Getting started with AWS DynamoDB

What is Cloud Computing

> You don't generate your own electricity. Why generate your own computing? - Jeff Bezos

Cloud computing is the idea of using computing as a utility. It allows users to pay for computing resources as they use them, with no upfront cost, and with effectively infinite scalability. One of the most popular uses of cloud computing is cloud data stores. This tutorial is meant to guide you through the process of using DynamoDB (a cloud computing database) to store data.

AWS DynamoDB at a glance

DynamoDB is a fully managed NoSQL database service offered by the Amazon AWS cloud stack. It provides quick, predictable performance with automatic scalabiltiy based on the provisioned throughput setitngs you can apply to your tables. DynamoDB tables can store and retrieve any amount of data, serve any level of request traffic you require for your applications, and run across multiple servers in the Amazon AWS cloud stack.

Creating a DynamoDB backend for your application

What you will need:

If you do not already have one, you can set up an AWS account here for free. We can connect to the AWS ecosystem in various ways but since we will be creating a java application, it is simplest to use Eclipse (a Java IDE), which has a plugin for AWS services called the AWS Toolkit for Eclipse. The instructions to download this are available here

Ensure that your AWS credentials are in the correct location in your file system - the config file is traditionally located in `~/.aws/credentials` on Mac/Linux.

In Eclipse, go to File -> New -> Other -> New AWS Java project.

Creating a DynamoDB client

Note that before creating the table, we must first create a client for DynamoDB. This client will need information about the AWS Access Key. We can use the ProfileCredentialsProvider class, which will return your default credential profile by reading from the credentials file located at `~/.aws/credentials`.

```
/**
* Create a client for DynamoDB. Initialize it with your AWS credentials.
**/
public void initialize() {
    AWSCredentials credentials = null;
    try {
        credentials = new ProfileCredentialsProvider().getCredentials();
    } catch (Exception e) {
        throw new AmazonClientException( "Cannot load the credentials from the credential prof
    }
    dynamoDB = new AmazonDynamoDBClient(credentials);
    Region usWest2 = Region.getRegion(Regions.US_WEST_2);
    dynamoDB.setRegion(usWest2);
}
```

Creating a table

Lets create a table called “movie-ratings” with 4 fields: “movie”, “rating”, “fans”, and “year of release”. DynamoDB requires that we specify a primary key - in this case, it makes sense to set the movie name - “movie” - as the primary key. When we create a table in DynamoDB, we must specify a table name, its primary key, and the required read and write throughput values. Since DynamoDB is a NoSQL database, except for the required primary key, it is schema-less and individual items can have any number of attributes.

DynamoDB supports two different kinds of primary keys:

- **Partition key**: Composed of one attribute. DynamoDB takes a hash of this value to determine the partition where the item will be stored. Thus must be unique because no two items in a table can have the same partition key value.
- **Partition and Sort key**: Hashes the partition key to determine the partition where the item will be stored. All items with the same partition are stored together, sorted by the sort key value.

A **KeySchemaElement** represents a single element of a key schema. It specifies the attributes that make up the primary key of a table, or the key attributes of an index. It must be a ScalarAttributeType - of either String, Number, or Binary. It cannot be nested within a List or a Map.

A key schema element’s **KeyType** an be of type Hash (partition key) and Range (sort key). Note that the *partition key* of an item is sometimes referred to as the hash attribute and the *sort key* is sometimes referred to its range attribute.

We must also specify the **ProvisionedThroughput** for the table. This determines the read/write performance desired from the table in terms of reads and writes per second.

```
DynamoDB dymanoDB = new DynamoDB(client);
String tableName = "movie-rating";
dynamoDB.createTable(tableName,
    Arrays.asList(
        new KeySchemaElement("movie", KeyType.HASH)), // Partition key
        Arrays.asList(
            new AttributeDefinition("rating", ScalarAttributeType.S),
            new AttributeDefinition("year", ScalarAttributeType.N),
            new AttributeDefinition("fans", ScalarAttributeType.S)),
        new ProvisionedThroughput(10L, 10L)); // 10L is the default
         table.waitForActive(); // polls database in a loop until the table created becomes ava
```

Add an entry to the table

Next, we will create an new entry and add it to the movie-reviews table. We can represent an entry as a map from a string (the primary key) to **AttributeValue**. We can write a helper method called 'newItem' that, given the values for the movie name, year and rating, returns a mapping from the primary key the the attribute value.

```
private static Map<String, AttributeValue> newItem(String name, int year, String rating, String
    Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
    item.put("name", new AttributeValue(name));
    item.put("year", new AttributeValue().withN(Integer.toString(year)));
    item.put("rating", new AttributeValue(rating));
    item.put("fans", new AttributeValue().withSS(fans));
}
```

Using this newItem method, we can create a **PutItemRequest** that serves as the input for the PutItem operation.

```
Map<String, AttributeValue> item = newItem("Bill & Ted's Excellent Adventure", 1989, "****", "
PutItemRequest putItemRequest = new PutItemRequest(tableName, item); // create putItemRequest

// the putItem method either creates a new item in the table, or, if the given primary key alre
dynamoDB.putItem(putItemRequest);
```

This creates a DynamoDB item (row) that maps a movie to its Attribute Value (in this case, the movies rating, year of release and fans).

Querying a table

Suppose we want to search the table for the values of a particular movie. Since we are looking up a movie entry by the movie name (which is the hash key for this table), we can use a **Query** . A query can look up a record given a hash key. If multiple records have the same hash key, then all of them will be returned, sorted by their range keys. Note that a hash key and a range key can be used to uniquely identify a record.

```
// Create a condition to evaluate the scan results and return the desired values
Condition condition = new Condition().withComparisonOperator(ComparisonOperator.GT.toString())

HahsMap<String, Condition> queryFilter = new HashMap<String, Condition>();
queryFilter.put("name", condition);
QueryRequest queryRequest = new QueryRequest().withTableName(tableName).withKeyConditions(query
QueryResult queryResult = dynamoDB.query(queryRequest);
```

Scanning a table

Suppose we don't know the name of the movie but we just want a list of movies rated 5 stars. We can search for items in the table by using the **Scan** operation. Using Scan will return one or more item attributes by accessing every item in a table. We pass in a **Condition** to the scanFilter to parse through the results and return the correct ones.

```
// Create a condition to evaluate the scan results and return the desired values
Condition condition = new Condition().withComparisonOperator(ComparisonOperator.GT.toString())

HahsMap<String, Condition> scanFiler = new HashMap<String, Condition>();
scanFilter.put("rating", condition);
ScanRequest scanRequest = new ScanRequest(tableName).withScanFilter(scanFilter);
ScanResult scanResult = dynamoDB.scan(scanRequest);
```

Concluding remarks

Setting up infrastructure to handle large amounts of data that needs to be collected and processed in realtime can be quite simple when deployed using the server less products currently available. This tutorial gave an introduction to using AWS managed services to store movie-ratings data using DynamoDB. Note that we did not have to spin up a single server yet DynamoDB can scale to read and write entire gigabytes of data per second.