Amazon DynamoDB

Getting Started Guide

API Version 2012-08-10

Amazon DynamoDB: Getting Started Guide

Table of Contents

Getting Started with Amazon DynamoDB

Welcome to the *Amazon DynamoDB Getting Started Guide*. This guide contains hands-on tutorials to help you learn about Amazon DynamoDB. We encourage you to:

- Become familiar with DynamoDB concepts.
- Download and run DynamoDB.
- Work through one of the language-specific tutorials. The sample code in these tutorials can run against either the downloadable version of DynamoDB or the Amazon DynamoDB web service.

  Note
  AWS SDKs are available for a wide variety of languages. For a complete list, see Tools for Amazon Web Services.

We also recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

Topics

Introduction to DynamoDB Concepts

This section briefly introduces some basic DynamoDB concepts. This helps you as you follow the steps in the tutorials.

Tables

Similar to other database management systems, DynamoDB stores data in tables. A *table* is a collection of data. For example, you could create a table named `People`, where you could store information about friends, family, or anyone else of interest. You could also have a `Cars` table to store information about vehicles that people drive.

Items

Each table contains multiple items. An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a `People` table, each item represents one person. For a `Cars` table, each item represents one vehicle. In many ways, items are similar to rows, records, or tuples in relational database systems. In DynamoDB, there is no limit to the number of items you can store in a table.

Attributes

Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, a `Department` item might have attributes such as `DepartmentID`, `Name`, `Manager`, and so on. An item in a `People` table could contain attributes such as `PersonID`, `LastName`, `FirstName`, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database management systems.

Primary Key

When you create a table, in addition to the table name, you must specify the primary key of the table. As in other databases, a primary key in DynamoDB uniquely identifies each item in the table, so that no two items can have the same key. When you add, update, or delete an item in the table, you must specify the primary key attribute values for that item. The key values are required; you cannot omit them.

DynamoDB supports two different kinds of primary keys:

- **Partition key—**A simple primary key, composed of one attribute known as the *partition key*. DynamoDB uses the partition key's value as input to an internal hash function; the output from the hash function determines the partition where the item is stored. With a simple primary key, no two items in a table can have the same partition key value.
- **Partition key and sort key—**A composite primary key, composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*. DynamoDB uses the partition key value as input to an internal hash function; the output from the hash function determines the partition where the item is stored. All items with the same partition key are stored together, in sorted order by sort key value. With a composite primary key, it is possible for two items to have the same partition key value, but those two items must have different sort key values.

Note
The partition key of an item is also known as its *hash attribute*. The term derives from the service's use of an internal hash function to evenly distribute data items across partitions, based on their partition key values.
The sort key of an item is also known as its *range attribute*. The term derives from the way DynamoDB stores items with the same partition key physically close together, in sorted order by the sort key value.

Secondary Indexes

In DynamoDB, you can read data in a table by providing primary key attribute values. If you want to read the data using non-key attributes, you can use a secondary index to do this. After you create a secondary index on a table, you can read data from the index in much the same way as you do from the table. By using secondary indexes, your applications can use many different query patterns, in addition to accessing the data by primary key values.

For more information, see Amazon DynamoDB: How It Works in the *Amazon DynamoDB Developer Guide*:

Running DynamoDB on Your Computer

In addition to the Amazon DynamoDB web service, AWS provides a downloadable version of DynamoDB that you can run locally. This lets you write applications without accessing the Amazon DynamoDB web service. Instead, the database is self-contained on your computer.

This local version of DynamoDB can help you save on provisioned throughput, data storage, and data transfer fees. In addition, you do not need to have an Internet connection while you are developing your application.

When you are ready to deploy your application in production, you can make some minor changes to your code so that it uses the Amazon DynamoDB web service. In the summary section of each language-specific tutorial we explain how to do this.

Topics

DynamoDB (Downloadable Version) vs. DynamoDB Web Service

The downloadable version of DynamoDB is for development and testing purposes only. By comparison, DynamoDB is a managed service with scalability, availability, and durability features that make it ideal for production use. The following table lists other key differences between DynamoDB running on your computer and the Amazon DynamoDB service:

	DynamoDB (downloadable version)	**Amazon DynamoDB (web service)**
Creating a Table	The table is created immediately.	Table creation takes some time, depending on its provisioned throughput settings. DynamoDB allocates sufficient resources to meet your specific read and write capacity requirements.
Provisioned Throughput	The downloadable version of DynamoDB ignores provisioned throughput settings.	Provisioned throughput is a fundamental concept in DynamoDB. The rate at which you can read and write data depends on your provisioned capacity settings. For more information, see Provisioned Throughput in the *Amazon DynamoDB Developer Guide*.
Reading and Writing Data	Reads and writes are performed as fast as possible, without any network overhead.	Read and write activity is regulated by the provisioned throughput settings on the table. To increase the maximum throughput, you must increase the throughput settings on the table. Network latency also affects throughput to an extent.

	DynamoDB (downloadable version)	Amazon DynamoDB (web service)
Deleting a Table	The table is deleted immediately.	Table deletion takes some time, as DynamoDB releases the resources that had been used by the table.

For more information, see Differences Between DynamoDB Running Locally and the Amazon DynamoDB Web Service in the *Amazon DynamoDB Developer Guide*.

Download and Run DynamoDB

DynamoDB is available as an executable `.jar` file. It runs on Windows, Linux, Mac OS X, and other platforms that support Java. Follow these steps to download and run DynamoDB on your computer.

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

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

   DynamoDB is also available as part of the AWS Toolkit for Eclipse or Maven. For more information, see AWS Toolkit For Eclipse and DynamoDB Local (Downloadable Version) and Maven, respectively.

   Important
   DynamoDB on your computer requires the Java Runtime Environment (JRE) version 6.x or newer; it will not run on older JRE versions.

2. Extract the contents of the downloaded archive and copy the extracted directory to a location of your choice.
3. To start DynamoDB, open a command prompt window, navigate to the directory where you extracted `DynamoDBLocal.jar`, and enter the following command:

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

   Note
   DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. For a complete list of DynamoDB runtime options, including `-port` , type this command:
   `java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -help`
   For more information, see Command Line Options in the *Amazon DynamoDB Developer Guide*.

4. You can now start to write applications.

Next Step

Work through one of the following tutorials:

- Java and DynamoDB (p. 6)
- JavaScript and DynamoDB (p. 24)
- Node.js and DynamoDB (p. 45)
- .NET and DynamoDB (p. 61)
- PHP and DynamoDB (p. 91)
- Python and DynamoDB (p. 110)
- Ruby and DynamoDB (p. 126)

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

Java and DynamoDB

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

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

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 22), we explain how to run the same code against the DynamoDB web service.

Cost: Free

Prerequisites

- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB on your computer. For more information, see Download and Run DynamoDB (p. 4). DynamoDB (Downloadable Version) is also available as part of the AWS Toolkit for Eclipse. For more information, see AWS Toolkit For Eclipse.
- Sign up for Amazon Web Services and create access keys. You need these credentials to use AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.
- Setup the AWS SDK for Java:
  - Install a Java development environment. If you are using Eclipse IDE, install the AWS Toolkit for Eclipse.
  - Install the AWS SDK for Java.
  - Setup your AWS security credentials for use with the SDK for Java.

For instructions, see Getting Started in the *AWS SDK for Java Developer Guide*.

Note
As you work through this tutorial, you can refer to the AWS SDK for Java API Reference.

Step 1: Create a Table

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

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

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

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

package com.amazonaws.codesamples.gsg;

import java.util.Arrays;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;

public class MoviesCreateTable {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

        String tableName = "Movies";

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

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

    }
}
```

Note

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

2. Compile and run the program.

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

    ...

]
```

In the JSON data, note the following:

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

The following is an example of movie data:

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

Step 2.1: Download the Sample Data File

1. Download the sample data archive by clicking this link: moviedata.zip
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file to your current directory.

Step 2.2: Load the Sample Data Into the Movies Table

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

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

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

package com.amazonaws.codesamples.gsg;

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

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class MoviesLoadData {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

        ObjectNode currentNode;

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

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

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

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

This program uses the open source Jackson library to process JSON. Jackson is included in the AWS SDK for Java. You do not need to install it separately.

2. Compile and run the program.

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

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

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

package com.amazonaws.codesamples.gsg;

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

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.PutItemOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class MoviesItemOps01 {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

        try {
            System.out.println("Adding a new item...");
            PutItemOutcome outcome = table
                .putItem(new Item().withPrimaryKey("year", year, "title",
 title).withMap("info", infoMap));

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

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

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

2. Compile and run the program.

Step 3.2: Read an Item

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

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

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

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

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

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.GetItemSpec;

public class MoviesItemOps02 {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

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

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

    }
}
```

2. Compile and run the program.

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

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

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

import java.util.Arrays;

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

public class MoviesItemOps03 {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

        try {
            System.out.println("Updating the item...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
 outcome.getItem().toJSONPretty());

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

Note
This program uses an `UpdateExpression` to describe all updates you want to perform on the specified item.
The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. Compile and run the program.

Step 3.4: Increment an Atomic Counter

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

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

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

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

package com.amazonaws.codesamples.gsg;

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

public class MoviesItemOps04 {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

        try {
            System.out.println("Incrementing an atomic counter...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
 outcome.getItem().toJSONPretty());

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

2. Compile and run the program.

Step 3.5: Update an Item (Conditionally)

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

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

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

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

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class MoviesItemOps05 {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey(new PrimaryKey("year", year, "title",
 title)).withUpdateExpression("remove info.actors[0]")
            .withConditionExpression("size(info.actors) > :num").withValueMap(new
 ValueMap().withNumber(":num", 3))
            .withReturnValues(ReturnValue.UPDATED_NEW);

        // Conditional update (we expect this to fail)
        try {
            System.out.println("Attempting a conditional update...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
 outcome.getItem().toJSONPretty());

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

2. Compile and run the program.

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

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

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

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

   The condition is now *greater than or equal to 3* instead of *greater than 3*.
4. Compile and run the program. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item

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

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

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

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

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesItemOps06 {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

        DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
            .withPrimaryKey(new PrimaryKey("year", year, "title",
 title)).withConditionExpression("info.rating <= :val")
            .withValueMap(new ValueMap().withNumber(":val", 5.0));

        // Conditional delete (we expect this to fail)

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

2. Compile and run the program.

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the rating for this particular move is greater than 5.
3. Modify the program to remove the condition in `DeleteItemSpec`.

   ```
   DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
   .withPrimaryKey(new PrimaryKey("year", 2015, "title", "The Big New Movie"));
   ```
4. Compile and run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

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

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

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

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

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query

The code included in this step performs the following queries:

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

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

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

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

import java.util.HashMap;
import java.util.Iterator;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;

public class MoviesQuery {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

        HashMap<String, String> nameMap = new HashMap<String, String>();
        nameMap.put("#yr", "year");

        HashMap<String, Object> valueMap = new HashMap<String, Object>();
        valueMap.put(":yyyy", 1985);

        QuerySpec querySpec = new QuerySpec().withKeyConditionExpression("#yr
 = :yyyy").withNameMap(nameMap)
            .withValueMap(valueMap);

        ItemCollection<QueryOutcome> items = null;
        Iterator<Item> iterator = null;
        Item item = null;

        try {
            System.out.println("Movies from 1985");
            items = table.query(querySpec);

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

        }
        catch (Exception e) {
            System.err.println("Unable to query movies from 1985");
            System.err.println(e.getMessage());
        }

        valueMap.put(":yyyy", 1992);
        valueMap.put(":letter1", "A");
        valueMap.put(":letter2", "L");

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

```
            .withValueMap(valueMap);

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

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

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

Note

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

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

2. Compile and run the program.

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 4.2: Scan

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

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

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

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

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

```
import java.util.Iterator;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.ScanSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesScan {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

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

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

In the code, note the following:

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

2. Compile and run the program.

Note
You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

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

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

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;

public class MoviesDeleteTable {

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

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

        DynamoDB dynamoDB = new DynamoDB(client);

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

        try {
            System.out.println("Attempting to delete table; please wait...");
            table.delete();
            table.waitForDelete();
            System.out.print("Success.");

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

2. Compile and run the program.

Summary

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

Using the Amazon DynamoDB Service

You need to change the endpoint in your application in order to use the Amazon DynamoDB service. To do this, remove the following import:

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

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

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

Now modify the client so that it will access an AWS region instead of a specific endpoint:

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

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

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

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

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

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

JavaScript and DynamoDB

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

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 43), we explain how to run the same code against the DynamoDB web service.

Cost: Free

Prerequisites

- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB on your computer. For more information, see Download and Run DynamoDB (p. 4).
- Set up the AWS SDK for JavaScript. To do this, add or modify the following script tag to your HTML pages:

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

  Note
  The version of AWS SDK for JavaScript might have been updated. For the latest version, see the AWS SDK for JavaScript API Reference.
- Enable CORS (Cross-Origin Resource Sharing) so communication between your computer's browser and the downloadable version of DynamoDB can occur.

  To do this and run the tutorial on your computer:

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

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

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

As you work through this tutorial, you can refer to the AWS SDK for JavaScript API Reference.

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

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

Step 1: Create a Table

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

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

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

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

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

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

var dynamodb = new AWS.DynamoDB();

function createMovies() {
    var params = {
        TableName : "Movies",
        KeySchema: [
            { AttributeName: "year", KeyType: "HASH"},
            { AttributeName: "title", KeyType: "RANGE" }
        ],
        AttributeDefinitions: [
            { AttributeName: "year", AttributeType: "N" },
            { AttributeName: "title", AttributeType: "S" }
        ],
        ProvisionedThroughput: {
            ReadCapacityUnits: 5,
            WriteCapacityUnits: 5
        }
    };

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

</script>
</head>

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

</body>
</html>
```

Note

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

2. Open the `MoviesCreateTable.html` file on your browser.
3. Choose **Create Table**.

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

    ...

]
```

In the JSON data, note the following:

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

The following is an example of movie data:

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

Step 2.1: Download the Sample Data File

1. Click moviedata.zip to download the sample data archive.
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file to your current directory.

Step 2.2: Load the Sample Data into the Movies Table

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

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

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

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

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

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

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

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

</script>
</head>

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

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

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

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

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

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

<script>
AWS.config.update({
  region: "us-west-2",
  endpoint: 'http://localhost:8000',
  // accessKeyId default can be used while using the downloadable version of DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
 Cognito instead.
  accessKeyId: "fakeMyKeyId",
  // secretAccessKey default can be used while using the downloadable version of
 DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
 Cognito instead.
  secretAccessKey: "fakeSecretAccessKey"
});

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

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

</script>
</head>

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

</body>
</html>
```

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

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

Step 3.2: Read an Item

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

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

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

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

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

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

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

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

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

    var params = {
        TableName: table,
        Key:{
            "year": year,
            "title": title
        }
    };
    docClient.get(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to read item: " +
 "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "GetItem succeeded: " +
 "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

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

</body>
```

```
</html>
```

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

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

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

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

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

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

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

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

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

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

</script>
</head>

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

</body>
</html>
```

Note

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

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

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

Step 3.4: Increment an Atomic Counter

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

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

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

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

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

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

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

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

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

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

</script>
</head>

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

</body>
</html>
```

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

Step 3.5: Update an Item (Conditionally)

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

In this case, the item is updatedonly if there are more than three actors.

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

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

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

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

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

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

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

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

</script>
</head>

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

</body>
</html>
```

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

   The program should fail with the following message:

   ```
   The conditional update failed
   ```

   This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.
4. Modify the program so that the `ConditionExpression` looks like this:

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

   The condition is now *greater than or equal to 3* instead of *greater than 3*.
5. Run the program again. The `updateItem` operation should now succeed.

Step 3.6: Delete an Item

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

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

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

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

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

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

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

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

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

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

</script>
</head>

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

</body>
</html>
```

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

   The program should fail with the following message:

   ```
   The conditional delete failed
   ```

   This is because the rating for this particular movie is greater than 5.
4. Modify the program to remove the condition from `params`.

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

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

Step 4: Query and Scan the Data

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

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

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

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

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query - All Movies Released in a Year

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

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

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

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

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

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

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

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

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

</script>
</head>

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

</body>
</html>
```

Note

`ExpressionAttributeNames` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you cannot use it directly in any expression, including `KeyConditionExpression`. For this reason, we use the expression attribute name `#yr`.

`ExpressionAttributeValues` provides value substitution. We use this because you cannot use literals in any expression, including `KeyConditionExpression`. For this reason, we use the expression attribute value `:yyyy`.

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

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

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

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

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

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

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

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

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

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

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

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

</script>
</head>

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

</body>
</html>
```

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

Step 4.3: Scan

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

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

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

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

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

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

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

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

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

    docClient.scan(params, onScan);

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

```
            });

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

</script>
</head>

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

</body>
</html>
```

In the code, note the following:

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

2. Open the `MoviesScan.html` file on your browser.
3. Choose **Scan**.

Note
You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional): Delete the Table

To delete the `Movies` table:

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

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

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

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

```
var dynamodb = new AWS.DynamoDB();

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

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

</script>
</head>

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

</body>
</html>
```

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

Summary

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

To modify your code so that it uses the Amazon DynamoDB service, do the following:

1. Sign up for Amazon Web Services and create access keys. You need these credentials to use the AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.
2. Update the AWS Configuration Region (p. 43).
3. Install and Configure the AWS CLI (p. 44).
4. Configure AWS Credentials in Your Files Using Amazon Cognito (p. 44).

Update the AWS Configuration Region

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

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

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

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

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

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

Install and Configure the AWS CLI

After you have obtained your AWS access key ID and secret key, you can set up the AWS CLI on your computer.

To install and configure the AWS CLI

1. Go to the AWS Command Line Interface User Guide.
2. Follow the instructions for Installing the AWS CLI and Configuring the AWS CLI.

Configure AWS Credentials in Your Files Using Amazon Cognito

The recommended way to obtain AWS credentials for your web and mobile applications is to use Amazon Cognito. Amazon Cognito helps you avoid hardcoding your AWS credentials on your files. Amazon Cognito uses IAM roles to generate temporary credentials for your application's authenticated and unauthenticated users.

For more information, see Configure AWS Credentials in Your Files Using Amazon Cognito.

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

Node.js and DynamoDB

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

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 59), we explain how to run the same code against the DynamoDB web service.

Cost: Free

Prerequisites

- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB on your computer. For more information, see Download and Run DynamoDB (p. 4).
- Sign up for Amazon Web Services and create access keys. You need these credentials to use AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.
- Create an AWS credentials file to store your access keys. For more information, see Loading Credentials from the Shared Credentials File in the *AWS SDK for JavaScript Developer Guide*.
- Set up the AWS SDK for JavaScript:
  - Go to http://nodejs.org and install Node.js.
  - Go to https://aws.amazon.com/sdk-for-node-js and install the AWS SDK for JavaScript.

  For more information, see the AWS SDK for JavaScript Getting Started Guide.

> **Note**
> As you work through this tutorial, you can refer to the AWS SDK for JavaScript API Reference.

Step 1: Create a Table

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

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

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

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

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

var dynamodb = new AWS.DynamoDB();

var params = {
    TableName : "Movies",
    KeySchema: [
        { AttributeName: "year", KeyType: "HASH"},  //Partition key
        { AttributeName: "title", KeyType: "RANGE" }  //Sort key
    ],
    AttributeDefinitions: [
        { AttributeName: "year", AttributeType: "N" },
        { AttributeName: "title", AttributeType: "S" }
    ],
    ProvisionedThroughput: {
        ReadCapacityUnits: 10,
        WriteCapacityUnits: 10
    }
};

dynamodb.createTable(params, function(err, data) {
    if (err) {
        console.error("Unable to create table. Error JSON:", JSON.stringify(err, null,
 2));
    } else {
        console.log("Created table. Table description JSON:", JSON.stringify(data,
 null, 2));
    }
});
```

Note

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

2. Type the following command to run the program:

```
node MoviesCreateTable.js
```

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

    ...

]
```

In the JSON data, note the following:

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

The following is an example of movie data:

```

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

Step 2.1: Download the Sample Data File

1. Download the sample data archive by clicking this link: moviedata.zip
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file to your current directory.

Step 2.2: Load the Sample Data Into the Movies Table

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

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

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

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

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

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

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

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

2. Type the following command to run the program:

```
node MoviesLoadData.js
```

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

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

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

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

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

var table = "Movies";

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

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

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

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

2. Type the following command to run the program:

```
node MoviesItemOps01.js
```

Step 3.2: Read an Item

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

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

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

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

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

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

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

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

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

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

2. Type the following command to run the program:

```
node MoviesItemOps02.js
```

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

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

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

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

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

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

// Update the item, unconditionally,

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

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

Note
This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.
The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`"UPDATED_NEW"`).

2. Type the following command to run the program:

```
node MoviesItemOps03.js
```

Step 3.4: Increment an Atomic Counter

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

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

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

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

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

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

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

// Increment an atomic counter

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

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

2. Type the following command to run the program:

```
node MoviesItemOps04.js
```

Step 3.5: Update an Item (Conditionally)

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

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

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

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

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

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

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

// Conditional update (will fail)

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

console.log("Attempting a conditional update...");
```

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

2. Type the following command to run the program:

   ```
   node MoviesItemOps05.js
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.
3. Modify the program so that the `ConditionExpression` looks like this:

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

   The condition is now *greater than or equal to 3* instead of *greater than 3*.
4. Run the program again. The `updateItem` operation should now succeed.

Step 3.6: Delete an Item

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

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

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

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

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

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

   var table = "Movies";

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

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

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

2. Type the following command to run the program:

   ```
   node MoviesItemOps06.js
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the rating for this particular movie is greater than 5.
3. Modify the program to remove the condition from `params`.

   ```
   var params = {
       TableName:table,
       Key:{
           "title":title,
           "year":year
       }
   };
   ```
4. Run the program again. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

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

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

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

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

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query - All Movies Released in a Year

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

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

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

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

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

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

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

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

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

2. Type the following command to run the program:

```
node MoviesQuery01.js
```

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

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

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

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

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

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

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

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

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

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

2. Type the following command to run the program:

```
node MoviesQuery02.js
```

Step 4.3: Scan

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

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

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

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

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

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

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

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

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

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

   In the code, note the following:

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

2. Type the following command to run the program:

```
node MoviesScan.js
```

Note
You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional): Delete the Table

To delete the `Movies` table:

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

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

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

var dynamodb = new AWS.DynamoDB();

var params = {
    TableName : "Movies"
};

dynamodb.deleteTable(params, function(err, data) {
    if (err) {
        console.error("Unable to delete table. Error JSON:", JSON.stringify(err, null,
 2));
    } else {
        console.log("Deleted table. Table description JSON:", JSON.stringify(data,
 null, 2));
    }
});
```

2. Type the following command to run the program:

```
node MoviesDeleteTable.js
```

Summary

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

Using the Amazon DynamoDB Service

You need to change the endpoint in your application in order to use the Amazon DynamoDB service. To do this, modify the code as follows:

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

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

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

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

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

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

.NET and DynamoDB

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

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

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 89), we explain how to run the same code against the DynamoDB service in the cloud.

Cost: Free

Prerequisites

- Use a computer running a recent version of Microsoft Windows and a current version of Microsoft Visual Studio. If you don't already have Visual Studio installed, you can download a free copy of the Community edition from the Visual Studio website.
- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB Local. For more information, see Running DynamoDB on Your Computer.
- Sign up for Amazon Web Services and create access keys. You need these credentials to use AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.

- Set up a security profile for DynamoDB in Visual Studio. See the step-by-step instructions for doing this in Creating Example Tables and Uploading Data Using the AWS SDK for .NET.
- In Visual Studio, create a new project called `DynamoDB_intro` using the **Console Application** template in the **Installed/Templates/Visual C#/** node. This is the project you use throughout this Getting Started tutorial.

  Note
  The following tutorial does not work with .NET core as it does not support synchronous methods. For more information, see AWS Asynchronous APIs for .NET.
- Install the NuGet package for the DynamoDB module of the AWS SDK for .NET, version 3 into you new `DynamoDB_intro` project. To do this, open the NuGet Package Manager Console from the Tools menu in Visual Studio and type the following command at the `PM>` prompt:

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

Step 1: Create a Table

The document model in the AWS SDK for .NET does not provide for creating tables, so you have to use the low-level APIs (see Working with Tables Using the AWS SDK for .NET Low-Level API).

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

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

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

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

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

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

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

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

            // Using the DynamoDB client, make a synchronous CreateTable request
            CreateTableResponse createResponse;
            try
            {
                createResponse = client.CreateTable(createRequest);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to create the new table; " +
ex.Message);
                PauseForDebugWindow();
                return;
            }

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

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

```
}
```

Note

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

2. Compile and run the program.

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

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

Step 2.1: Download the Sample Data File

1. Download the sample data archive by clicking this link: moviedata.zip
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the`moviedata.json` file to the `bin/Debug` folder of your `DynamoDB_intro` Visual Studio project.

Step 2.2: Load the Sample Data Into the Movies Table

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

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

   Load the `Json.NET` library into your project by opening the NuGet Package Manager Console from the Tools menu in Visual Studio and typing the following command at the `PM>` prompt:

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

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

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

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

   using Newtonsoft;
   using Newtonsoft.Json;
   using Newtonsoft.Json.Linq;

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

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

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

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

            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject("Movies");
            if (table == null)
            {
                PauseForDebugWindow();
                return;
            }

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

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

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

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

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

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

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

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

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

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

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

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

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

Note
The primary key is required. In this table, the primary key is a composite of both a partition-key attribute (`year`) and a sort-key attribute (`title`).
This code writes an item to the table that has both of the two primary key attributes (`year` + `title`), and a complex `info` attribute that stores more information about the movie.

2. Compile and run the program.

Step 3.2: Read an Item

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

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

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

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

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

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

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

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

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

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

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

2. Compile and run the program.

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

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

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

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

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

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

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

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

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

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

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

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

Note
Because the Document Model in the AWS SDK for .NET does not support updating nested attributes, we need to use the `AmazonDynamoDBClient.UpdateItem` API instead of `Table.UpdateItem` to update attributes under the top-level `info` attribute.
To do this, we create an `UpdateItemRequest` that specifies the item we want to update and the new values we want to set.

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

2. Compile and run the program.

Step 3.4: Increment an Atomic Counter

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

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

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

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

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

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

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

            // Create an UpdateItemRequest to modify two existing nested attributes
            // and add a new one
            UpdateItemRequest updateRequest = new UpdateItemRequest()
            {
                TableName = "Movies",
                Key = new Dictionary<string, AttributeValue>
            {
                { "year",  new AttributeValue {
                      N = "2015"
                  } },
                { "title", new AttributeValue {
                      S = "The Big New Movie"
                  }}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>
            {
                { ":inc", new AttributeValue {
                      N = "1"
                  } }
            },
                UpdateExpression = "SET info.rating = info.rating + :inc",
                ReturnValues = "UPDATED_NEW"
            };
            // Use AmazonDynamoDBClient.UpdateItem to update the specified attributes
            UpdateItemResponse uir = null;
            try
            {
                uir = client.UpdateItem(updateRequest);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\nError: UpdateItem failed, because " +
ex.Message);
                if (uir != null)
                    Console.WriteLine("    Status code was: " +
uir.HttpStatusCode.ToString());
                PauseForDebugWindow();
                return;
            }

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

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

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

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

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

2. Compile and run the program.

Step 3.5: Update an Item (Conditionally)

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

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

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

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

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

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

            // Create an UpdateItemRequest to modify two existing nested attributes
            // and add a new one
            UpdateItemRequest updateRequest = new UpdateItemRequest()
            {
                TableName = "Movies",
                Key = new Dictionary<string, AttributeValue>
            {
                { "year",  new AttributeValue {
                      N = "2015"
                  } },
                { "title", new AttributeValue {
                      S = "The Big New Movie"
                  } }
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>
            {
                { ":n", new AttributeValue {
                      N = "3"
                  } }
            },
                ConditionExpression = "size(info.actors) > :n",
                UpdateExpression = "REMOVE info.actors",
                ReturnValues = "UPDATED_NEW"
            };

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

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

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

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

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

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

2. Compile and run the program.

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.
3. Modify the program so that the number of actors that the `ConditionExpression` uses is 2 instead of 3:

   ```
   { ":n", new AttributeValue { N = "2" } }
   ```

   The condition now specifies that the number of actors must be greater than 2.
4. When you compile and run the program now, the `UpdateItem` operation should succeed.

Step 3.6: Delete an Item

You can use the `Table.DeleteItem` operation to delete an item by specifying its primary key. You can optionally provide a condition in the `DeleteItemOperationConfig` parameter, to prevent item deletion if the condition is not met.

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

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

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

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

namespace DynamoDB_intro
{
    class Program
    {
```

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

            // Create the condition
            DeleteItemOperationConfig opConfig = new DeleteItemOperationConfig();
            lopConfig.ConditionalExpression = new Expression();
            opConfig.ConditionalExpression.ExpressionAttributeValues[":val"] = "5.0";
            opConfig.ConditionalExpression.ExpressionStatement = "info.rating
<= :val";

            // Delete this item
            try
            {
                table.DeleteItem(2015, "The Big New Movie", opConfig);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: Could not delete the movie item with
year={0}, title=\"{1}\"\n   Reason: {2}.",
                           2015, "The Big New Movie", ex.Message);
            }

            // Try to retrieve it, to see if it has been deleted
            Document document = table.GetItem(2015, "The Big New Movie");
            if (document == null)
                Console.WriteLine("\n The movie item with year={0}, title=\"{1}\" has
been deleted.",
                           2015, "The Big New Movie");
            else
                Console.WriteLine("\nRead back the item: \n" +
document.ToJsonPretty());

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

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

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

2. Compile and run the program.

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the rating for this particular move is greater than 5.
3. Modify the program to remove the `DeleteItemOperationConfig` named `opConfig` from the call to `table.DeleteItem`:

   ```
   try { table.DeleteItem( 2015, "The Big New Movie" ); }
   ```
4. Compile and run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

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

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

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

To find all movies released during a year, you need to specify only the `year` partition-key attribute. You can add the `title` sort-key attribute to retrieve a subset of movies based on some condition (on the sort-key attribute), such as finding movies released in 2014 that have a title starting with the letter "A".

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query

The C# code included in this step performs the following queries:

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

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

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

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

namespace DynamoDB_intro
{
    class Program
    {
        static string commaSep = ", ";
        static string movieFormatString = "    \"{0}\", lead actor: {1}, genres: {2}";

        static void Main(string[] args)
        {
            // Get an AmazonDynamoDBClient for the local DynamoDB database
            AmazonDynamoDBClient client = GetLocalClient();

            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject(client, "Movies");
            if (table == null)
            {
                PauseForDebugWindow();
                return;
            }

            /*-----------------------------------------------------------------------
             *  4.1.1:  Call Table.Query to initiate a query for all movies with
             *          year == 1985, using an empty filter expression.
             *-----------------------------------------------------------------------
*/
            Search search;
            try
            {
                search = table.Query(1985, new Expression());
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: 1985 query failed because: " +
 ex.Message);
                PauseForDebugWindow();
                return;
            }

            // Display the titles of the movies returned by this query
            List<Document> docList = new List<Document>();
            Console.WriteLine("\n All movies released in 1985:" +
                       "\n-----------------------------------------------");
            do
            {
                try { docList = search.GetNextSet(); }
                catch (Exception ex)
                {
                    Console.WriteLine("\n Error: Search.GetNextStep failed because: " +
 ex.Message);
                    break;
                }
                foreach (var doc in docList)
```

```
                    Console.WriteLine("    " + doc["title"]);
            } while (!search.IsDone);


            /*-----------------------------------------------------------------------
             *  4.1.2a:  Call Table.Query to initiate a query for all movies where
             *           year equals 1992 AND title is between "B" and "Hzzz",
             *           returning the lead actor and genres of each.
             *-----------------------------------------------------------------------
*/
            Primitive y_1992 = new Primitive("1992", true);
            QueryOperationConfig config = new QueryOperationConfig();
            config.Filter = new QueryFilter();
            config.Filter.AddCondition("year", QueryOperator.Equal, new DynamoDBEntry[]
 { 1992 });
            config.Filter.AddCondition("title", QueryOperator.Between, new
 DynamoDBEntry[] { "B", "Hzz" });
            config.AttributesToGet = new List<string> { "title", "info" };
            config.Select = SelectValues.SpecificAttributes;

            try
            {
                search = table.Query(config);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: 1992 query failed because: " +
 ex.Message);
                PauseForDebugWindow();
                return;
            }

            // Display the movie information returned by this query
            Console.WriteLine("\n\n Movies released in 1992 with titles between \"B\"
 and \"Hzz\" (Document Model):" +

 "\n-----------------------------------------------------------------------------");
            docList = new List<Document>();
            Document infoDoc;
            do
            {
                try
                {
                    docList = search.GetNextSet();
                }
                catch (Exception ex)
                {
                    Console.WriteLine("\n Error: Search.GetNextStep failed because: " +
 ex.Message);
                    break;
                }
                foreach (var doc in docList)
                {
                    infoDoc = doc["info"].AsDocument();
                    Console.WriteLine(movieFormatString,
                               doc["title"],
                               infoDoc["actors"].AsArrayOfString()[0],
                               string.Join(commaSep,
 infoDoc["genres"].AsArrayOfString()));
                }
            } while (!search.IsDone);


            /*-----------------------------------------------------------------------
             *  4.1.2b:  Call AmazonDynamoDBClient.Query to initiate a query for all
             *           movies where year equals 1992 AND title is between M and Tzz,
             *           returning the genres and the lead actor of each.
             *-----------------------------------------------------------------------
*/
            QueryRequest qRequest = new QueryRequest
            {
                TableName = "Movies",
                ExpressionAttributeNames = new Dictionary<string, string>
            {
                { "#yr", "year" }
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>
            {
                { ":y_1992",  new AttributeValue {
                      N = "1992"
                  } },
                { ":M",       new AttributeValue {
                      S = "M"
                  } },
                { ":Tzz",     new AttributeValue {
                      S = "Tzz"
                  } }
            },
                KeyConditionExpression = "#yr = :y_1992 and title between :M
 and :Tzz",
                ProjectionExpression = "title, info.actors[0], info.genres"
            };

            QueryResponse qResponse;
            try
            {
                qResponse = client.Query(qRequest);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: Low-level query failed, because: " +
 ex.Message);
                PauseForDebugWindow();
                return;
            }

            // Display the movie information returned by this query
            Console.WriteLine("\n\n Movies released in 1992 with titles between \"M\"
 and \"Tzz\" (low-level):" +

 "\n-----------------------------------------------------------------------");
            foreach (Dictionary<string, AttributeValue> item in qResponse.Items)
            {
                Dictionary<string, AttributeValue> info = item["info"].M;
                Console.WriteLine(movieFormatString,
                           item["title"].S,
                           info["actors"].L[0].S,
                           GetDdbListAsString(info["genres"].L));
            }
        }

        public static string GetDdbListAsString(List<AttributeValue> strList)
        {
            StringBuilder sb = new StringBuilder();
            string str = null;
            AttributeValue av;
            for (int i = 0; i < strList.Count; i++)
            {
                av = strList[i];
                if (av.S != null)
                    str = av.S;
                else if (av.N != null)
```

```
                    str = av.N;
                else if (av.SS != null)
                    str = string.Join(commaSep, av.SS.ToArray());
                else if (av.NS != null)
                    str = string.Join(commaSep, av.NS.ToArray());
                if (str != null)
                {
                    if (i > 0)
                        sb.Append(commaSep);
                    sb.Append(str);
                }
            }
            return (sb.ToString());
        }

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


        public static Table GetTableObject(AmazonDynamoDBClient client, string
 tableName)
        {
            Table table = null;
            try
            {
                table = Table.LoadTable(client, tableName);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
 ex.Message);
                return (null);
            }
            return (table);
        }

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

Note

- In the first query, for all movies released in 1985, an empty expression indicates that no filtering on the sort-key part of the primary key is desired.
- In the second query, which uses the AWS SDK for .NET Document Model to query for all movies released in 1992 with titles starting with the letters A through L, we can only query for top-level attributes, and so must retrieve the entire `info` attribute. Our display code then accesses the nested attributes we're interested in.
- In the third query, we use the low-level AWS SDK for .NET API, which gives more control over what is returned. Here, we are able to retrieve only those nested attributes within the `info` attribute that we are interested in, namely `info.genres` and `info.actors[0]`.

2. Compile and run the program.

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can also optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 4.2: Scan

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

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

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

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

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

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

            // Get a Table object for the table that you created in Step 1
            Table table = GetTableObject(client, "Movies");
            if (table == null)
            {
                PauseForDebugWindow();
                return;
            }


            /*--------------------------------------------------------------------------
             *  4.2a:  Call Table.Scan to return the movies released in the 1950's,
             *         displaying title, year, lead actor and lead director.
             *--------------------------------------------------------------------------
*/
            ScanFilter filter = new ScanFilter();
            filter.AddCondition("year", ScanOperator.Between, new DynamoDBEntry[]
 { 1950, 1959 });
            ScanOperationConfig config = new ScanOperationConfig
            {
                AttributesToGet = new List<string> { "year, title, info" },
                Filter = filter
            };
            Search search = table.Scan(filter);

            // Display the movie information returned by this query
            Console.WriteLine("\n\n Movies released in the 1950's (Document Model):" +
                       "\n--------------------------------------------------");
            List<Document> docList = new List<Document>();
            Document infoDoc;
            string movieFormatString = "    \"{0}\" ({1})-- lead actor: {2}, lead
 director: {3}";
            do
            {
                try
                {
                    docList = search.GetNextSet();
                }
                catch (Exception ex)
                {
                    Console.WriteLine("\n Error: Search.GetNextStep failed because: " +
 ex.Message);
                    break;
                }
                foreach (var doc in docList)
                {
                    infoDoc = doc["info"].AsDocument();
                    Console.WriteLine(movieFormatString,
                               doc["title"],
                               doc["year"],
                               infoDoc["actors"].AsArrayOfString()[0],
                               infoDoc["directors"].AsArrayOfString()[0]);
                }
            } while (!search.IsDone);


            /*--------------------------------------------------------------------------
             *  4.2b:  Call AmazonDynamoDBClient.Scan to return all movies released
             *         in the 1960's, only downloading the title, year, lead
             *         actor and lead director attributes.
             *--------------------------------------------------------------------------
*/
            ScanRequest sRequest = new ScanRequest
            {
                TableName = "Movies",
                ExpressionAttributeNames = new Dictionary<string, string>
            {
                { "#yr", "year" }
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>
            {
                { ":y_a", new AttributeValue {
                      N = "1960"
```

```
                  } },
                { ":y_z", new AttributeValue {
                      N = "1969"
                  } },
            },
                FilterExpression = "#yr between :y_a and :y_z",
                ProjectionExpression = "#yr, title, info.actors[0], info.directors[0]"
            };

            ScanResponse sResponse;
            try
            {
                sResponse = client.Scan(sRequest);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: Low-level scan failed, because: " +
ex.Message);
                PauseForDebugWindow();
                return;
            }

            // Display the movie information returned by this scan
            Console.WriteLine("\n\n Movies released in the 1960's (low-level):" +
                       "\n-----------------------------------------");
            foreach (Dictionary<string, AttributeValue> item in sResponse.Items)
            {
                Dictionary<string, AttributeValue> info = item["info"].M;
                Console.WriteLine(movieFormatString,
                           item["title"].S,
                           item["year"].N,
                           info["actors"].L[0].S,
                           info["directors"].L[0].S);
            }
        }

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


        public static Table GetTableObject(AmazonDynamoDBClient client, string
tableName)
        {
            Table table = null;
            try
            {
                table = Table.LoadTable(client, tableName);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: failed to load the 'Movies' table; " +
 ex.Message);
                return (null);
            }
            return (table);
        }

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

In the code, note the following:

- The first scan uses the AWS SDK for .NET Document Model to scan the `Movies` table and return movies released in the 1950's. because the Document Model does not support nested attributes in the `AttributesToGet` field, we must download the entire `info` attribute to have access to the lead actor and director.
- The second scan uses the AWS SDK for .NET low-level API to scan the `Movies` table and return movies released in the 1960's. In this case, we can download only those attribute values in `info` that we are interested in, namely `info.actors[0]` and `info.directors[0]`.

2. Compile and run the program.

Note
You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

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

```
using System;
using System.Text;

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

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

            try
            {
                client.DeleteTable("Movies");
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n Error: the \'Movies\" table could not be deleted!
\n    Reason: " + ex.Message);
                Console.Write("\n\n ...Press any key to continue");
                Console.ReadKey();
                Console.WriteLine();
                return;
            }
            Console.WriteLine("\n Deleted the \'Movies\" table successfully!");
        }

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

2. Compile and run the program.

Summary

In this tutorial, you created the `Movies` table in DynamoDB on your computer and performed basic operations. DynamoDB Local is useful during application development and testing. However, when you are ready to run your application in a production environment, you need to modify your code so that it uses the Amazon DynamoDB service.

Using the Amazon DynamoDB Service

You need to change the endpoint in your application in order to use the Amazon DynamoDB service as follows. To do this, first remove the following line:

```
ddbConfig.ServiceURL = "http://localhost:8000";
```

Next, add a new line that specifies the AWS region you want to access:

```
ddbConfig.RegionEndpoint = RegionEndpoint.REGION;
```

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

```
ddbConfig.RegionEndpoint = RegionEndpoint.USWest2;
```

Instead of using DynamoDB Local, the program now uses the DynamoDB service endpoint in US West (Oregon).

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

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

PHP and DynamoDB

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

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 108), we explain how to run the same code against the DynamoDB service.

Cost: Free

Prerequisites

- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB on your computer. For more information, see Download and Run DynamoDB (p. 4).
- Sign up for Amazon Web Services and create access keys. You need these credentials to use AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.
- Create an AWS credentials file. For more information, see Using the AWS credentials file and credential profiles in the *AWS SDK for PHP Getting Started Guide*.
- Set up the AWS SDK for PHP:
  - Go to http://php.net and install PHP.
  - Go to https://aws.amazon.com/sdk-for-php and install the AWS SDK for PHP.

  For more information, see Getting Started in the *AWS SDK for PHP Getting Started Guide*.

> **Note**
> As you work through this tutorial, you can refer to the AWS SDK for PHP Developer Guide. The Amazon DynamoDB section in the *AWS SDK for PHP API Reference* describes the parameters and results for DynamoDB operations.

Step 1: Create a Table

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

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

1. Copy the following program into a file named `MoviesCreateTable.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();

$params = [
    'TableName' => 'Movies',
    'KeySchema' => [
        [
            'AttributeName' => 'year',
            'KeyType' => 'HASH'  //Partition key
        ],
        [
            'AttributeName' => 'title',
            'KeyType' => 'RANGE'  //Sort key
        ]
    ],
    'AttributeDefinitions' => [
        [
            'AttributeName' => 'year',
            'AttributeType' => 'N'
        ],
        [
            'AttributeName' => 'title',
            'AttributeType' => 'S'
        ],

    ],
    'ProvisionedThroughput' => [
        'ReadCapacityUnits' => 10,
        'WriteCapacityUnits' => 10
    ]
];

try {
    $result = $dynamodb->createTable($params);
    echo 'Created table.  Status: ' .
        $result['TableDescription']['TableStatus'] ."\n";

} catch (DynamoDbException $e) {
    echo "Unable to create table:\n";
    echo $e->getMessage() . "\n";
}

?>
```

Note

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

2. Type the following command to run the program:

```
php MoviesCreateTable.php
```

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

    ...

]
```

In the JSON data, note the following:

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

The following is an example of movie data:

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

Step 2.1: Download the Sample Data File

1. Download the sample data archive by clicking this link: moviedata.zip
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file to your current directory.

Step 2.2: Load the Sample Data Into the Movies Table

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

1. Copy the following program into a file named `MoviesLoadData.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';
```

```
$movies = json_decode(file_get_contents('moviedata.json'), true);

foreach ($movies as $movie) {

    $year = $movie['year'];
    $title = $movie['title'];
    $info = $movie['info'];

    $json = json_encode([
        'year' => $year,
        'title' => $title,
        'info' => $info
    ]);

    $params = [
        'TableName' => $tableName,
        'Item' => $marshaler->marshalJson($json)
    ];

    try {
        $result = $dynamodb->putItem($params);
        echo "Added movie: " . $movie['year'] . " " . $movie['title'] . "\n";
    } catch (DynamoDbException $e) {
        echo "Unable to add movie:\n";
        echo $e->getMessage() . "\n";
        break;
    }

}

?>
```

Note
The DynamoDB Marshaler class has methods for converting JSON documents and PHP arrays to the DynamoDB format. In this program, `$marshaler->marshalJson($json)` takes a JSON document and converts it into a DynamoDB item.

2. Type the following command to run the program:

```
php MoviesLoadData.php
```

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

1. Copy the following program into a file named `MoviesItemOps01.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$item = $marshaler->marshalJson('
    {
        "year": ' . $year . ',
        "title": "' . $title . '",
        "info": {
            "plot": "Nothing happens at all.",
            "rating": 0
        }
    }
');

$params = [
    'TableName' => 'Movies',
    'Item' => $item
];


try {
    $result = $dynamodb->putItem($params);
    echo "Added item: $year - $title\n";

} catch (DynamoDbException $e) {
    echo "Unable to add item:\n";
    echo $e->getMessage() . "\n";
}

?>
```

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

2. Type the following command to run the program:

```
php MoviesItemOps01.php
```

Step 3.2: Read an Item

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

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

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

1. Copy the following program into a file named `MoviesItemOps02.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson('
    {
        "year": ' . $year . ',
        "title": "' . $title . '"
    }
');

$params = [
    'TableName' => $tableName,
    'Key' => $key
];

try {
    $result = $dynamodb->getItem($params);
    print_r($result["Item"]);

} catch (DynamoDbException $e) {
    echo "Unable to get item:\n";
    echo $e->getMessage() . "\n";
}

?>
```

2. Type the following command to run the program:

   ```
   php MoviesItemOps02.php
   ```

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

1. Copy the following program into a file named `MoviesItemOps03.php`.

   ```
   <?php
   require 'vendor/autoload.php';

   date_default_timezone_set('UTC');

   use Aws\DynamoDb\Exception\DynamoDbException;
   use Aws\DynamoDb\Marshaler;

   $sdk = new Aws\Sdk([
       'endpoint'   => 'http://localhost:8000',
       'region'   => 'us-west-2',
       'version'  => 'latest'
   ]);

   $dynamodb = $sdk->createDynamoDb();
   ```

```
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson('
    {
        "year": ' . $year . ',
        "title": "' . $title . '"
    }
');


$eav = $marshaler->marshalJson('
    {
        ":r": 5.5 ,
        ":p": "Everything happens all at once.",
        ":a": [ "Larry", "Moe", "Curly" ]
    }
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' =>
        'set info.rating = :r, info.plot=:p, info.actors=:a',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];

try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item.\n";
    print_r($result['Attributes']);

} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}

?>
```

Note
This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.
The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. Type the following command to run the program:

```
php MoviesItemOps03.php
```

Step 3.4: Increment an Atomic Counter

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

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

1. Copy the following program into a file named `MoviesItemOps04.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson('
    {
        "year": ' . $year . ',
        "title": "' . $title . '"
    }
');

$eav = $marshaler->marshalJson('
    {
        ":val": 1
    }
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' => 'set info.rating = info.rating + :val',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];

try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item. ReturnValues are:\n";
    print_r($result['Attributes']);

} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}

?>
```

2. Type the following command to run the program:

```
php MoviesItemOps04.php
```

Step 3.5: Update an Item (Conditionally)

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

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

1. Copy the following program into a file named `MoviesItemOps05.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson('
    {
        "year": ' . $year . ',
        "title": "' . $title . '"
    }
');

$eav = $marshaler->marshalJson('
    {
        ":num": 3
    }
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' => 'remove info.actors[0]',
    'ConditionExpression' => 'size(info.actors) > :num',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];

try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item. ReturnValues are:\n";
    print_r($result['Attributes']);

} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}
```

```
?>
```

2. Type the following command to run the program:

   ```
   php MoviesItemOps05.php
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.
3. Modify the program so that the `ConditionExpression` looks like this:

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

   The condition is now *greater than or equal to 3* instead of *greater than 3*.
4. Run the program again. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item

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

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

1. Copy the following program into a file named `MoviesItemOps06.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson('
    {
        "year": ' . $year . ',
        "title": "' . $title . '"
    }
');

$eav = $marshaler->marshalJson('
    {
        ":val": 5
```

```
    }
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'ConditionExpression' => 'info.rating <= :val',
    'ExpressionAttributeValues'=> $eav
];

try {
    $result = $dynamodb->deleteItem($params);
    echo "Deleted item.\n";

} catch (DynamoDbException $e) {
    echo "Unable to delete item:\n";
    echo $e->getMessage() . "\n";
}

?>
```

2. Type the following command to run the program:

   ```
   php MoviesItemOps06.php
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the rating for this particular move is greater than 5.
3. Modify the program to remove the condition:

   ```
   $params = [
       'TableName' => $tableName,
       'Key' => $key
   ];
   ```

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

Step 4: Query and Scan the Data

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

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

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

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

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query - All Movies Released in a Year

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

1. Copy the following program into a file named `MoviesQuery01.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$eav = $marshaler->marshalJson('
    {
        ":yyyy": 1985
    }
');

$params = [
    'TableName' => $tableName,
    'KeyConditionExpression' => '#yr = :yyyy',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];

echo "Querying for movies from 1985.\n";

try {
    $result = $dynamodb->query($params);

    echo "Query succeeded.\n";

    foreach ($result['Items'] as $movie) {
        echo $marshaler->unmarshalValue($movie['year']) . ': ' .
            $marshaler->unmarshalValue($movie['title']) . "\n";
    }

} catch (DynamoDbException $e) {
    echo "Unable to query:\n";
    echo $e->getMessage() . "\n";
}

?>
```

Note

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

2. Type the following command to run the program:

```
php MoviesItemQuery01.php
```

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

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

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

1. Copy the following program into a file named `MoviesQuery02.php`:

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$eav = $marshaler->marshalJson('
    {
        ":yyyy":1992,
        ":letter1": "A",
        ":letter2": "L"
    }
');

$params = [
    'TableName' => $tableName,
    'ProjectionExpression' => '#yr, title, info.genres, info.actors[0]',
    'KeyConditionExpression' =>
```

```
        '#yr = :yyyy and title between :letter1 and :letter2',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];

echo "Querying for movies from 1992 - titles A-L, with genres and lead actor\n";

try {
    $result = $dynamodb->query($params);

    echo "Query succeeded.\n";

    foreach ($result['Items'] as $i) {
        $movie = $marshaler->unmarshalItem($i);
        print $movie['year'] . ': ' . $movie['title'] . ' ... ';

        foreach ($movie['info']['genres'] as $gen) {
            print $gen . ' ';
        }

        echo ' ... ' . $movie['info']['actors'][0] . "\n";
    }

} catch (DynamoDbException $e) {
    echo "Unable to query:\n";
    echo $e->getMessage() . "\n";
}

?>
```

2. Type the following command to run the program:

```
php MoviesQuery02.php
```

Step 4.3: Scan

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

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

1. Copy the following program into a file named `MoviesScan.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
```

```
$marshaler = new Marshaler();

//Expression attribute values
$eav = $marshaler->marshalJson('
    {
        ":start_yr": 1950,
        ":end_yr": 1959
    }
');

$params = [
    'TableName' => 'Movies',
    'ProjectionExpression' => '#yr, title, info.rating',
    'FilterExpression' => '#yr between :start_yr and :end_yr',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];

echo "Scanning Movies table.\n";

try {
    while (true) {
        $result = $dynamodb->scan($params);

        foreach ($result['Items'] as $i) {
            $movie = $marshaler->unmarshalItem($i);
            echo $movie['year'] . ': ' . $movie['title'];
            echo ' ... ' . $movie['info']['rating']
                . "\n";
        }

        if (isset($result['LastEvaluatedKey'])) {
            $params['ExclusiveStartKey'] = $result['LastEvaluatedKey'];
        } else {
            break;
        }
    }

} catch (DynamoDbException $e) {
    echo "Unable to scan:\n";
    echo $e->getMessage() . "\n";
}

?>
```

In the code, note the following:

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

2. Type the following command to run the program:

```
php MoviesScan.php
```

Note
You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy the following program into a file named `MoviesDeleteTable.php`.

```
<?php
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;

$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();

$params = [
    'TableName' => 'Movies'
];

try {
    $result = $dynamodb->deleteTable($params);
    echo "Deleted table.\n";

} catch (DynamoDbException $e) {
    echo "Unable to delete table:\n";
    echo $e->getMessage() . "\n";
}

?>
```

2. Type the following command to run the program:

```
php MoviesDeleteTable.php
```

Summary

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

Using the Amazon DynamoDB Service

You need to change the endpoint in your application in order to use the Amazon DynamoDB service. To do this, find the following lines in the code:

```
$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);
```

Now remove the `endpoint` parameter so that the code looks like this:

```
$sdk = new Aws\Sdk([
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);
```

After you remove this line, your code can access the DynamoDB service in the region specified by the `region` config value. For example, the following line specifies that you want to use the US West (Oregon) region:

```
'region'   => 'us-west-2',
```

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

Amazon DynamoDB is available in several regions worldwide. For the complete list, see Regions and Endpoints in the *AWS General Reference*. For more information about setting regions and endpoints in your code, see the boto: A Python interface to Amazon Web Services.

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

Python and DynamoDB

In this tutorial, you use the AWS SDK for Python (Boto 3) to write simple programs to perform the following Amazon DynamoDB operations:

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 124), we explain how to run the same code against the DynamoDB web service.

Cost: Free

Prerequisites

- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB on your computer. For more information, see Download and Run DynamoDB (p. 4).
- Sign up for Amazon Web Services and create access keys. You need these credentials to use AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.
- Create an AWS credentials file. For more information, see Configuration in the Boto 3 documentation.
- Install Python 2.6 or later. For more information, see https://www.python.org/downloads.

For instructions, see Quickstart in the Boto 3 documentation.

Note
As you work through this tutorial, you can refer to the AWS SDK for Python (Boto) documentation at http://boto.readthedocs.org/en/latest/. The following sections are specific to DynamoDB:

- DynamoDB tutorial

- DynamoDB low-level client

Step 1: Create a Table

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

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

1. Copy the following program into a file named `MoviesCreateTable.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")


table = dynamodb.create_table(
    TableName='Movies',
    KeySchema=[
        {
            'AttributeName': 'year',
            'KeyType': 'HASH'  #Partition key
        },
        {
            'AttributeName': 'title',
            'KeyType': 'RANGE'  #Sort key
        }
    ],
    AttributeDefinitions=[
        {
            'AttributeName': 'year',
            'AttributeType': 'N'
        },
        {
            'AttributeName': 'title',
            'AttributeType': 'S'
        },

    ],
    ProvisionedThroughput={
        'ReadCapacityUnits': 10,
        'WriteCapacityUnits': 10
    }
)

print("Table status:", table.table_status)
```

Note

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

- These examples use the Python 3 style `print` function. The line `from __future__ import print_function` enables Python 3 printing in Python 2.6 and later.

2. Type the following command to run the program:

```
python MoviesCreateTable.py
```

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

    ...

]
```

In the JSON data, note the following:

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

The following is an example of movie data:

```

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

Step 2.1: Download the Sample Data File

1. Download the sample data archive by clicking this link: moviedata.zip
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file to your current directory.

Step 2.2: Load the Sample Data Into the Movies Table

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

1. Copy the following program into a file named `MoviesLoadData.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

with open("moviedata.json") as json_file:
    movies = json.load(json_file, parse_float = decimal.Decimal)
    for movie in movies:
        year = int(movie['year'])
        title = movie['title']
        info = movie['info']

        print("Adding movie:", year, title)

        table.put_item(
           Item={
               'year': year,
               'title': title,
               'info': info,
            }
        )
```

2. Type the following command to run the program:

```
python MoviesLoadData.py
```

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

1. Copy the following program into a file named `MoviesItemOps01.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

response = table.put_item(
   Item={
        'year': year,
        'title': title,
        'info': {
            'plot':"Nothing happens at all.",
            'rating': decimal.Decimal(0)
        }
    }
)

print("PutItem succeeded:")
print(json.dumps(response, indent=4, cls=DecimalEncoder))
```

Note

- The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.
- The `DecimalEncoder` class is used to print out numbers stored using the `Decimal` class. The Boto SDK uses the `Decimal` class to hold DynamoDB number values.

2. Type the following command to run the program:

```
python MoviesItemOps01.py
```

Step 3.2: Read an Item

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

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

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

1. Copy the following program into a file named `MoviesItemOps02.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr
from botocore.exceptions import ClientError

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource("dynamodb", region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015
```

```
try:
    response = table.get_item(
        Key={
            'year': year,
            'title': title
        }
    )
except ClientError as e:
    print(e.response['Error']['Message'])
else:
    item = response['Item']
    print("GetItem succeeded:")
    print(json.dumps(item, indent=4, cls=DecimalEncoder))
```

2. Type the following command to run the program:

```
python MoviesItemOps02.py
```

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

1. Copy the following program into a file named `MoviesItemOps03.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
```

```

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

response = table.update_item(
    Key={
        'year': year,
        'title': title
    },
    UpdateExpression="set info.rating = :r, info.plot=:p, info.actors=:a",
    ExpressionAttributeValues={
        ':r': decimal.Decimal(5.5),
        ':p': "Everything happens all at once.",
        ':a': ["Larry", "Moe", "Curly"]
    },
    ReturnValues="UPDATED_NEW"
)

print("UpdateItem succeeded:")
print(json.dumps(response, indent=4, cls=DecimalEncoder))
```

Note
This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.
The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. Type the following command to run the program:

```
python MoviesItemOps03.py
```

Step 3.4: Increment an Atomic Counter

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

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

1. Copy the following program into a file named `MoviesItemOps04.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal

```

```
# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

response = table.update_item(
    Key={
        'year': year,
        'title': title
    },
    UpdateExpression="set info.rating = info.rating + :val",
    ExpressionAttributeValues={
        ':val': decimal.Decimal(1)
    },
    ReturnValues="UPDATED_NEW"
)

print("UpdateItem succeeded:")
print(json.dumps(response, indent=4, cls=DecimalEncoder))
```

2. Type the following command to run the program:

```
python MoviesItemOps04.py
```

Step 3.5: Update an Item (Conditionally)

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

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

1. Copy the following program into a file named `MoviesItemOps05.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
from botocore.exceptions import ClientError
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)
```

```
dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

# Conditional update (will fail)
print("Attempting conditional update...")

try:
    response = table.update_item(
        Key={
            'year': year,
            'title': title
        },
        UpdateExpression="remove info.actors[0]",
        ConditionExpression="size(info.actors) > :num",
        ExpressionAttributeValues={
            ':num': 3
        },
        ReturnValues="UPDATED_NEW"
    )
except ClientError as e:
    if e.response['Error']['Code'] == "ConditionalCheckFailedException":
        print(e.response['Error']['Message'])
    else:
        raise
else:
    print("UpdateItem succeeded:")
    print(json.dumps(response, indent=4, cls=DecimalEncoder))
```

2. Type the following command to run the program:

   ```
   python MoviesItemOps05.py
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

   This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.
3. Modify the program so that the `ConditionExpression` looks like this:

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

   The condition is now *greater than or equal to 3* instead of *greater than 3*.
4. Run the program again. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item

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

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

1. Copy the following program into a file named `MoviesItemOps06.py`.

```
from __future__ import print_function # Python 2/3 compatibility
```

```
import boto3
from botocore.exceptions import ClientError
import json
import decimal

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

title = "The Big New Movie"
year = 2015

print("Attempting a conditional delete...")

try:
    response = table.delete_item(
        Key={
            'year': year,
            'title': title
        },
        ConditionExpression="info.rating <= :val",
        ExpressionAttributeValues= {
            ":val": decimal.Decimal(5)
        }
    )
except ClientError as e:
    if e.response['Error']['Code'] == "ConditionalCheckFailedException":
        print(e.response['Error']['Message'])
    else:
        raise
else:
    print("DeleteItem succeeded:")
    print(json.dumps(response, indent=4, cls=DecimalEncoder))
```

2. Type the following command to run the program:

   ```
   python MoviesItemOps06.py
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

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

3. Now, modify the program to remove the condition in `table.delete_item`.

   ```
   response = table.delete_item(
       Key={
           'year': year,
           'title': title
       }
   )
   ```

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

Step 4: Query and Scan the Data

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

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

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

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

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query - All Movies Released in a Year

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

1. Copy the following program into a file named `MoviesQuery01.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

print("Movies from 1985")

response = table.query(
    KeyConditionExpression=Key('year').eq(1985)
)
```

```
for i in response['Items']:
    print(i['year'], ":", i['title'])
```

Note
The Boto 3 SDK constructs a ConditionExpression for you when you use the `Key` and `Attr` functions imported from `boto3.dynamodb.conditions`. You can also specify a ConditionExpression as a string.
For a list of available conditions for DynamoDB, see the DynamoDB Conditions in *AWS SDK for Python (Boto 3) Getting Started*.
For more information, see Condition Expressions in the *Amazon DynamoDB Developer Guide*.

2. Type the following command to run the program:

```
python MoviesQuery01.py
```

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

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

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

1. Copy the following program into a file named `MoviesQuery02.py`:

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            return str(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

print("Movies from 1992 - titles A-L, with genres and lead actor")

response = table.query(
    ProjectionExpression="#yr, title, info.genres, info.actors[0]",
    ExpressionAttributeNames={ "#yr": "year" }, # Expression Attribute Names for
 Projection Expression only.
    KeyConditionExpression=Key('year').eq(1992) & Key('title').between('A', 'L')
)

for i in response[u'Items']:
    print(json.dumps(i, cls=DecimalEncoder))
```

2. Type the following command to run the program:

   ```
   python MoviesQuery02.py
   ```

Step 4.3: Scan

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

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

1. Copy the following program into a file named `MoviesScan.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3
import json
import decimal
from boto3.dynamodb.conditions import Key, Attr

# Helper class to convert a DynamoDB item to JSON.
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

fe = Key('year').between(1950, 1959);
pe = "#yr, title, info.rating"
# Expression Attribute Names for Projection Expression only.
ean = { "#yr": "year", }
esk = None


response = table.scan(
    FilterExpression=fe,
    ProjectionExpression=pe,
    ExpressionAttributeNames=ean
    )

for i in response['Items']:
    print(json.dumps(i, cls=DecimalEncoder))

while 'LastEvaluatedKey' in response:
    response = table.scan(
        ProjectionExpression=pe,
        FilterExpression=fe,
        ExpressionAttributeNames= ean,
        ExclusiveStartKey=response['LastEvaluatedKey']
        )

    for i in response['Items']:
```

```
        print(json.dumps(i, cls=DecimalEncoder))
```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
- The `scan` method returns a subset of the the items each time, called a page. The `LastEvaluatedKey` value in the response is then passed to the `scan` method via the `ExclusiveStartKey` parameter. When the last page is returned, `LastEvaluatedKey` is not part of the response.

Note

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

2. Type the following command to run the program:

```
python MoviesScan.py
```

Note
You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy the following program into a file named `MoviesDeleteTable.py`.

```
from __future__ import print_function # Python 2/3 compatibility
import boto3

dynamodb = boto3.resource('dynamodb', region_name='us-west-2', endpoint_url="http://
localhost:8000")

table = dynamodb.Table('Movies')

table.delete()
```

2. Type the following command to run the program:

```
python MoviesDeleteTable.py
```

Summary

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

Moving to the Amazon DynamoDB Service

You need to change the endpoint in your application in order to use the Amazon DynamoDB service. To do this, modify the following line:

```
dynamodb = boto3.resource('dynamodb',endpoint_url="http://localhost:8000")
```

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

```
dynamodb = boto3.resource('dynamodb',region_name='us-west-2')
```

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

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

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.

Ruby and DynamoDB

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

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

You use the downloadable version of DynamoDB in this tutorial. In the Summary (p. 141), we explain how to run the same code against the DynamoDB service.

Cost: Free

Prerequisites

- Read Introduction to DynamoDB Concepts (p. 1).
- Download and run DynamoDB on your computer. For more information, see Download and Run DynamoDB (p. 4).
- Sign up for Amazon Web Services and create access keys. You need these credentials to use AWS SDKs. To create an AWS account, go to https://aws.amazon.com/, choose **Create an AWS Account**, and then follow the online instructions.
- Create an AWS credentials file. For more information, see Configuration in the *AWS SDK for Ruby API Reference*.
- Set up the AWS SDK for Ruby as follows:
  - Go to https://www.ruby-lang.org/en/documentation/installation/ and install Ruby.
  - Go to https://aws.amazon.com/sdk-for-ruby and install the AWS SDK for Ruby.

  For more information, see Installation in the *AWS SDK for Ruby API Reference*.

Note
As you work through this tutorial, you can refer to the AWS SDK for Ruby API Reference. The DynamoDB section describes the parameters and results for DynamoDB operations.

Step 1: Create a Table

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

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

1. Copy the following program into a file named `MoviesCreateTable.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

params = {
    table_name: "Movies",
    key_schema: [
        {
            attribute_name: "year",
            key_type: "HASH"  #Partition key
        },
        {
            attribute_name: "title",
            key_type: "RANGE" #Sort key
        }
    ],
    attribute_definitions: [
        {
            attribute_name: "year",
            attribute_type: "N"
        },
        {
            attribute_name: "title",
            attribute_type: "S"
        },

    ],
    provisioned_throughput: {
        read_capacity_units: 10,
        write_capacity_units: 10
  }
}

begin
    result = dynamodb.create_table(params)
    puts "Created table. Status: " +
        result.table_description.table_status;

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to create table:"
    puts "#{error.message}"
end
```

Note

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

2. Type the following command to run the program:

```
ruby MoviesCreateTable.rb
```

To learn more about managing tables, see Working with Tables in the *Amazon DynamoDB Developer Guide*.

Step 2: Load Sample Data

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

Topics

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

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

    ...

]
```

In the JSON data, note the following:

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

The following is an example of movie data:

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

Step 2.1: Download the Sample Data File

1. Download the sample data archive by clicking this link: moviedata.zip
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file to your current directory.

Step 2.2: Load the Sample Data Into the Movies Table

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

1. Copy the following program into a file named `MoviesLoadData.rb`:

```
require "aws-sdk-core"
require "json"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

file = File.read('moviedata.json')
movies = JSON.parse(file)
movies.each{|movie|

    params = {
        table_name: tableName,
        item: movie
    }
    begin
        result = dynamodb.put_item(params)
        puts "Added movie: #{movie["year"]} #{movie["title"]}"

    rescue  Aws::DynamoDB::Errors::ServiceError => error
        puts "Unable to add movie:"
        puts "#{error.message}"
    end
}
```

2. Type the following command to run the program:

```
ruby MoviesLoadData.rb
```

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

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

To learn more about reading and writing data, see Working with Items in the *Amazon DynamoDB Developer Guide*.

Topics

Step 3.1: Create a New Item

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

1. Copy the following program into a file named `MoviesItemOps01.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

item = {
    year: year,
    title: title,
    info: {
            plot: "Nothing happens at all.",
            rating: 0
    }
}

params = {
    table_name: "Movies",
    item: item
}

begin
    result = dynamodb.put_item(params)
    puts "Added item: #{year}  - #{title}"

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to add item:"
    puts "#{error.message}"
end
```

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

2. Type the following command to run the program:

```
ruby MoviesItemOps01.rb
```

Step 3.2: Read an Item

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

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

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

1. Copy the following program into a file named `MoviesItemOps02.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

key = {
    year: year,
    title: title
}

params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    }
}

begin
    result = dynamodb.get_item(params)
    printf "%i - %s\n%s\n%d\n",
        result.item["year"],
        result.item["title"],
        result.item["info"]["plot"],
        result.item["info"]["rating"]

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to read item:"
    puts "#{error.message}"
end
```

2. Type the following command to run the program:

   ```
   ruby MoviesItemOps02.rb
   ```

Step 3.3: Update an Item

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

In this example, you perform the following updates:

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

The item changes from:

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

To the following:

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

```
}
```

1. Copy the following program into a file named `MoviesItemOps03.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    },
    update_expression: "set info.rating = :r, info.plot=:p, info.actors=:a",
    expression_attribute_values: {
        ":r" => 5.5,
        ":p" => "Everything happens all at once.", # value
 <Hash,Array,String,Numeric,Boolean,IO,Set,nil>
        ":a" => ["Larry", "Moe", "Curly"]
    },
    return_values: "UPDATED_NEW"
}

begin
    result = dynamodb.update_item(params)
    puts "Added item: #{year}  - #{title}"

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to add item:"
    puts "#{error.message}"
end
```

Note
This program uses `update_expression` to describe all updates you want to perform on the specified item.
The `return_values` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. Type the following command to run the program:

```
ruby MoviesItemOps03.rb
```

Step 3.4: Increment an Atomic Counter

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

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

1. Copy the following program into a file named `MoviesItemOps04.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    },
    update_expression: "set info.rating = info.rating + :val",
    expression_attribute_values: {
        ":val" => 1
    },
    return_values: "UPDATED_NEW"
}

begin
    result = dynamodb.update_item(params)
    puts "Updated item. ReturnValues are:"
    result.attributes["info"].each do |key, value|
        if key == "rating"
            puts "#{key}: #{value.to_f}"
        else
            puts "#{key}: #{value}"
        end
    end
rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to update item:"
    puts "#{error.message}"
end
```

2. Type the following command to run the program:

```
ruby MoviesItemOps04.rb
```

Step 3.5: Update an Item (Conditionally)

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

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

1. Copy the following program into a file named `MoviesItemOps05.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    },
    update_expression: "remove info.actors[0]",
    condition_expression: "size(info.actors) > :num",
    expression_attribute_values: {
        ":num" => 3
    },
    return_values: "UPDATED_NEW"
}

begin
    result = dynamodb.update_item(params)
    puts "Updated item. ReturnValues are:"
    result.attributes["info"].each do |key, value|
        if key == "rating"
            puts "#{key}: #{value.to_f}"
        else
            puts "#{key}: #{value}"
        end
    end

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to update item:"
    puts "#{error.message}"
end
```

2. Type the following command to run the program:

   ```
   ruby MoviesItemOps05.rb
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

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

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

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

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

4. Run the program again. The `update_item` method should now succeed.

Step 3.6: Delete an Item

You can use the `delete_item` method to delete one item by specifying its primary key. You can optionally provide a `condition_expression` to prevent item deletion if the condition is not met.

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

1. Copy the following program into a file named `MoviesItemOps06.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    },
    condition_expression: "info.rating <= :val",
    expression_attribute_values: {
        ":val" => 5
    }
}

begin
    result = dynamodb.delete_item(params)
    puts "Deleted item."

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to update item:"
    puts "#{error.message}"
end
```

2. Type the following command to run the program:

   ```
   ruby MoviesItemOps06.rb
   ```

   The program should fail with the following message:

   ```
   The conditional request failed
   ```

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

3. Modify the program to remove the condition:

```
params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    }
}
```

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

Step 4: Query and Scan the Data

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

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

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

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

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

To learn more about querying and scanning data, see Query and Scan in the *Amazon DynamoDB Developer Guide*.

Topics

Step 4.1: Query - All Movies Released in a Year

The following program retrieves all movies released in the `year` 1985.

1. Copy the following program into a file named `MoviesQuery01.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = "Movies"

params = {
    table_name: tableName,
    key_condition_expression: "#yr = :yyyy",
    expression_attribute_names: {
        "#yr" => "year"
    },
    expression_attribute_values: {
        ":yyyy" => 1985
    }
}

puts "Querying for movies from 1985.";

begin
    result = dynamodb.query(params)
    puts "Query succeeded."
    result.items.each{|movie|
         puts "#{movie["year"].to_i} #{movie["title"]}"
    }

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to query table:"
    puts "#{error.message}"
end
```

Note

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

2. Type the following command to run the program:

```
ruby MoviesItemQuery01.rb
```

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

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

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

1. Copy the following program into a file named `MoviesQuery02.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = "Movies"

params = {
    table_name: tableName,
    projection_expression: "#yr, title, info.genres, info.actors[0]",
    key_condition_expression:
        "#yr = :yyyy and title between :letter1 and :letter2",
    expression_attribute_names: {
        "#yr" => "year"
    },
    expression_attribute_values: {
        ":yyyy" => 1992,
        ":letter1" => "A",
        ":letter2" => "L"
    }
}

puts "Querying for movies from 1992 - titles A-L, with genres and lead actor";

begin
    result = dynamodb.query(params)
    puts "Query succeeded."

    result.items.each{|movie|
         print "#{movie["year"].to_i}: #{movie["title"]} ... "

         movie['info']['genres'].each{|gen|
            print gen + " "
         }

         print " ... #{movie["info"]["actors"][0]}\n"
    }

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to query table:"
    puts "#{error.message}"
end
```

2. Type the following command to run the program:

```
ruby MoviesQuery02.rb
```

Step 4.3: Scan

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

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

1. Copy the following program into a file named `MoviesScan.rb`:

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

tableName = "Movies"

params = {
    table_name: tableName,
    projection_expression: "#yr, title, info.rating",
    filter_expression: "#yr between :start_yr and :end_yr",
    expression_attribute_names: {"#yr"=> "year"},
    expression_attribute_values: {
        ":start_yr" => 1950,
        ":end_yr" => 1959
    }
}

puts "Scanning Movies table."

begin
    loop do
        result = dynamodb.scan(params)

        result.items.each{|movie|
            puts "#{movie["year"].to_i}: " +
                "#{movie["title"]} ... " +
                "#{movie["info"]["rating"].to_f}"
        }

        break if result.last_evaluated_key.nil?

        puts "Scanning for more..."
        params[:exclusive_start_key] = result.last_evaluated_key
    end

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to scan:"
    puts "#{error.message}"
end
```

In the code, note the following:

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

2. Type the following command to run the program:

```
ruby MoviesScan.rb
```

Note
You can also use the `scan` method with any secondary indexes that you have created on the table. For more information, see Secondary Indexes in the *Amazon DynamoDB Developer Guide*.

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy the following program into a file named `MoviesDeleteTable.rb`.

```
require "aws-sdk-core"

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

dynamodb = Aws::DynamoDB::Client.new

params = {
    table_name: "Movies"
}

begin
    result = dynamodb.delete_table(params)
    puts "Deleted table."

rescue  Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to delete table:"
    puts "#{error.message}"
end
```

2. Type the following command to run the program:

   ```
   ruby MoviesDeleteTable.rb
   ```

Summary

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

Using the Amazon DynamoDB Service

You need to change the endpoint in your application in order to use the Amazon DynamoDB service. To do this, find the following lines in the code:

```
$sdk = new Aws\Sdk([
    'endpoint'   => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);
```

Now remove the `endpoint` parameter so that the code looks like this:

```
$sdk = new Aws\Sdk([
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);
```

After you remove this line, your code can access the DynamoDB service in the region specified by the `region` config value. For example, the following line specifies that you want to use the US West (Oregon) region:

```
'region'   => 'us-west-2',
```

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

Amazon DynamoDB is available in several regions worldwide. For the complete list, see Regions and Endpoints in the *AWS General Reference*. For more information, see the AWS SDK for Ruby Getting Started Guide.

Finally, we recommend that you read the Amazon DynamoDB Developer Guide. It provides more in-depth information about DynamoDB, including sample code and best practices.