new AttributeValue().withS("8.5 x 11.0 x
1.5"));

            item.put("PageCount", new AttributeValue().withN("600"));

            item.put("InPublication", new AttributeValue().withN("0"));

            item.put("ProductCategory", new AttributeValue().withS("Book"));
```

```
            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();



            // Add bikes.

            item.put("Id", new AttributeValue().withN("201"));

            item.put("Title", new AttributeValue().withS("18-Bike-201")); //
Size, followed by some title.

            item.put("Description", new AttributeValue().withS("201 Descrip
tion"));

            item.put("BicycleType", new AttributeValue().withS("Road"));

            item.put("Brand", new AttributeValue().withS("Mountain A")); //
Trek, Specialized.

            item.put("Price", new AttributeValue().withN("100"));

            item.put("Gender", new AttributeValue().withS("M")); // Men's

            item.put("Color", new AttributeValue().withSS(Arrays.asList("Red",
 "Black")));

           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);

            client.putItem(itemRequest);

            item.clear();



            item.put("Id", new AttributeValue().withN("202"));

            item.put("Title", new AttributeValue().withS("21-Bike-202"));

            item.put("Description", new AttributeValue().withS("202 Descrip
tion"));

            item.put("BicycleType", new AttributeValue().withS("Road"));

            item.put("Brand", new AttributeValue().withS("Brand-Company A"));

            item.put("Price", new AttributeValue().withN("200"));
```

```
            item.put("Gender", new AttributeValue().withS("M"));
          item.put("Color", new AttributeValue().withSS(Arrays.asList("Green",
 "Black")));
           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);
            client.putItem(itemRequest);
            item.clear();


            item.put("Id", new AttributeValue().withN("203"));
            item.put("Title", new AttributeValue().withS("19-Bike-203"));
            item.put("Description", new AttributeValue().withS("203 Descrip
tion"));
            item.put("BicycleType", new AttributeValue().withS("Road"));
            item.put("Brand", new AttributeValue().withS("Brand-Company B"));
            item.put("Price", new AttributeValue().withN("300"));
            item.put("Gender", new AttributeValue().withS("W")); // Women's
            item.put("Color", new AttributeValue().withSS(Arrays.asList("Red",
 "Green", "Black")));
           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);
            client.putItem(itemRequest);
            item.clear();


            item.put("Id", new AttributeValue().withN("204"));
            item.put("Title", new AttributeValue().withS("18-Bike-204"));
            item.put("Description", new AttributeValue().withS("204 Descrip
tion"));
```

```
            item.put("BicycleType", new AttributeValue().withS("Mountain"));
            item.put("Brand", new AttributeValue().withS("Brand-Company B"));
            item.put("Price", new AttributeValue().withN("400"));
            item.put("Gender", new AttributeValue().withS("W"));
          item.put("Color", new AttributeValue().withSS(Arrays.asList("Red")));

           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);
            client.putItem(itemRequest);
            item.clear();


            item.put("Id", new AttributeValue().withN("205"));
            item.put("Title", new AttributeValue().withS("20-Bike-205"));
            item.put("Description", new AttributeValue().withS("205 Descrip
tion"));
            item.put("BicycleType", new AttributeValue().withS("Hybrid"));
            item.put("Brand", new AttributeValue().withS("Brand-Company C"));
            item.put("Price", new AttributeValue().withN("500"));
            item.put("Gender", new AttributeValue().withS("B")); // Boy's
            item.put("Color", new AttributeValue().withSS(Arrays.asList("Red",
 "Black")));
           item.put("ProductCategory", new AttributeValue().withS("Bicycle"));



            itemRequest = new PutItemRequest().withTableName(tableName).with
Item(item);
            client.putItem(itemRequest);



        }   catch (AmazonServiceException ase) {
```

```
            System.err.println("Failed to create item in " + tableName);
        }

    }

    private static void uploadSampleForums(String tableName) {
        try {
            // Add forums.
            Map<String, AttributeValue> forum = new HashMap<String, Attribute
Value>();
            forum.put("Name", new AttributeValue().withS("Amazon DynamoDB"));
            forum.put("Category", new AttributeValue().withS("Amazon Web Ser
vices"));
            forum.put("Threads", new AttributeValue().withN("2"));
            forum.put("Messages", new AttributeValue().withN("4"));
            forum.put("Views", new AttributeValue().withN("1000"));

            PutItemRequest forumRequest = new PutItemRequest().withTable
Name(tableName).withItem(forum);
            client.putItem(forumRequest);
            forum.clear();

            forum.put("Name", new AttributeValue().withS("Amazon S3"));
            forum.put("Category", new AttributeValue().withS("Amazon Web Ser
vices"));
            forum.put("Threads", new AttributeValue().withN("0"));

            forumRequest = new PutItemRequest().withTableName(tableName).with
Item(forum);
            client.putItem(forumRequest);
```

```
        }   catch (AmazonServiceException ase) {
            System.err.println("Failed to create item in " + tableName);
        }
    }

    private static void uploadSampleThreads(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);

            // Add threads.
            Map<String, AttributeValue> forum = new HashMap<String, Attribute
Value>();
            forum.put("ForumName", new AttributeValue().withS("Amazon Dy
namoDB"));
            forum.put("Subject", new AttributeValue().withS("DynamoDB Thread
1"));
            forum.put("Message", new AttributeValue().withS("DynamoDB thread 1
 message"));
            forum.put("LastPostedBy", new AttributeValue().withS("User A"));
```

```
            forum.put("LastPostedDateTime", new AttributeValue().withS(date
Formatter.format(date2)));

            forum.put("Views", new AttributeValue().withN("0"));

            forum.put("Replies", new AttributeValue().withN("0"));

            forum.put("Answered", new AttributeValue().withN("0"));

          forum.put("Tags", new AttributeValue().withSS(Arrays.asList("index",
 "primarykey", "table")));



            PutItemRequest forumRequest = new PutItemRequest().withTable
Name(tableName).withItem(forum);

            client.putItem(forumRequest);



            forum.clear();



            forum.put("ForumName", new AttributeValue().withS("Amazon Dy
namoDB"));

            forum.put("Subject", new AttributeValue().withS("DynamoDB Thread
2"));

            forum.put("Message", new AttributeValue().withS("DynamoDB thread 2
 message"));

            forum.put("LastPostedBy", new AttributeValue().withS("User A"));

            forum.put("LastPostedDateTime", new AttributeValue().withS(date
Formatter.format(date3)));

            forum.put("Views", new AttributeValue().withN("0"));

            forum.put("Replies", new AttributeValue().withN("0"));

            forum.put("Answered", new AttributeValue().withN("0"));

          forum.put("Tags", new AttributeValue().withSS(Arrays.asList("index",
 "primarykey", "rangekey")));



            forumRequest = new PutItemRequest().withTableName(tableName).with
Item(forum);

            client.putItem(forumRequest);
```

```
            forum.clear();


            forum.put("ForumName", new AttributeValue().withS("Amazon S3"));

            forum.put("Subject", new AttributeValue().withS("S3 Thread 1"));

            forum.put("Message", new AttributeValue().withS("S3 Thread 3 mes
sage"));

            forum.put("LastPostedBy", new AttributeValue().withS("User A"));

            forum.put("LastPostedDateTime", new AttributeValue().withS(date
Formatter.format(date1)));

            forum.put("Views", new AttributeValue().withN("0"));

            forum.put("Replies", new AttributeValue().withN("0"));

            forum.put("Answered", new AttributeValue().withN("0"));

          forum.put("Tags", new AttributeValue().withSS(Arrays.asList("largeo
bjects", "multipart upload")));


            forumRequest = new PutItemRequest().withTableName(tableName).with
Item(forum);


            client.putItem(forumRequest);


        }   catch (AmazonServiceException ase) {

            System.err.println("Failed to create item in " + tableName);

        }


    }


    private static void uploadSampleReplies(String tableName) {

        try {

            long time0 = (new Date()).getTime() - (1*24*60*60*1000); // 1 day
ago

            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 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);


            // Add threads.
            Map<String, AttributeValue> reply = new HashMap<String, Attribute
Value>();
          reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 1"));
            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date3)));
            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 1
 Reply 1 text"));
            reply.put("PostedBy", new AttributeValue().withS("User A"));


            PutItemRequest replyRequest = new PutItemRequest().withTable
Name(tableName).withItem(reply);
            client.putItem(replyRequest);


            reply.clear();
```

```
            reply = new HashMap<String, AttributeValue>();
          reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 1"));
            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date2)));
            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 1
 Reply 2 text"));
            reply.put("PostedBy", new AttributeValue().withS("User B"));


            replyRequest = new PutItemRequest().withTableName(tableName).with
Item(reply);
            client.putItem(replyRequest);


            reply.clear();


            reply = new HashMap<String, AttributeValue>();
          reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
 Thread 2"));
            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date1)));
            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 2
 Reply 1 text"));
            reply.put("PostedBy", new AttributeValue().withS("User A"));


            replyRequest = new PutItemRequest().withTableName(tableName).with
Item(reply);
            client.putItem(replyRequest);


            reply.clear();


            reply = new HashMap<String, AttributeValue>();
          reply.put("Id", new AttributeValue().withS("Amazon DynamoDB#DynamoDB
```

```
 Thread 2"));

            reply.put("ReplyDateTime", new AttributeValue().withS(dateFormat
ter.format(date0)));

            reply.put("Message", new AttributeValue().withS("DynamoDB Thread 2
 Reply 2 text"));

            reply.put("PostedBy", new AttributeValue().withS("User A"));



            replyRequest = new PutItemRequest().withTableName(tableName).with
Item(reply);



            client.putItem(replyRequest);



        }   catch (AmazonServiceException ase) {

            System.err.println("Failed to create item in " + tableName);

        }

    }



    private static void waitForTableToBecomeAvailable(String tableName) {

       System.out.println("Waiting for " + tableName + " to become ACTIVE...");



        long startTime = System.currentTimeMillis();

        long endTime = startTime + (10 * 60 * 1000);

        while (System.currentTimeMillis() < endTime) {

            try {

                Thread.sleep(1000 * 20);

            } catch (Exception e) {

            }

            try {

                DescribeTableRequest request = new DescribeTableRequest()

                        .withTableName(tableName);
```

```
                TableDescription tableDescription = client.describeTable(

                        request).getTable();

                String tableStatus = tableDescription.getTableStatus();

                System.out.println("  - current state: " + tableStatus);

                if (tableStatus.equals(TableStatus.ACTIVE.toString()))

                    return;

            } catch (AmazonServiceException ase) {

                if (ase.getErrorCode().equalsIgnoreCase(

                        "ResourceNotFoundException") == false)

                    throw ase;

            }

        }

       throw new RuntimeException("Table " + tableName + " never went active");

    }



    private static void waitForTableToBeDeleted(String tableName) {

        System.out.println("Waiting for " + tableName + " while status DELET
ING...");



        long startTime = System.currentTimeMillis();

        long endTime = startTime + (10 * 60 * 1000);

        while (System.currentTimeMillis() < endTime) {

            try {Thread.sleep(1000 * 20);} catch (Exception e) {}

            try {

              DescribeTableRequest request = new DescribeTableRequest().withT
ableName(tableName);

                TableDescription tableDescription = client.describeTable(re
quest).getTable();

                String tableStatus = tableDescription.getTableStatus();

                System.out.println("  - current state: " + tableStatus);
```

```
                if (tableStatus.equals(TableStatus.ACTIVE.toString())) return;

            } catch (AmazonServiceException ase) {

               if (ase.getErrorCode().equalsIgnoreCase("ResourceNotFoundExcep
tion") == false) {

                    System.out.println("Table " + tableName + " is not found.
It was deleted.");

                    return;

                }

                else {

                    throw ase;

                }

            }

        }

        throw new RuntimeException("Table " + tableName + " never went active");

    }

}
```

Creating Example Tables and Uploading Data Using the AWS SDK for .NET Low-Level API

The following C# code example creates tables and uploads data to the the tables. The resulting table structure and data is shown in Example Tables and Data in Amazon DynamoDB (p. 445). For step-by-step instructions to run this code in Visual Studio, see Running .NET Examples for Amazon DynamoDB (p. 306).

```
using System;

using System.Collections.Generic;

using Amazon.DynamoDB;

using Amazon.DynamoDB.DocumentModel;

using Amazon.DynamoDB.Model;

using Amazon.Runtime;

using Amazon.SecurityToken;
```

```
namespace amazon.dynamodb.documentation

{

  class Program

  {

    private static AmazonDynamoDBClient client;


    static void Main(string[] args)

    {

      CreateClient();


      CreateTablesUploadSampleItems(client);

    }


    private static void CreateClient()

    {

      var userCredentials = new EnvironmentAWSCredentials();

      var stsClient = new AmazonSecurityTokenServiceClient(userCredentials);

      var sessionCredentials = new RefreshingSessionAWSCredentials(stsClient);

      client = new AmazonDynamoDBClient(sessionCredentials);


    }


    private static void CreateTablesUploadSampleItems(AmazonDynamoDBClient 
client)

    {

      try

      {

        //DeleteAllTables(client);

        DeleteTable(client, "ProductCatalog");

```

```
        DeleteTable(client, "Forum");

        DeleteTable(client, "Thread");

        DeleteTable(client, "Reply");


        // Create tables (using the AWS SDK for .NET low-level API).


        CreateTableProductCatalog(client);

        CreateTableForum(client);

        CreateTableThread(client); // ForumTitle, Subject

        CreateTableReply(client);


        // Upload data (using the .NET SDK helper API to upload data)

        UploadSampleProducts(client);

        UploadSampleForums(client);

        UploadSampleThreads(client);

        UploadSampleReplies(client);

        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(AmazonDynamoDBClient client, string table
Name)

    {

      try

      {
```

```
        var deleteTableResponse = client.DeleteTable(new DeleteTableRequest()
{ TableName = tableName });

        WaitTillTableDeleted(client, tableName, deleteTableResponse);

      }

      catch (ResourceNotFoundException resourceNotFound)

      {

        // There is no such table.

      }

    }


    private static void CreateTableProductCatalog(AmazonDynamoDBClient client)

    {

      string tableName = "ProductCatalog";


      var response = client.CreateTable(new CreateTableRequest

      {

        TableName = tableName,

        KeySchema = new KeySchema

        {

          HashKeyElement = new KeySchemaElement

          {

            AttributeName = "Id",

            AttributeType = "N"

          }

        },

        ProvisionedThroughput = new ProvisionedThroughput

        {

          ReadCapacityUnits = 10,

          WriteCapacityUnits = 5
```

```
    }
  });

  WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableForum(AmazonDynamoDBClient client)
{
  string tableName = "Forum";

  var response = client.CreateTable(new CreateTableRequest
  {
    TableName = tableName,
    KeySchema = new KeySchema
    {
      HashKeyElement = new KeySchemaElement
      {
        AttributeName = "Name", // forum Title
        AttributeType = "S"
      }
    },
    ProvisionedThroughput = new ProvisionedThroughput
    {
      ReadCapacityUnits = 10,
      WriteCapacityUnits = 5
    }
  });
```

```
  WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableThread(AmazonDynamoDBClient client)
{
  string tableName = "Thread";

  var response = client.CreateTable(new CreateTableRequest
  {
    TableName = tableName,
    KeySchema = new KeySchema
    {
      HashKeyElement = new KeySchemaElement
      {
        AttributeName = "ForumName", // Hash attribute.
        AttributeType = "S"
      },
      RangeKeyElement = new KeySchemaElement
      {
        AttributeName = "Subject", // Range attribute.
        AttributeType = "S"
      }
    },
    ProvisionedThroughput = new ProvisionedThroughput
    {
      ReadCapacityUnits = 10,
      WriteCapacityUnits = 5
    }
  });
```

```
    WaitTillTableCreated(client, tableName, response);
  }

  private static void CreateTableReply(AmazonDynamoDBClient client)
  {
    string tableName = "Reply";
    var response = client.CreateTable(new CreateTableRequest
    {
      TableName = tableName,
      KeySchema = new KeySchema
      {
        HashKeyElement = new KeySchemaElement
        {
          AttributeName = "Id",
          AttributeType = "S"
        },
        RangeKeyElement = new KeySchemaElement
        {
          AttributeName = "ReplyDateTime",
          AttributeType = "S"
        }
      },
      ProvisionedThroughput = new ProvisionedThroughput
      {
        ReadCapacityUnits = 10,
        WriteCapacityUnits = 5
      }
    });
```

```
      WaitTillTableCreated(client, tableName, response);
    }

    private static void WaitTillTableCreated(AmazonDynamoDBClient client, string
 tableName,
                                             CreateTableResponse response)
    {
      var tableDescription = response.CreateTableResult.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.DescribeTableRes
ult.Table.TableName,
                                                            res.DescribeTableRes
ult.Table.TableStatus);
          status = res.DescribeTableResult.Table.TableStatus;
        }
```

```
        // Try-catch to handle potential eventual-consistency issue.
        catch (ResourceNotFoundException resourceNotFound)
        { }
      }
    }


    private static void WaitTillTableDeleted(AmazonDynamoDBClient client, string
 tableName,
                                             DeleteTableResponse response)
    {
      var tableDescription = response.DeleteTableResult.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.DescribeTableRes
ult.Table.TableName,
                                                          res.DescribeTableRes
ult.Table.TableStatus);
```

```
          status = res.DescribeTableResult.Table.TableStatus;

        }


      }

      catch (ResourceNotFoundException tableNotFound)

      {

        // Table deleted.

      }

    }


    private static void UploadSampleProducts(AmazonDynamoDB client)

    {

      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 UploadSampleForums(AmazonDynamoDB client)
  {
```

```
    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 UploadSampleThreads(AmazonDynamoDB client)
  {
    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 UploadSampleReplies(AmazonDynamoDB client)

    {

      Table replyTable = Table.LoadTable(client, "Reply");



      // Reply 1 - thread 1.

      var thread1Reply1 = new Document();

     thread1Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash attrib
ute.

     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 attrib
ute.

     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 attrib
ute.

     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 attrib
ute.

     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 attrib
ute.
```

```
     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 in Amazon DynamoDB (p. 445). For step-by-step instructions to run this code, see Running PHP Examples for Amazon DynamoDB (p. 369).

```
<?php

// If necessary, reference the sdk.class.php file.

// For example, the following line assumes the sdk.class.php file is

// in an sdk sub-directory relative to this file

require_once dirname(__FILE__) . '/sdk/sdk.class.php';


// Instantiate the class

$dynamodb = new AmazonDynamoDB();


$table_name1 = 'ProductCatalog';

$table_name2 = 'Forum';

$table_name3 = 'Thread';

$table_name4 = 'Reply';


####################################################################
```

```
# Create a new DynamoDB table

$response = $dynamodb->create_table(array(
    'TableName' => $table_name1,
    'KeySchema' => array(
    'HashKeyElement' => array(
        'AttributeName' => 'Id',
        'AttributeType' => AmazonDynamoDB::TYPE_NUMBER
        )
    ),
    'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 10,
    'WriteCapacityUnits' => 5
    )
));

// Check for success...
if ($response->isOK())
{
echo '# Kicked off the creation of the DynamoDB table...' . PHP_EOL;
}
else
{
print_r($response);
}


####################################################################
# Sleep and poll until the table has been created
```

```
$count = 0;
do {
sleep(1);
$count++;

$response = $dynamodb->describe_table(array(
    'TableName' => $table_name1
));
}
while ((string) $response->body->Table->TableStatus !== 'ACTIVE');




####################################################################
# Create a new DynamoDB table

$response = $dynamodb->create_table(array(
    'TableName' => $table_name2,
    'KeySchema' => array(
    'HashKeyElement' => array(
        'AttributeName' => 'Name',
        'AttributeType' => AmazonDynamoDB::TYPE_STRING
        )
    ),
    'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 10,
    'WriteCapacityUnits' => 5
    )
```

```
));


// Check for success...
if ($response->isOK())
{
echo '# Kicked off the creation of the DynamoDB table...' . PHP_EOL;
}
else
{
print_r($response);
}



####################################################################
# Sleep and poll until the table has been created


$count = 0;
do {
sleep(1);
$count++;


$response = $dynamodb->describe_table(array(
    'TableName' => $table_name2
));
}
while ((string) $response->body->Table->TableStatus !== 'ACTIVE');



####################################################################
```

```
# Create a new DynamoDB table


$response = $dynamodb->create_table(array(

'TableName' => $table_name3,

'KeySchema' => array(

    'HashKeyElement' => array(

        'AttributeName' => 'ForumName',

        'AttributeType' => AmazonDynamoDB::TYPE_STRING

        ),

    'RangeKeyElement' => array(

        'AttributeName' => 'Subject',

        'AttributeType' => AmazonDynamoDB::TYPE_STRING

        )

    ),

    'ProvisionedThroughput' => array(

    'ReadCapacityUnits' => 10,

    'WriteCapacityUnits' => 5

    )

));


// Check for success...

if ($response->isOK())

{

echo '# Kicked off the creation of the DynamoDB table...' . PHP_EOL;

}

else

{

print_r($response);

}
```

```
####################################################################
# Sleep and poll until the table has been created

$count = 0;
do {
sleep(1);
$count++;

$response = $dynamodb->describe_table(array(
    'TableName' => $table_name3
));
}
while ((string) $response->body->Table->TableStatus !== 'ACTIVE');


####################################################################
# Create a new DynamoDB table

$response = $dynamodb->create_table(array(
    'TableName' => $table_name4,
    'KeySchema' => array(
    'HashKeyElement' => array(
        'AttributeName' => 'Id',
        'AttributeType' => AmazonDynamoDB::TYPE_STRING
        ),
    'RangeKeyElement' => array(
        'AttributeName' => 'ReplyDateTime',
```

```
        'AttributeType' => AmazonDynamoDB::TYPE_STRING
        )
    ),
    'ProvisionedThroughput' => array(
    'ReadCapacityUnits' => 10,
    'WriteCapacityUnits' => 5
    )
));

// Check for success...
if ($response->isOK())
{
echo '# Kicked off the creation of the DynamoDB table...' . PHP_EOL;
}
else
{
print_r($response);
}


####################################################################
# Sleep and poll until the table has been created

$count = 0;
do {
sleep(1);
$count++;

$response = $dynamodb->describe_table(array(
```

```
    'TableName' => $table_name4
));
}
while ((string) $response->body->Table->TableStatus !== 'ACTIVE');




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

echo PHP_EOL . PHP_EOL;
echo '# Collecting a complete list of tables in the account...' . PHP_EOL;

$response = $dynamodb->list_tables();
print_r($response->body->TableNames()->map_string());

?>
```

The following PHP code example uploads data to the the tables. The resulting table structure and data is shown in Example Tables and Data in Amazon DynamoDB (p. 445).

```
<?php
// If necessary, reference the sdk.class.php file.
// For example, the following line assumes the sdk.class.php file is
// in an sdk sub-directory relative to this file
require_once dirname(__FILE__) . '/sdk/sdk.class.php';

// Instantiate the class
$dynamodb = new AmazonDynamoDB();

####################################################################
```

```
# Setup some local variables for dates


$one_day_ago = date('Y-m-d H:i:s', strtotime("-1 days"));

$seven_days_ago = date('Y-m-d H:i:s', strtotime("-7 days"));

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

$twenty_one_days_ago = date('Y-m-d H:i:s', strtotime("-21 days"));


####################################################################

# Adding data to the table


echo PHP_EOL . PHP_EOL;

echo "# Adding data to the table..." . PHP_EOL;


# Adding data to the table

echo "# Adding data to the table..." . PHP_EOL;


// Set up batch requests

$queue = new CFBatchRequest();

$queue->use_credentials($dynamodb->credentials);


// Add items to the batch

$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'101'              ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Book 101 Title'   ),

        'ISBN'            => array( AmazonDynamoDB::TYPE_STRING           =>
'111-1111111111'   ),

        'Authors'         => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
```

```
array('Author1')   ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER          =>
'2'               ),

        'Dimensions'      => array( AmazonDynamoDB::TYPE_STRING          =>
'8.5 x 11.0 x 0.5' ),

        'PageCount'       => array( AmazonDynamoDB::TYPE_NUMBER          =>
'500'             ),

        'InPublication'   => array( AmazonDynamoDB::TYPE_NUMBER          =>
'1'               ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING          =>
'Book'            )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER          =>
'102'                        ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING          =>
'Book 102 Title'             ),

        'ISBN'            => array( AmazonDynamoDB::TYPE_STRING          =>
'222-2222222222'             ),

        'Authors'         => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Author1', 'Author2') ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER          =>
'20'                         ),

        'Dimensions'      => array( AmazonDynamoDB::TYPE_STRING          =>
'8.5 x 11.0 x 0.8'           ),

        'PageCount'       => array( AmazonDynamoDB::TYPE_NUMBER          =>
'600'                        ),

        'InPublication'   => array( AmazonDynamoDB::TYPE_NUMBER          =>
'1'                          ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING          =>
'Book'                       )

    )
```

```
));


$dynamodb->batch($queue)->put_item(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'103'                       ), // Hash Key
        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Book 103 Title'            ),
        'ISBN'            => array( AmazonDynamoDB::TYPE_STRING           =>
'333-3333333333'            ),
        'Authors'         => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Author1', 'Author2') ),
        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'2000'                      ),
        'Dimensions'      => array( AmazonDynamoDB::TYPE_STRING           =>
'8.5 x 11.0 x 1.5'          ),
        'PageCount'       => array( AmazonDynamoDB::TYPE_NUMBER           =>
'600'                       ),
        'InPublication'   => array( AmazonDynamoDB::TYPE_NUMBER           =>
'0'                         ),
        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Book'                      )
    )
));


$dynamodb->batch($queue)->put_item(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'201'                 ), // Hash Key
        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'18-Bike-201'         ),
        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'201 Description'     ),
```

```
        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Road'                ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Mountain A'          ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'100'                 ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'M'                   ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'             )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'202'                   ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'21-Bike-202'           ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'202 Description'       ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Road'                  ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company A'       ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'200'                   ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'M'                     ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Green', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'               )

    )
```

```
));

$dynamodb->batch($queue)->put_item(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'203'                         ), // Hash Key
        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'19-Bike-203'                 ),
        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'203 Description'             ),
        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Road'                        ),
        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company B'             ),
        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'300'                         ),
        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'W'                           ),
        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red', 'Green', 'Black') ),
        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'                     )
    )
));

$dynamodb->batch($queue)->put_item(array(
    'TableName' => 'ProductCatalog',
    'Item' => array(
        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'204'             ), // Hash Key
        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'18-Bike-204'     ),
        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'204 Description' ),
```

```
        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Mountain'        ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company B' ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'400'             ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'W'               ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red')      ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'         )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'ProductCatalog',

    'Item' => array(

        'Id'              => array( AmazonDynamoDB::TYPE_NUMBER           =>
'205'                 ), // Hash Key

        'Title'           => array( AmazonDynamoDB::TYPE_STRING           =>
'20-Bike-205'         ),

        'Description'     => array( AmazonDynamoDB::TYPE_STRING           =>
'205 Description'     ),

        'BicycleType'     => array( AmazonDynamoDB::TYPE_STRING           =>
'Hybrid'              ),

        'Brand'           => array( AmazonDynamoDB::TYPE_STRING           =>
'Brand-Company C'     ),

        'Price'           => array( AmazonDynamoDB::TYPE_NUMBER           =>
'500'                 ),

        'Gender'          => array( AmazonDynamoDB::TYPE_STRING           =>
'B'                   ),

        'Color'           => array( AmazonDynamoDB::TYPE_ARRAY_OF_STRINGS =>
array('Red', 'Black') ),

        'ProductCategory' => array( AmazonDynamoDB::TYPE_STRING           =>
'Bicycle'             )
```

```
    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Forum',

    'Item' => array(

        'Name'     => array( AmazonDynamoDB::TYPE_STRING => 'Amazon DynamoDB'
    ), // Hash Key

       'Category' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Web Services'
 ),

        'Threads'  => array( AmazonDynamoDB::TYPE_NUMBER => '0'
    ),

        'Messages' => array( AmazonDynamoDB::TYPE_NUMBER => '0'
    ),

        'Views'    => array( AmazonDynamoDB::TYPE_NUMBER => '1000'
    ),

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Forum',

    'Item' => array(

        'Name'     => array( AmazonDynamoDB::TYPE_STRING => 'Amazon S3'
    ), // Hash Key

       'Category' => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Web Services'
 ),

        'Threads'  => array( AmazonDynamoDB::TYPE_NUMBER => '0'
    )

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Reply',
```

```
    'Item' => array(
        'Id'             => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 1' ), // Hash Key
        'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING => $four
teen_days_ago                  ), // Range Key
        'Message'        => array( AmazonDynamoDB::TYPE_STRING => 'DynamoDB
Thread 1 Reply 2 text'    ),
        'PostedBy'       => array( AmazonDynamoDB::TYPE_STRING => 'User B'
                       ),
    )
));
$dynamodb->batch($queue)->put_item(array(
    'TableName' => 'Reply',
    'Item' => array(
        'Id'             => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 2' ), // Hash Key
        'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING =>
$twenty_one_days_ago                    ), // Range Key
        'Message'        => array( AmazonDynamoDB::TYPE_STRING  => 'DynamoDB
Thread 2 Reply 3 text'   ),
        'PostedBy'       => array( AmazonDynamoDB::TYPE_STRING => 'User B'
                       ),
    )
));


$dynamodb->batch($queue)->put_item(array(
    'TableName' => 'Reply',
    'Item' => array(
        'Id'             => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 2' ), // Hash Key
       'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING => $seven_days_ago
                    ), // Range Key
        'Message'        => array( AmazonDynamoDB::TYPE_STRING  => 'DynamoDB
Thread 2 Reply 2 text'   ),
        'PostedBy'       => array( AmazonDynamoDB::TYPE_STRING => 'User A'
```

```
                        ),

    )

));



$dynamodb->batch($queue)->put_item(array(

    'TableName' => 'Reply',

    'Item' => array(

        'Id'             => array( AmazonDynamoDB::TYPE_STRING => 'Amazon Dy
namoDB#DynamoDB Thread 2' ), // Hash Key

        'ReplyDateTime' => array( AmazonDynamoDB::TYPE_STRING => $one_day_ago
                      ), // Range Key

        'Message'       => array( AmazonDynamoDB::TYPE_STRING  => 'DynamoDB
Thread 2 Reply 1 text'   ),

        'PostedBy'      => array( AmazonDynamoDB::TYPE_STRING => 'User A'
                        ),

    )

));



// Execute the batch of requests in parallel

$responses = $dynamodb->batch($queue)->send();



// Check for success...

if ($responses->areOK())

{

    echo "The data has been added to the table." . PHP_EOL;

}

    else

{

    print_r($responses);

}

?>
```