Module 14
DynamoDB: NoSQL Database Service

Overview

In Module 12 and Project 1 you experimented with a relational database service (RDS) such as PostgreSQL. You are already familiar with relational databases from the database course you took. The most pronounced difference between RDS and a traditional database installed in a local network is: RDS is deployed in the cloud and is a fully **managed service**. That is, AWS does the heavy lifting of hardware provisioning, installing the database, storage allocation, backup & restore, version upgrade, etc. You just use the database engine as a service. Once you connect to it from your code (e.g., Lambda), you use it in a similar way you use a local database (open connection, execute some SQL query, etc.).

Before we delve into the topic of another database technology (NoSQL databases), let's first define two terms that you will often hear: **vertically-scalable** and **horizontally-scalable**. These terms are applicable and important in many disciplines and not just databases. We will start with them to drive the discussion.

- **Vertically Scalable**:   Vertically Scalable means if you want to improve something, buy a bigger and more powerful version of it (Figure 1).

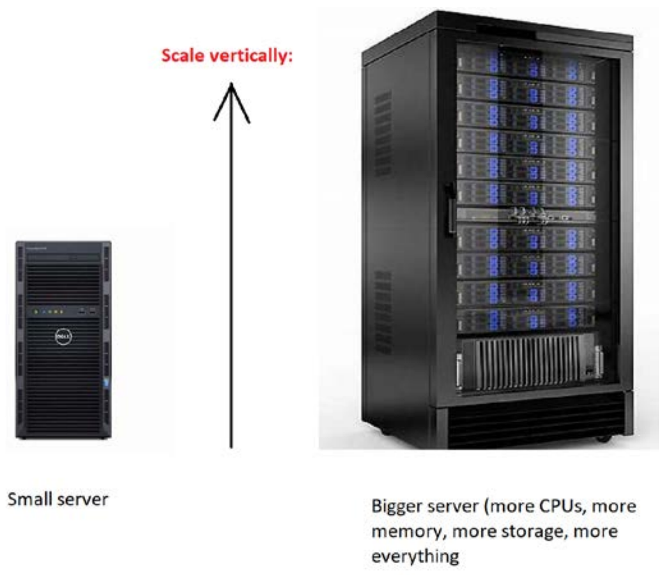

Scale vertically:
Small server
Bigger server (more CPUs, more memory, more storage, more everything

Figure 1: Assume we have a database on the small server shown on the left. If the hope is to improve the performance of the database, one thing that can help (maybe, or up to a point) is installing it on a bigger and more powerful machine (on the right). The idea is: all else being equal, a bigger server (with more CPUs, memory, storage) is faster than a smaller one.

- **Horizontally Scalable**: Horizontally Scalable means if you want to improve something, buy more instances of it (Figure 2).

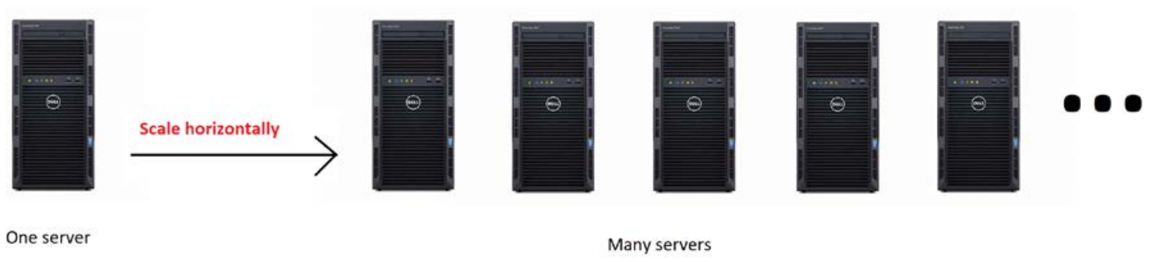


Figure 2: Assume we have a database on the small server shown on the left. If we want to improve the performance of the database, one thing that can help is spreading the work on multiple machines. The idea is: all else being equal, many servers can accomplish more work than one server.

By comparing Figures 1 and 2, it becomes evident that vertical scalability cannot grow at the same scale as horizontal scalability. With vertical scalability, physical and technological limitations makes it such that the server cannot be made infinitely powerful and you will soon hit a limit. With horizontal scalability, however, hundreds of thousands of servers can be used to create ever larger clusters.

Why and how is this related to the topic of databases? It turns out that it is very difficult to adapt a relational database that was designed to run on one server to distribute its data and run on multiple servers. For example, assume a relational database stores one table (Table1) on server1, a second table (Table2) on server2, and a third table (Table3) on server3. If the developer writes a query that uses all 3 tables using INNER JOINS, the database engine needs to fetch and compare keys from 3 servers to build a result set. This requires network communications among the 3 servers to compare keys and fulfill the query. This considerably slows down the database. There are also no clear strategies to guide how the data should be split. If queries were such that tables are queried in isolation (that is, only one table is involved in a query), then placing each table on a distinct server would work well. But this is not how relational databases are designed. Tables are usually related to each other and a single query can involve half a dozen tables linked together in complex ways via keys. Therefore, the only practical way to scale relational databases is vertical scalability. A bigger server with more CPUs and memory will improve performance, but only up to a limit (the limit at which the server can no longer be made more powerful).

This vertical scaling may provide a short-term advantage but only goes so far. Furthermore, even with best tuning practices, relational database performance usually degrades as the database size increase (billions of rows). Some companies (e.g., Teradata) have tried to horizontally scale relational databases. Such offerings also eventually go so far and are not as scalable as NoSQL databases.

From an application design perspective, the reliance on a relational database makes frequent changes to an application difficult and risky. This is because the schema of a relational database is somewhat rigid (you have a set of tables each having a well-defined set of columns). It is not possible to add a new piece of information if there is no column for it in some table. And very often adding one piece of information may cause a ripple effect where a few tables and relationships may need to change as well, thus disturbing the model. In addition, this almost always requires simultaneous in-step changes to the client application and anything in between the front end and the database (e.g., a middle-tier database layer in a 3-tier architecture). This by itself is not a weakness as many types of applications have well-structured data. But many other applications don't have well-structured data and are frequently updated with new features. For these, a relational database is a not a recommended data store as it makes incremental and fast changes difficult to achieve.

Reading

- Dynamo DB (Amazon's NoSQL database) uses **Consistent Hashing** to distribute data on different servers and be able to infinitely scale the database. Read and understand **Consistent Hashing** by reading this article, which does a good job explaining how strategies other than Consistent Hashing cannot efficiently address scalability.

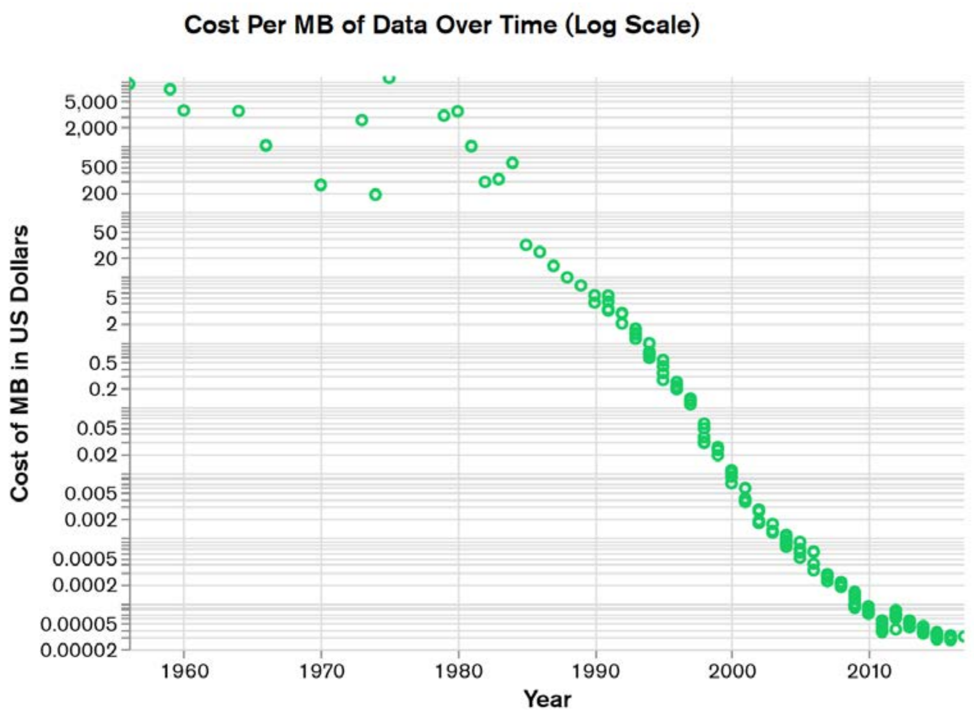
- Read this paper (search the paper for "consistent hashing" and read around how it is used)

NoSQL Databases

NoSQL databases address the above relational databases shortcomings. Let's look at them each a time:

Infinitely Scalable

One of the main reasons relational databases do not horizontally scale has to do with table joins. Joins are usually done to normalize data to reduce storing redundant information. This works well on a single machine and was needed when databases were invented 50 years ago because storage then was very expensive (Figure 3). You can see from the figure that 1 MB had a cost of ~ $5,000 in the late 1950s. Today 1 MB costs close to 0.001 cents.



s

Figure 3: Historical cost of storage (source).

If you split tables and create joins across multiple computers, a client query with a join may need to fetch data from different machines with lots of keys comparisons to satisfy the query. This necessitates a lot of network traffic in addition to the computation needed on each computer.

In a NoSQL database you are supposed to think ahead of time of how you will retrieve your data. You design your tables such that a given query is satisfied by one machine. NoSQL databases do away with joins and trade storage efficiency to gain in scalability – an acceptable tradeoff because storage cost today is much lower than what it was 50 years ago (see Figure 3).

The DynamoDB paper that you read describes how **Consistent Hashing** is used to determine on what node to store a particular key/value. The Consistent Hashing distribution scheme makes it such that adding/removing a server only requires redistribution of keys in the neighborhood around that server location on the ring. Keys in other locations don't need re-distribution. And this makes the cluster as a whole scalable and elastic.

Schema-less and Flexible

This is explained well in this page. Look at sections "*Scoping for changing requirements*" and "*Flexibility for Faster Development*". Look at Figure 1 in the link. It shows you an example where if the twitter account of a user needs to be saved, the schema needs to be changed. This slows down the development and can cause ripple effects in many layers of the application (each layer had to be modified to account for this new twitter data item).

There are many NoSQL databases. The most widely used ones are Dynamo DB (Amazon), Cosmos DB (Microsoft), and Mongo DB. We will use Dynamo DB to experiment with it.

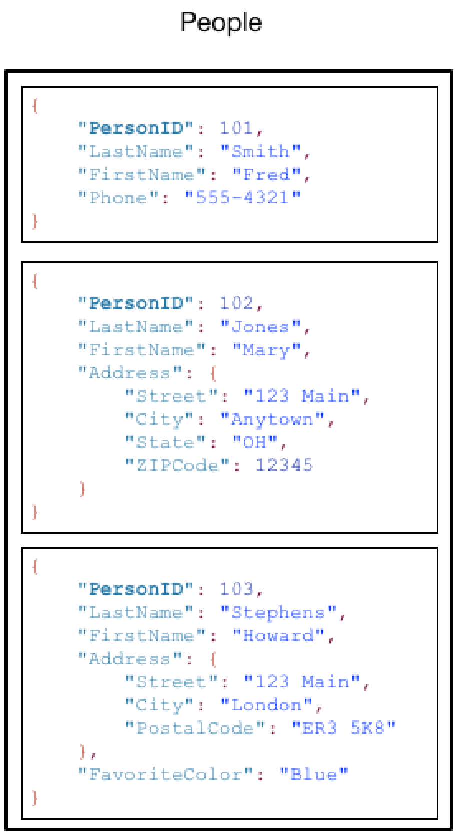
Dynamo DB

1. Terminology: read this and understand the core components of Dynamo DB: **Table**, **Item**, and **Attribute**.

2. Partitioning: read this and understand how partitioning works.

   After reading the above 2 pages, observe the following:

   a. A Dynamo table is schema-less. Unlike in a relational database where a table has a well-defined fixed set of columns (attributes) and all rows have these same attributes, different items in Dynamo DB can have different attributes. For example, in the People table in Figure 4 below, Mary and Howard have an Address, but Fred doesn't. Howard has a FavoriteColor that both Fred and Mary don't have. And Fred has a phone whereas Mary and Howard don't.



```
{
    "PersonID": 101,
    "LastName": "Smith",
    "FirstName": "Fred",
    "Phone": "555-4321"
}
```

```
{
    "PersonID": 102,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
        "Street": "123 Main",
        "City": "Anytown",
        "State": "OH",
        "ZIPCode": 12345
    }
}
```

```
{
    "PersonID": 103,
    "LastName": "Stephens",
    "FirstName": "Howard",
    "Address": {
        "Street": "123 Main",
        "City": "London",
        "PostalCode": "ER3 5K8"
    },
    "FavoriteColor": "Blue"
}
```

Figure 4: People table (source).

b. An attribute can have nested attributes (like the Address attribute in Figure 4). In a relational database this is usually set up as a second table with primary/foreign keys relating the two tables.

c. Each item has a primary key (in the People table it is the PersonID attribute). The primary key is a unique identifier that distinguishes items (no two items can have the same primary key). The primary key is known as the **partition key** and determines the partition (physical storage) where that item gets stored. For example in Figure 5 the partition key is AnimalType. The value of the AnimalType ("Dog" in this case) is inputted to a hash function. The output of the hash function determines the physical partition where the item gets stored. The same happens when a user requests the item back by supplying the partition key. The database hashes the partition key again to determine in what partition it is located. Note that partitioning is internal to Dynamo DB and is transparent to the programmer.

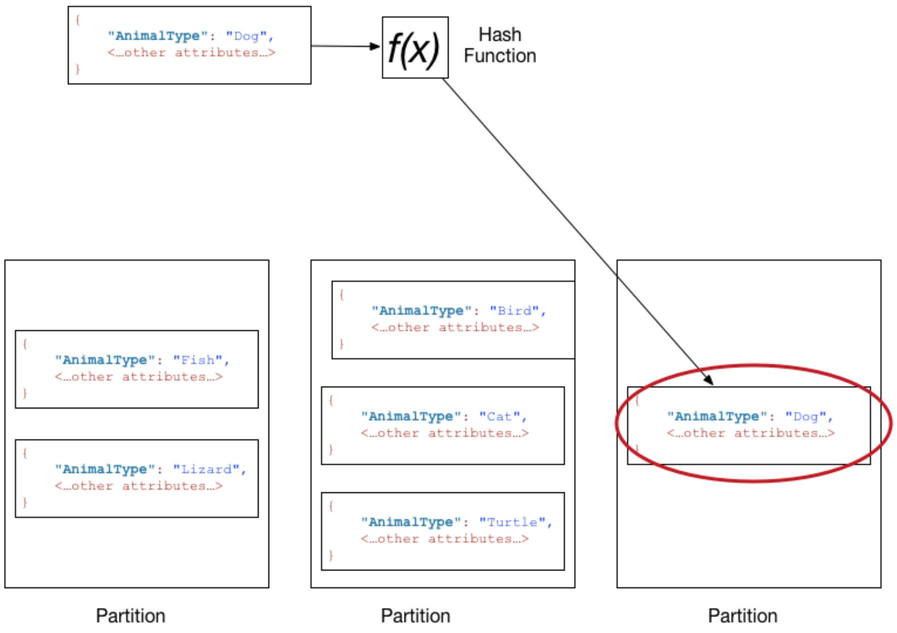
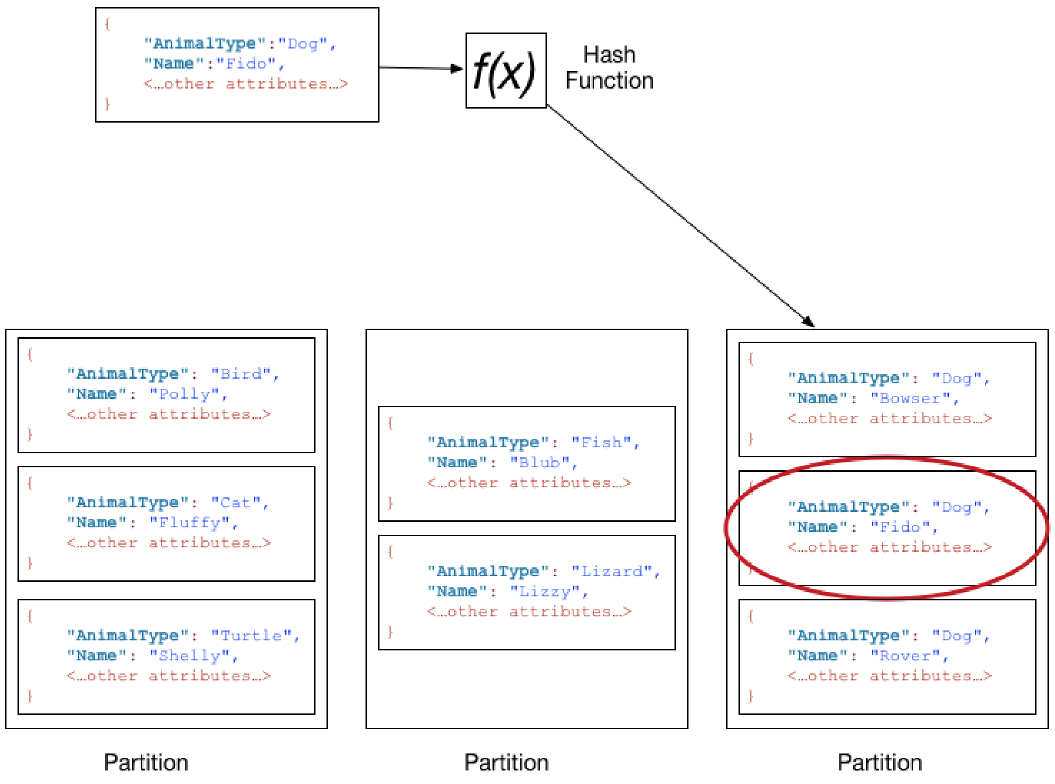

"AnimalType": "Dog", <...other attributes...>
f(x)
Hash Function
"AnimalType": "Fish", <...other attributes...>
"AnimalType": "Lizard", <...other attributes...>
"AnimalType": "Bird", <...other attributes...>
"AnimalType": "Cat", <...other attributes...>
"AnimalType": "Turtle", <...other attributes...>
"AnimalType": "Dog", <...other attributes...>
Partition
Partition
Partition

Figure 5: Pets table. The partition key dictates the partition where an item gets stored (source).

Because the partitioning is transparent to the programmer, you should think of the Pets table as **one** table and not three tables. It is one table that Dynamo internally spread its data on 3 partitions.

d. In Figures 4 and 5 the primary key is a single attribute (PersonID in the People table and AnimalType in the Pets table). Dynamo DB supports a primary key made up of two attributes (known as *composite primary key*). The first attribute is the **partition key** and the second attribute is the **sort key**. The partition key is still used to determine the partition where the item gets stored. The sort key is used to sort all items that happen to be on a given partition. Figure 6 illustrates this concept where the Pets table has a composite primary key: the partition key AnimalType and the sort key Name. The value of the partition key ("Dog") determines what partition the item ends up on (in this case the one on the far right). The value of the sort key ("Fido") determines the location of this item on its partition. In this case the items on the far right partition are alphabetically sorted by the Name sort key. This speeds up search when trying to retrieve an item.



{ "AnimalType":"Dog", "Name":"Fido", <…other attributes…> }
f(x)
Hash Function
{ "AnimalType": "Bird", "Name": "Polly", <…other attributes…> }
{ "AnimalType": "Cat", "Name": "Fluffy", <…other attributes…> }
{ "AnimalType": "Turtle", "Name": "Shelly", <…other attributes…> }
{ "AnimalType": "Fish", "Name": "Blub", <…other attributes…> }
{ "AnimalType": "Lizard", "Name": "Lizzy", <…other attributes…> }
{ "AnimalType": "Dog", "Name": "Bowser", <…other attributes…> }
{ "AnimalType": "Dog", "Name": "Fido", <…other attributes…> }
{ "AnimalType": "Dog", "Name": "Rover", <…other attributes…> }
Partition
Partition
Partition

Figure 6: Pets table with a *composite primary key* (primary key made up of two keys: a partition key and a sort key). The partition key dictates the partition where an item gets stored. The sort key dictates the location of an item in its partition (source).

e. The partition key is also known as the **hash attribute**.
The sort key is also known as the **range attribute**.

f. Partitions are automatically replicated across AZs in a region.

Create a DynamoDB table

1. Login to the AWS Console and go to the DynamoDB service.

2. Go to this page and do the first 4 steps

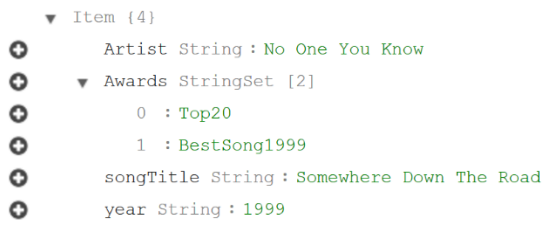
    a. Ignore step 1-e that says "*Next, you will enable DynamoDB auto scaling for your table.*"
    b. Do not do step 5 (Delete a NoSQL Table) which deletes the table.

    At the end of step 4, you should have a Music table with a few items in it.

3. Click on any one of the items.

4. Hover over the + sign to the left of a field and select menu Insert → String. Then enter a field name and its value (e.g., "year" for field and "1999" for value).
5. Hover over the + sign to the left of a field and select menu Insert → StringSet. Then enter "Awards" for the field name. And enter two different values: BestSong1999, Top20.

    When done you should have something that resembles the below (the order of fields is not important):



```
▾ Item {4}
    Artist String : No One You Know
  ▾ Awards StringSet [2]
        0  : Top20
        1  : BestSong1999
    songTitle String : Somewhere Down The Road
    year String : 1999
```

6. Click the **Save** button.

7. Observe that you now have a year and Awards fields in the table. In addition, other items in the table don't necessarily need to have these fields.

    This demonstrates the flexibility discussed in section "*Schema-less and Flexible*" on page 4. Specifically, you were able to:

    a. For a given item, add 2 fields on the fly. This wouldn't be possible in a relational database where you would first need to first change the structure of the table (by adding two columns to hold the values). And this is no easy feat – such minor change would likely require updating your object model (how you are modeling the data in classes and objects) in many layers of your application.

    b. For a given attribute ("Awards"), add nested attributes.

8. Click the item you edited to open the Edit item window again.

9. In the top left of the window, switch from **Tree** view to **Text** view. Notice how the data is a JSON document:

```
{
  "Artist": {
    "S": "No One You Know"
  },
  "Awards": {
    "SS": [
      "Top20",
      "BestSong1999"
    ]
  },
  "songTitle": {
    "S": "Somewhere Down The Road"
  },
  "year": {
    "S": "1999"
  }
}
```

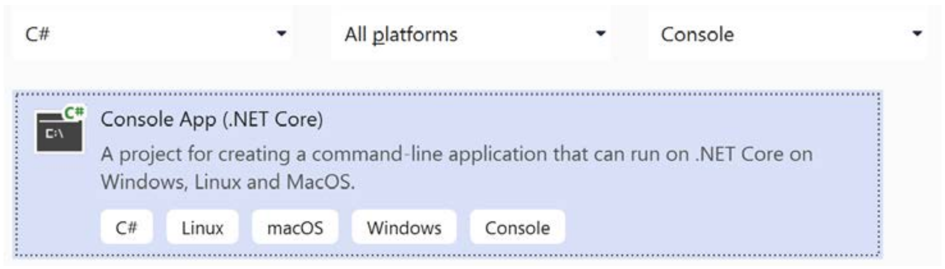
10. Click the Triggers tab and read the sentence in the blue box. This is an analogous to the S3 triggers you experimented with in Project 1 where adding a file to S3 triggered an invocation of a Lambda function. In this case adding or modifying a Dynamo DB item can also be configured to trigger a Lambda function. You can obviously think how this allows accomplishing many use case scenarios where we need code to act on some new data.

    Note that relational databases also have triggers. They are different from the triggers of Dynamo DB.

Interact with a Dynamo DB table from a Client

1. Create a Visual Studio solution and name it **DynamoDBSolution**.

2. Under the above solution, create a Console App (.NET Core) project and name it **DynamoDBClient**.

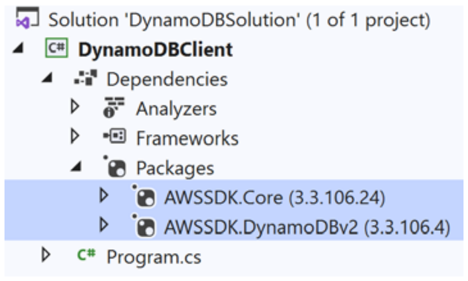

C#
All platforms
Console
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C#
Linux
macOS
Windows
Console

3. In the Solution Explorer pane, right click on Dependencies and choose menu **Manage NuGet Packages…**.

4. Choose the Browse tab and search for:

AWSSDK.Core:	Install it by clicking the Install button.
AWSSDK.DynamoDBv2:	Install it by clicking the Install button.

   Your project should now have these 2 packages:

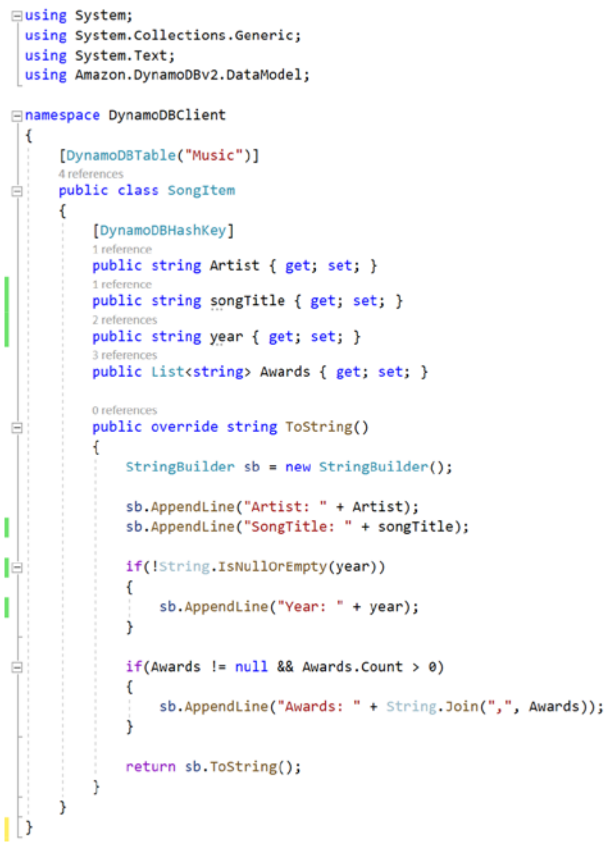


```
Solution 'DynamoDBSolution' (1 of 1 project)
  DynamoDBClient
    Dependencies
      Analyzers
      Frameworks
      Packages
        AWSSDK.Core (3.3.106.24)
        AWSSDK.DynamoDBv2 (3.3.106.4)
    Program.cs
```

5. Open file Program.cs and add the following using statements:

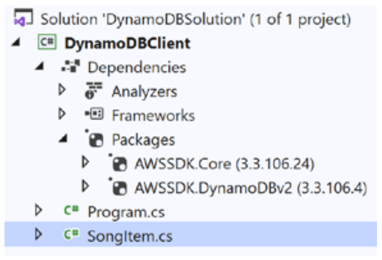
```
using Amazon;
using Amazon.Runtime;
using Amazon.Runtime.CredentialManagement;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
```

6. Add to the DynamoDBClient project a class named **SongItem**. Complete its code as shown below. (Note that the case of properties names should match the attributes names you used in your Dynamo DB table. Example: year and not Year).



```
using System;
using System.Collections.Generic;
using System.Text;
using Amazon.DynamoDBv2.DataModel;

namespace DynamoDBClient
{
    [DynamoDBTable("Music")]
    4 references
    public class SongItem
    {
        [DynamoDBHashKey]
        1 reference
        public string Artist { get; set; }
        1 reference
        public string songTitle { get; set; }
        2 references
        public string year { get; set; }
        3 references
        public List<string> Awards { get; set; }

        0 references
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();

            sb.AppendLine("Artist: " + Artist);
            sb.AppendLine("SongTitle: " + songTitle);

            if(!String.IsNullOrEmpty(year))
            {
                sb.AppendLine("Year: " + year);
            }

            if(Awards != null && Awards.Count > 0)
            {
                sb.AppendLine("Awards: " + String.Join(",", Awards));
            }

            return sb.ToString();
        }
    }
}
```

Your project files structure should now look like this (with the new SongItem class added):


Solution 'DynamoDBSolution' (1 of 1 project)
DynamoDBClient
Dependencies
Analyzers
Frameworks
Packages
AWSSDK.Core (3.3.106.24)
AWSSDK.DynamoDBv2 (3.3.106.4)
Program.cs
SongItem.cs

7. Go back to the Program.cs class and add three methods to the Program class:

```
private static void ListTables(AmazonDynamoDBClient client)

private static void GetItemsByArtist(DynamoDBContext context, string artist)

private static void SaveItem(DynamoDBContext context, SongItem newItem)
```

Exercise to Do and Submit

1. Complete the implementations of the 3 methods listed above in item 7.

   HINTS: I used the following to complete this exercise (this does not mean that these are the only way the exercise can be completed. As always, there might be other ways to accomplish the same thing. However, I was able to complete it with using the below).

   Classes:

   ```
   AmazonDynamoDBClient
   DynamoDBContext
   AsyncSearch
   ```

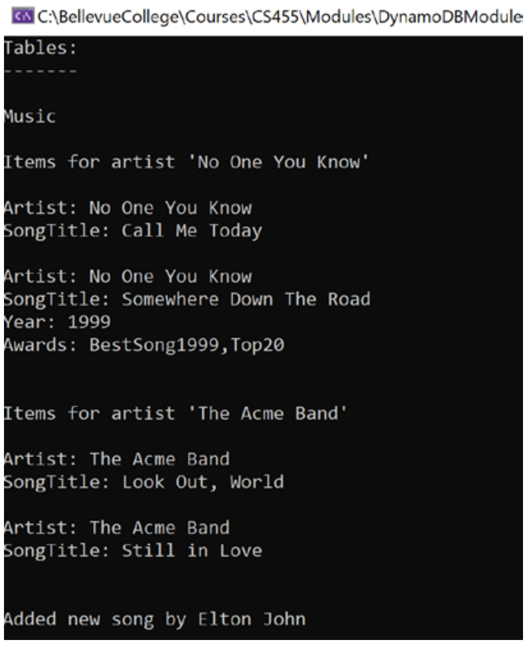
   Methods:

   ```
   ListTablesAsync
   QueryAsync
   GetRemainingAsync
   SaveAsync
   ```

2. Call the 3 methods from Main to test them. For the 3rd method, `SaveItem`, you can add the following song item:

```
SongItem newSong = new SongItem()
{
    Artist = "Elton John",
    songTitle = "Circle of Life",
    year = "1995",
    Awards = new List<string>() { "BestSoundtrack" }
};
```

3. Test your console app. Your output should somehow resembles what is shown in Figure 7:

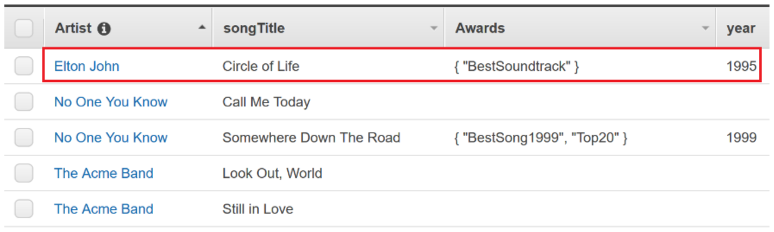


C:\BellevueCollege\Courses\CS455\Modules\DynamoDBModule

```
Tables:
-------

Music

Items for artist 'No One You Know'

Artist: No One You Know
SongTitle: Call Me Today

Artist: No One You Know
SongTitle: Somewhere Down The Road
Year: 1999
Awards: BestSong1999,Top20


Items for artist 'The Acme Band'

Artist: The Acme Band
SongTitle: Look Out, World

Artist: The Acme Band
SongTitle: Still in Love


Added new song by Elton John
```

Figure 7: Output of DynamoDBClient.

4. Go to DynamoDB in the AWS Console and verify that the new song was added (Figure 8).



Artist	songTitle	Awards	year
Elton John	Circle of Life	{ "BestSoundtrack" }	1995
No One You Know	Call Me Today		
No One You Know	Somewhere Down The Road	{ "BestSong1999", "Top20" }	1999
The Acme Band	Look Out, World		
The Acme Band	Still in Love		

Figure 8: The Music table after adding a new item from the DynamoDBClient app.

What to Submit

Complete **Module14-Quiz** after you finish this module.