Practical: DynamoDB NoSQL database service (Javascript)

Table of Contents

AWS DynamoDB is a cloud database service that uses a NoSQL data model. In this practical we will go over basic usage of DynamoDB using the AWS-SDK for Javascript.

Prerequisites and references

- DynamoDB guide (https://docs.aws.amazon.com/dynamodb/)
- DynamoDB SDK example code (https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javascriptv3/example_code/dynamodb#code-examples)
- Data modelling for DynamoDB (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/data-modeling.html)

DynamoDB basics

Overview

- AWS cloud service
- Document oriented NoSQL data model
- Good scalability
- Support for ACID transactions

Data model

DynamoDB uses a document oriented data model. The main components are:

- Tables: these hold one or more items
- Items: a group of attributes that are uniquely identifiable within the table
- Attributes: key-value pairs associated with an item

Each table has a *primary key* which consists of:

- partition key: (required) used to help distribute items across multiple storage devices. Required
- sort key: (optional) additional index that allows for queries with <, >, begins_with, etc.

The value of the primary key (i.e. the pair of values for the partition and sort keys) for an item is unique within a table, similar to how a filename identifies a unique file within a directory.

Attributes have a key, which is the name of the attribute, and the value. Values can be numbers, strings, binary data, Boolean values, lists or maps. A map is analogous to a JSON object which can contain multiple key-value pairs within it in a nested fashion.

See **Data Modelling (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/data-modeling.html)** and **Data types (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.NamingRulesDataTypes.html)** for more information.

Query model

There are two ways of searching for items within a table: *query* and *scan*:

- Query: retrieve items based on the primary key:
  - partition key: required and must be a single value
  - sort key: optional and supports additional operators like range, begins_with, <, >
  - see **API reference for Query (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html)**
- Scan: retrieve items based on item attributes:
  - reads all items it a table. This is an expensive and slow operation.
  - much more flexible than queries
  - see **API reference for Scan (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Scan.html)**

Best practice is to design your data model for your application to support the use of queries rather than scans whenever possible, for example using hierarchical values for your sort key to make use of begins_with queries (see **Sort key best practices (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-sort-keys.html)**).

Using DynamoDB in the QUT CAB432 account

For the AWS account used in this unit we have restrictions in place that limit the items that you can interact with. In particular, you must set the partition key for tables to `qut-username` and the value of this key to your QUT username (like `n1234567@qut.edu.au`).

The permissions will not let you tag tables that you create, so this is an exception to our tagging rule.

Using DynamoDB with the AWS SDK

Install the node modules

```
mkdir dynamodbdemo
cd dynamodbdemo
npm init -y
npm i @aws-sdk/client-dynamodb
npm i @aws-sdk/lib-dynamodb
npm i dotenv
```

Code to load the modules

- Create `index.js` and add the following contents:

```
require("dotenv").config();
const DynamoDB = require("@aws-sdk/client-dynamodb");
const DynamoDBLib = require("@aws-sdk/lib-dynamodb");

const qutUsername = "n1234567@qut.edu.au";
const tableName = "n1234567-kitties";
const sortKey = "name";

async function main() {
   const client = new DynamoDB.DynamoDBClient({ region: "ap-southeast
-2" });
   const docClient = DynamoDBLib.DynamoDBDocumentClient.from(client);

  // Add more content here

}

main();
```

- Change `n1234567` in `qutUsername` and in `tableName` to your own username, otherwise things will not work.

We are using to different libraries related to DynamoDB. `client-dynamodb` is the base library. This will do everything, but `lib-dynamodb` adds the *document* based commands, which allow you to use JSON formatted without specifying types. This is a much more convenient way of doing things, so we will use the document based commands wherever possible.

Create a table

- Insert the following above the `Add more content here` comment in `index.js`

```
// Create a new table
command = new DynamoDB.CreateTableCommand({
  TableName: tableName,
  AttributeDefinitions: [
      {
        AttributeName: "qut-username",
        AttributeType: "S",
      },
      {
        AttributeName: sortKey,
        AttributeType: "S", // Setting the sort key to String type
      },
  ],
  KeySchema: [
      {
        AttributeName: "qut-username",
        KeyType: "HASH",
      },
      {
        AttributeName: sortKey,
        KeyType: "RANGE",
      },
  ],
  ProvisionedThroughput: {
      ReadCapacityUnits: 1,
      WriteCapacityUnits: 1,
  },
});

// Send the command to create the table
try {
  const response = await client.send(command);
  console.log("Create Table command response:", response);
} catch (err) {
  console.log(err);
}
```

There are several things going on here:

- `AttributeDefinitions` sets the types for our partition and sort keys. They are strings, as indicated by `S`
- `KeySchema` specifies the partition and sort keys, which are `qut-username` and `name` in this case (stored in the `sortKey` variable)
- `ProvisionedThroughput` sets up AWS's resources for what capacity for reading and writing we are reserving. Leave these set to 1.

You can run the program now with `node index.js`. You should see a response indicating success, or, if you run it multiple times, an error indicating that the table already exists. If not, check for problems with authentication.

Put an item

- Insert the following above the `Add more content here` comment in `index.js`

```
// Put an object
command = new DynamoDBLib.PutCommand({
  TableName: tableName,
  Item: {
      "qut-username": qutUsername,
      [sortKey]: "Boots",
      colour: "black and white",
  },
});

// Send the command to put an item
try {
  const response = await docClient.send(command);
  console.log("Put command response:", response);
} catch (err) {
  console.log(err);
}
```

Note that we are using `docClient` here rather than `client` since we want to use the document API calls, otherwise we would need to use a more cumbersome syntax for `Item` that specifies the types for each attribute. Note that we used `[sortKey]` here to indicate that we want to use the *value* stored in `sortKey` as the attribute, rather than the string `sortKey`. That is to say, we want `name` to be the attribute name.

Here we have added `colour` as an attribute, but you can add others. The values can be strings, as here, or other things such as lists, numbers, and even objects. Basically, whatever you can describe in JSON syntax can be in the item, as long as the partition and sort key are set according to the types specified when creating the table.

Once again, you should be able to run this now with `node index.js` and see a 200 response code in the response logged to the console. If you see errors about the resource not existing then possibly the table has not finished creating yet. Try again after a few seconds.

Get an item

If you know the values for the partition key and sort key for an object then you can retrieve it directly.

- Insert the following above the `Add more content here` comment in `index.js`

```
    // Get an object
    command = new DynamoDBLib.GetCommand({
       TableName: tableName,
       Key: {
          "qut-username": qutUsername,
          [sortKey]: "Boots",
       },
    });

    // Send the command to get an item
    try {
       const response = await docClient.send(command);
       console.log("Item data:", response.Item);
    } catch (err) {
       console.log(err);
    }
```

Once again we are using the document based interface. The syntax for the command is similar to `PutCommand`, but we use `Key` instead of `Item`, and specify only the values of the partition and sort keys.

If you run the program now you should see in the console the item that we previously put.

Making a query

A query looks for objects that match an expression on the partition key and the sort key. Note that the partition key must match exactly, but there is more flexibility with the sort key. See **Query (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html)** in the API reference for more information about queries.

- Insert the following above the `Add more content here` comment in `index.js`

```
    // Query
    command = new DynamoDBLib.QueryCommand({
       TableName: tableName,
       KeyConditionExpression:
          "#partitionKey = :username AND begins_with(#sortKey, :nameSt
 art)",
       ExpressionAttributeNames: {
          "#partitionKey": "qut-username",
          "#sortKey": sortKey,
       },
       ExpressionAttributeValues: {
          ":username": qutUsername,
          ":nameStart": "Boo",
       },
    });

    // Send the command to run a query
    try {
       const response = await docClient.send(command);
       console.log("Query found these items: ", response.Items);
    } catch (err) {
       console.log(err);
    }
```

Queries are more complex than get or put commands:

- `KeyConditionExpression` specifies how we want to match against the partition and sort keys. Note that the partition key must use the `=` operator. We have used `#partitionKey` instead of the actual partition key name `qut-username` because `-` is not allowed in this expression. Instead we use the `#partitionKey` which will be replaced by the string we specify. We've used something similar for the sort key, and used `:username` and `:nameStart` for other values that we want to substitute in.
- `ExpressionAttributeNames` specifies how to replace `#partitionKey` and `#sortKey` with actual strings. This lets us use the `sortKey` variable and also gets around the restriction of not using `-` in the expression. We can use other tokens, but they must start with `#`. `#P` and `#S`, for example, would also be fine, as long as we change them in the `KeyConditionExpression` as well
- `ExpressionAttributeValues` specifies how to replace `:username` and `:nameStart` with actual strings. We used `qutUsername` as the value for `:username` and a hard-coded string for `:nameStart`.

The overall condition then is:

- the value of the partition key must be your username
- the value of the sort key (`name`) must start with `Boo`.

If all goes well, when you run the program now you should see a list containing the single object that matches our condition.

Other operations

There are other operations that are possible which we leave to you to explore. Have a look at the **DynamoDB SDK example code (https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javascriptv3/example_code/dynamodb#code-examples)** to see how to use them. You might be interested in:

- updating an item
- scanning for items (like queries, but looks at attributes other than the primary key)
- deleting an item

Delete the table

- When you are done, delete the table using the AWS console

Complete code: **dynamodbdemo.zip (https://canvas.qut.edu.au/courses/20367/files/6583676/download)**