Exercise 8

[*version_1.0.2*]

Exercise: Single table with Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

Overview

In this exercise, you will learn how to *develop* with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will work with the single table concept of DynamoDB and use the AWS SDK. This lab gives you hands-on experience with both Amazon DynamoDB and AWS Cloud9.

Objectives

After completing this lab, you will be able to use the AWS SDKs to do the following:

- Create a table for the dragon data using `single table design`.
- Uploading multiple items to a single table DynamoDB table.
- Do advanced querying using composite sort key

Story continued

So your work with Mary is done. She is delighted with the work you have done for her, and her developers can now work towards creating the full game engine.

However you want to push your DynamoDB skills further by implementing a more advanced schema (single table). You wanted to see how this would affect the existing queries and if you could add more elaborate search functionality to the site.

As this is just going to be an experiment to show Mary the single table approach, you are not going to reimplement all the editing and password protected areas for this. You are simply going to stand on the shoulders of the older `index2.html` by creating a new experimental version (`index5.html`). That way you can prove to Mary that not only is the same functionality possible using a single table, but also that more advanced queries can be carried out.

To save time, you figure you will be able to leverage much of the boilerplate code you have used thus far and tweak it accordingly for your new API.

Your ideas are as follows:

- Create a `single table` version of dragon data .
- Ensure that the basic version of the site can maintain the same functionality using this `single table` pattern
- Prove to yourself, and Mary that there are real advantages in using the `single table` concept.

Your thought process

You first reflect on the dragon queries that you have on the current *(old index2.html)* site, and think about how you can achieve the same thing with a single table schema.

What you had in `index2.html`:

- Ability to return all dragons filtering out all but a few attributes, using a `scan and filter`.
- Ability to return information on a specific dragon, again returning only a few attributes, using a `query`.

Since you built `index2.html`, you have learned (in lab 4) that a scan is **usually** less efficient that a query, so we will keep that in mind when we look at the single table version.

It is important that you can still do these types of query, so your first draft of the single table is as follows.

PK (PK)	**Location**	
	<Location_country_value>:<Location_state_value>:<Location_city_value>:<Location_neighborhood_value>	dragon_name:, protection:, damage:, description:, family:

So the biggest change here is that we have a new Primary Key called `PK`. It is going to be a random key of type `String`, with no sort key. Similar to how you did the session table earlier in lab 5.

Also note we are creating a `location` attribute as a `composite key` value of *all* the location attributes. This will reduce table size a little bit, and allow us to create GSIs (later on) that will enable *advanced* searching. (more on that stuff later).

So in pseudocode (not real code), you would probably do something conceptually a bit like this:

```
SCAN single_dragon_table --FILTER where dragon_name EXISTS
```

This is not super efficient, but *not really any worse* that what you had before with the `dragon_stats` scan.

However things start to fall down when you come to search for a specific dragon. It becomes pretty inefficient to use the "scan and filter" approach as you saw in lab 4.

You really *want* to use a query, but you can't...Well, not on this `single_dragon_table` at least.

So you think, "an Index to the rescue!"

Great idea!

You could create a GSI using `dragon_name` as the primary key, with no sort key. You could call it `dragon_stats_index`, projecting only the things you want returning, such as `description`, `protection` and `damage`.

Note the `primary key (PK)` of this index `(dragon_name)` will always be returned even when not specified in the projection

The `dragon_stats_index` would look like this:

PK	**`<attributes>`**
`<dragon_name_value>`	protection:`<value>`, damage:`<value>`, description:`<value>`, family:`<value>` *[uuid-PK added by default]*

Notice no location info bloating the index. #win

So in pseudocode, getting just one dragon would look a bit like this

```
QUERY dragon_stats_index WHERE dragon_name (PK) = dragon_requested
```

So you think using these 2 types of queries, you could achieve the same site functionality.

The advantage here is that the GSI is a sparse index, only keeping items that have a `dragon_name`, so later on when you add more items to the single table that are not items with a `dragon name` *attribute*, it can efficiently pull only what it needs.

Let's think a little more about the concept of a scan (again in pseudocode)

```
SCAN single_dragon_table --FILTER where dragon_name EXISTS
```

This actually would be slightly worse than the non single table version scan because when you add many items later it will scan all of the items, which is far from ideal.

If we move to the single table concept it needs to be better not worse! Otherwise what's the point.

The query we can do on the single table's index is really good, however the scan is not so good. But guess what? You already have the perfect scan opportunity, you simply scan the index!

So in pseudocode this is definitely the way to go.

```
SCAN dragon_stats_index
```

This is very efficient it only returns what you need, with no filtering required, other than having to ignore the PK which gets returned whether we like it or not. This scan doesn't touch any items that don't need returning. This is the best type of scan.

Ok, so now you know the plan of action to recreate the base site functionality, let's create our first attempt of a single table and build an API Gateway and the Lambda function behind it.

Goals of this lab:

- Exercise 1) Create the single table (single_dragon_table) along with the GSI (dragon_name_index)
- Exercise 2) Seed the single table by uploading items via JSON (similar to what you did before)

NOTE, as you go through later parts of tis lab, you will add other items from the other tables into this single table. For now (Part 2), we are just concentrating on getting the dragon stats in there, to ensure we can replicate the base site functionality first .

As mentioned, we are also not going to integrate all the editing features we did in lab 7 or the membership stuff from lab 5, as at this stage we are simply proving that the same base functionality is possible *(and slightly better)* using the `single table` concept.

- Exercise 3) Create and test a new API and Lambda backend, that will talk to the `single table`'s index.
- Exercise 4) Test the new website works in the same way (but more efficiently) that the old website.

Prepare the lab

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that has been prepared for you.

1. From your Cloud 9 Environment collapse any folders and close any tabs you are not longer using.

2. Ensure you are in the right path in your Cloud9 terminal using:

```
cd /home/ec2-user/environment
```

To get the files that will be used for this exercise, go to the Cloud 9 **bash terminal** (at the bottom of the page) and run the following `wget` command:

```
wget https://s3.amazonaws.com/awsu-hosting/edx_dynamo/c9/dynamo-single/lab8.zip -P
/home/ec2-user/environment
```

You should also see that a root folder called **dynamolab** with a `lab8.zip` file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).

4. To unzip the `lab8.zip` file, by running the following command:

```
unzip lab8.zip
```

This may take a few moments. In your Cloud9 filesystem.

5. To keep things clean, run the following commands to remove the zip file:

```
rm lab8.zip && cd lab8
```

6. Select the black arrow next to the `lab8` folder (top left) to expand it. Notice inside this `lab7` folder there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**

Step 1: Create the single table (single_dragon_table)

Creating a table and creating an index shouldn't take you long because you have done this sort of thing before,

1. Open the SDK docs and find the method for creating a table and index.

Language	AWS documentation deep link
NODE.JS (8.16.0)	https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#createTable-property.

You should be in your Cloud 9 environment.

It's time to write some code that creates a new table and index.

1. Open up the `create_single_table_and_index.js` file inside the `lab8` folder by double clicking on it.

2. Have the SDK docs open (as above) to help you

    1. Replace the sections of the code in that file, so that the code creates a new table called `single_dragon_table` along with an index called `dragon_stats_index` in `us-east-1`.

        This is the structure we are going for:

PK	
<dragon_name_value>	protection:, damage:, description:, family: *[uuid-PK added by default]*

3. Save the file.

4. Go to the terminal and run your file using the respective run command below

```
node create_single_table_and_index.js
```

IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER.

Confirm that your code worked.

You should see something like this in the console.

```
null { TableDescription:
   { AttributeDefinitions: [ [Object], [Object] ],
     TableName: 'single_dragon_table',
     KeySchema: [ [Object] ],
     TableStatus: 'CREATING',
     CreationDateTime: 2019-05-30T15:13:06.336Z,
     ProvisionedThroughput:
      { NumberOfDecreasesToday: 0,
        ReadCapacityUnits: 0,
```

```
        WriteCapacityUnits: 0 },
    TableSizeBytes: 0,
    ItemCount: 0,
    TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/single_dragon_table',
    TableId: 'db146e33-d82f-4143-8103-fe45db8e0cca',
    BillingModeSummary: { BillingMode: 'PAY_PER_REQUEST' },
    GlobalSecondaryIndexes: [ [Object] ] } }
```

Head over to your DynamoDB console, and you will see that you have a new table called `single_table` being created.

⚠ Wait until its says `ACTIVE` before moving on, because you can't update a table when it is not `ACTIVE`, and we need to do that next. *(This can take upto 5 minutes)*

Step 2: Seed the single table with dragon stats

Once your table is `ACTIVE` we can seed it like we did before.

The only ~~tricky~~ different part here is that you will need to create a `composite location` attribute.

You are going to use this `location` attributes (later on) as part of a GSI where it will play the role of Sort Key, to enable some cool search functionality.

```
location_country_value#location_state_value#location_city_value#location_neighborhood_value
```

Written like so: (spaces are ok), all lowercase.

*FYI: Location is where these dragons were last spotted roaming in our Universe by our **dragon drones**.

```
usa#nevada#las vegas#spring valley
```

1. Open the SDK docs and find the method for **batchWriteItem**

Language	AWS documentation deep link
NODE.JS (8.16.0)	https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#batchWriteItem-property.

You should be in your Cloud 9 environment.

It's time to write some code that creates a new table and index.

1. Open up the `seed_single_table.js` file inside the `lab8` folder by double clicking on it.

2. Have the SDK docs open (as above) to help you

1. Replace the sections of the code in that file, so that the code reads the items from the old `dragon_stats` JSON files you had in lab 3 (copied over to `lab 8/resources`, and use this to fill in the new `single_dragon_table`. You'll probably notice the code is very similar to what you had in lab 3, just with a different schema

3. Once done, save the file.

4. Go to the Cloud 9 terminal and run your file using the command below

> ⚠ *Make sure you don't run this following command twice, as the PK is a unique identifier (UUID) with negligible chance of collisions, so each time you run this command it will create new items over and over. You could add an `exists` condition (which we will talk about later) to prevent this, but it is **just easier** not to seed it twice ;)*

```
node seed_single_table.js # just the once please ;)
```

IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER.

Confirm that your code worked.

You should see something like this in the console.

```
[ { UnprocessedItems: {} }, { UnprocessedItems: {} } ]
HowFastWasThat: 89.387ms
```

- Check your DynamoDB console and press the **single_dragon_table**.
- Now press items. You should see items in there with `UUID` as the Primary Key.
- Where you see the word `scan` in the drop down next to it you can choose the Index `dragon_stats_index`. Then press **scan** and you will see the same items, except with only the projected attributes that you set for that index.

Step 3: Create and test a new API resource along with a Lambda backend, that will talk to the `single table`'s index

You will need to create a new API gateway resource here, as we don't want to interfere with the working website.

We will call this resource `single`

We have provided you with a clone of `index2.html` called `index5.html` with a few tweeks in the JavaScript, that you will use (later) to check your code worked.

I.e its basically the old base site without all the membership and editing features,

The first step is to create a new Lambda function that will connect to the `dragon_stats_index` instead of the *old* table/index. Therefore the code will look very similar as before.

1. Open the SDK docs and find the method for querying DynamoDB

Language	AWS documentation deep link
NODE.JS (8.16.0)	https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#query-property

It's time to write some code that queries your single tables index.

1. Close down any Cloud 9 tabs you are no longer using.

2. Open up the `index_query.js` file inside the `lab8` folder by double clicking on it.

3. Have the SDK docs open (as above) to help you

    1. Replace the sections of the code in that file, so that the code queries from the new `dragon_stats_index`.
    2. For lab brevity you don't need to integrate Xray right now.

4. Save the file.

5. Go to the terminal and run your file using the respective run command below

```
node index_query.js test "Fireball"
```

This shoudl trigger the `justOneDragon` method and you should see something like this:

```
Local test for a dragon called Fireball
null [ { family: { S: 'red' },
    damage: { N: '2' },
    description:
     { S: 'Fireball is a young dragon in training. He is learning how to control his fire,
but is still lethal.' },
    pk: { S: 'd5ec708a-65d0-423c-a7c6-8523bf5a0572' },
    protection: { N: '6' },
    dragon_name: { S: 'Fireball' } } ]
```

Now try a scan (i.e find all dragons) - *uses the `scanIndex` method*

```
node index_query.js test "All"
```

```
 .......
 ....... (many records ^^^

  { family: { S: 'green' },
    damage: { N: '3' },
    description:
     { S: 'Jerichombur is a dragon of mischief. His earth crushing roar can be heard for
miles.' },
    pk: { S: '1e71d947-7d89-4a59-8272-0cc4fb39bfc9' },
    protection: { N: '5' },
    dragon_name: { S: 'Jerichombur' } } ]
Total items scanned: 48
```

Did you notice how the code is much simpler than what you had before? No `ExpressionAttributeNames` required, no `ProjectionExpression` required. So this is another benefit of the single table concept and being smart with indexes. Less code... #winning

Real world tip: The only thing is, PK comes back too, but that is easy enough to ignore and is only a few bytes anyway. However you might be glad you got this returned anyway, as is allows you to potentially to do very fast item lookups using the table directly if the situation needs it.

We just need to create a lambda function out of this code that we just tested. So we will use the same code and steps (almost) as you did last time to publish the lambda function.

Step 3A): Create the new Lambda function with your new `index_query` code

You already have your new and improved `index_query` script ready and tested, so all you need to do now is create a Lambda function out of that code.

You are not creating this inside the VPC like in lab 6, as this is just an experimental proof of concept.

Once Mary and her team are all sold on the idea of the single table, they can always move it into the VPC and add XRAY (like in lab 4 and 6), along with all the other features such as membership and editing.

Your code in the `index_query` script was set up to work both in a Cloud9 testing environment and also within a Lambda environment. You do not need to alter the code.

You will create a Lambda function, passing in that role you just created back in lab 2 called `call-dynamodb-role`, then paste in your code "as is" from your new `index_query` script.

TIP: You would normally use the principle of least privilege and remove things like access to S3 and so forth, but this is just an experiment for you and Mary, so its just faster and easier for you to just use the role you already created earlier.

Follow these steps to publish your code as a lambda function:

1. Choose **services** and search for **lambda**.
2. Choose **lambda** from the drop-down list.
3. Choose **Create function**.

4. Type in `ImprovedDragonSearch` for the **Function name**.
5. Use **Node.js 10x** for the **Runtime**.
6. Under **Permissions** choose **Choose or create an execution role**.
7. Under **Execution role** choose **Use an existing role**.
8. In the **Existing role** drop-down choose the role we created above `call-dynamodb-role`.
9. Finally choose **Create function**. *Ignore any warnings.*
10. Copy and paste the code from your `index_query.js` file into the code editor replacing the contents of `index.js`.
11. Under **Basic settings**. Change the timeout to `10` sec.
12. Choose **Save**.
13. Configure a test event called `Fireball` in the usual way

```
{
    "dragon_name_str": "Fireball"
}
```

Under Details you shoud see:

```
[
  {
    "family": {
      "S": "red"
    },
    "damage": {
      "N": "2"
    },
    "description": {
      "S": "Fireball is a young dragon in training. He is learning how to control his fire,
but is still lethal."
    },
    "pk": {
      "S": "d5ec708a-65d0-423c-a7c6-8523bf5a0572"
    },
    "protection": {
      "N": "6"
    },
    "dragon_name": {
      "S": "Fireball"
    }
  }
]
```

14. Now try creating a new test event called `All`

```
{
    "dragon_name_str": "All"
}
```

View the details section, and you should now see **all** the dragon info.

Remember we are not validating like before, where we protected the cards from valid sessions. This is just a proof of concept to show Mary the single table idea.

Next we need to create the **API Gateway** resource and point your new Lambda function.

Step 3B): Create a new resource in the DragonSearchAPI

1. Go to **API gateway** and select your **DragonSearchAPI**.
2. Select Resources
3. Select the `/` so it is highlighted.
4. Choose **Actions** and **Create Resource**.
5. Leave **CORS** and **configure as proxy** UNCHECKED.
6. Call the resource `single` (lowercase).
7. Choose **Create Resource**.
8. Select `/single` and choose **Actions** and **Create Method**.
9. Choose **POST** and click the checkmark.
10. Under Lambda Function type in `imporvedDraagonSearch`.
11. Set the **timeout** to `10000` seconds.
12. Choose **Save**.
13. Click **OK** to bypass the permissions pop-up.
14. Choose **TEST** and leave the **Request Body** blank.

You should see all dragons data returned (without location info).

Test it one more time with this in the **Request Body**:

```
{
    "dragon_name_str": "Fireball"
}
```

You should see

```
[
  {
    "family": {
      "S": "red"
    },
    "damage": {
      "N": "2"
    },
    "description": {
      "S": "Fireball is a young dragon in training. He is learning how to control his fire,
but is still lethal."
```

```
    },
    "pk": {
      "S": "d5ec708a-65d0-423c-a7c6-8523bf5a0572"
    },
    "protection": {
      "N": "6"
    },
    "dragon_name": {
      "S": "Fireball"
    }
  }
]
```

15. Now select `/single` so it is highlighted and choose **Actions** and **Enable CORS**.
16. Select the **DEFAULT 4XX** and **DEFAULT 5XX** checkboxes and press **Enable CORS and replace existing CORS headers**.
17. Choose **Yes, replace existing values** (if that message appears).
18. Choose **Actions** and choose **Deploy API**.
19. Choose `prod` and choose **Deploy**.
20. Visit your S3 site and append `index5.html` to the end of the link.

Is the site is working as expected (without the login and editing features)?

If so, **congrats** you are now running the all the dragon queries using the improved single table version.

PART 2 - The real single table concept

Ok, so this is great. You are able to show Mary a rough proof of concept without affecting what she already has with the old site.

Everything works the same (slightly more efficient this time though), but now its time to *really experiment* with the single table concept, as really all you have right now is a reconstructed version of the old `dragon_stats` table. *That was already single table really right ;)*

To make this a **true** `single table`, we need to consolidate all the other dragon data tables into it, so we have 1 big table instead of many little ones. Which I am sure would be an easy sell when talking to Mary and her developers.

You also want to be sure that this concept will also allow them to query the dragon data in interesting ways, as that is what is really going to convince them to migrate to a `single table`.

So let's think about the schema again.

Here is what we have currently.

PK (PK)	Location	
	<Location_country_value>:<Location_city_value>: <Location_neighborhood_value>	dragon_name:, protection:, damage:, description:, family:

On the live site we have a total of 4 tables with dragon data. However one of the tables is the odd one out. It is the one called `current_power_table` and it's going to be an in-game session table.

It looks a bit like this:

game_id (PK)	dragon_name	current_will_not_fight_credits	current_endurance (dynamic)
56syjdh8756	Cassidiuma	2	8

This table is different that the others that contain actual dragon data. This is a table of in-game progress and is designed to keep track of current games and sessions. This will be a high activity table with many users and it makes much more sense to have this one extracted into (or left alone as) its own table, just like it makes sense to keep a user table and a session table separate.

Real world tip, Some people get obsessive about making a single table contain every thing. Thats a lot of hard work and I would not recommend you go that route. Sometimes having an occasional extra table when appropriate helps you stay sane.

The *actual dragon data* is really scoped to the 3 tables below. So we will disregard the `current power table` one for now, and see if we can consolidate these 3 into 1 single table (*#win*) and at the same time offer a way to do advance queries (*#double_win*).

Here are the 3 tables we had:

DragonStats

dragon_name (PK)	damage	description	protection	family	location	location_city	loc
Cassidiuma	7	*Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior.*	10	red	usa	las vegas	nev

We already have done that one :). So one down, two to go

It now looks more like this, with `UUID`s the Primary Key and `location` consolidated (for advanced querys later).

single_dragon_table

PK (PK)	dragon_name	damage	description	protection	family	location
c38af3d5-8c03-4beb-bcdb-5c4942cbc683	Cassidiuma	7	*Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior.*	10	red	usa#kansas#c st

So what about the other two tables that currently look like this?

DragonBonusAttack

breath_attack(PK)	description	extra_damage	range (SK)
acid	spews acid	3	5
electricty	bolts fly from mouth	3	5
water	high pressure jet over a large area	1	10
fear	Prevent all attacks next round	0	4
fire	Short blast of fire	8	4

DragonFamily

breath_attack	damage_modifer	description	family (PK)	protection_modifier
acid	-2	Better defense	green	2
fire	2	Attacks faster	red	-2
water	1	Happy in water	blue	-1
fear	0	Prefers to bite	black	0

The question is how do we bring these 2 into our `dragon_single_table` and create indexes that will allow the game developers (later on) to query the data in sane ways?

The first thing you should always do when working with DynamoDB Schema design is to **think about he queries first!**

So let's think of a few things that the site would maybe like to do in future iterations.

Here are some ideas:

1. Return all the dragons that can spew acid.
2. Return all dragons that are green.
3. Return the dragons in range attack (order).
4. Give me all the dragons that live in Arizona, USA (Dragons like the warmth apparently).

Let's address each of these one at a time and discuss what we would normally have had to do with say a traditional database or where we had multiple tables in DynamoDB.

Part 2 - Step 1: Return all the dragons that can spew acid

If you had a traditional DB this would normally require looking up information in the `DragonBonusAttack` table first a bit like so:

```
SELECT breath_attack FROM DragonBonusAttack WHERE description = "spews acid"
```

Then finding the family such as `red` (family).

```
SELECT family FROM DragonFamily WHERE breath_attack = breath_attack
```

Then finally:

```
SELECT * FROM DragonStats WHERE family = family
```

Even in DynamoDB where we have multiple tables that would require a very similar approach.

Using the single table approach we can do better than that.

All we need to do is add items using the same primary key as follows:

PK (PK)	**SK (SK)**	**Location**	**`<attributes>`**
`<SAME-UUID>`	stats	<Location_country_value>: <Location_state_value> <Location_city_value>: <Location_neighborhood_value>	dragon_name:`<value>`, protection: `<value>`, damage:, description:`<value>`, family:`<value>`
`<SAME-UUID>`	bonus		extra_damage:`<value>`, range:`<value>`, bonus_description: `<value>`
`<SAME-UUID>`	family		breath_attack:`<value>`, damage_modifier:`<value>`, protection_modifier:`<value>`, family_description: `<value>`

The first time you seee the single table pattern it's mindbending I know, but bear with us..

Notice we had to change the attribute of `description` to `family_description` to not conflict with the description attribute used for dragon stats, and we also did this for the bonus table's `description` for the same reason.

Real World Tip: We could use description as an overloaded attribute, which is very useful for reducing the amount of GSIs needed to do certain queries, refer to the course notes.

Notice that we are adding a Primary Key with the same value representing one dragon.

This might look very strange at first, but this is correct for what we are trying to do.

Normally you can't have a duplicate primary key, unless you have a Sort Key.

So as you can see above, we now have a Sort Key called `SK` and are assigning three string values to each of the 3 items. One called `stats`, one called `bonus` and one called `family`. These are strings of the words, and not variables. i.e the strings **"family"**, "**bonus**", and "**stats**".

The old family table had a `family` value, which we removed, because it is already there in the `stats` item for this dragon. We are also taking away `breath_attack` out of `bonus` as it is already there in the `family` item.

We could have just duplicated it but it seems unnecessary, as none of our queries that we are envisioning would need it. **Hence the value of thinking about queries first**. It makes it a *little less flexible* in case you want to do ad-hoc queries. However that is the compromise you make by using this approach.

So the prior queries still work as expected, we haven't messed anything up there ;). All we have really done here is added a couple of new items with the same UUID assigning a different SK to each one and removed some duplication.

So before we go ahead and create an improved single table with this new structure, it is always better to measure twice and cut once.

So let's really think about our queries some more and figure out if we are going to need any GSIs.

It is going to be easier to do all that, in one go.

So query number 1) We want to **Return all the dragons that can spew acid**.

Let's think about that.

This needs to be done in two stages.

Stage 1) Get all the UUID of all the dragons that can spew acid.

This will require an index which we could call `bonus_description_index` with `bonus_description` as the Primary Key.

Stage 2) provide the UUIDs to a new scan that can return the dragon details (stats).

This will require exactly the same index as we created before called `dragon_stats_index`. So we will definitely keep that as a GSI moving forward.

Even without coding any of this, we can see that we have a sensible enough schema and GSI to accommodate both of these actions.

So we can move on. We will do the code for this stuff in a bit, but for now, let's continue conceptualizing our other queries to make sure that **a)** they are all possible and **b)** if we are going to need some more GSIs .

Part 2 - Step 2: Return all dragons that are green.

Let's see if this is possible using what we already have proposed, or if we need to modify our proposed table again.

It turns out that the `family` e.g `green` is easily accessible in our single table on the item that has the Sort Key of `stats`. Remember we removed it from the other items as it was not needed.

If would be as simple as this

```
aws dynamodb scan --index-name dragon_stats_index --table-name single_dragon_tables  --
filter family = "green"
```

Ok, I like that one, it's simple and it will work using an existing index with a `scan and filter`.

Real World Tip: Some people get hung up on only using queries and never using scans, however think about this one: We have 48 cards, so 12 of them will be any color, we are essentially scanning and throwing away 36 cards from the scan, this is not something you should concern yourself with. For large scale or high volume apps where scans can become kind of unwieldy or expensive, sure, optimize further, create GSI, but for this one (and many real world situations) `scan and filter` *is just fine. #evils_of_early_optimization*

That's 2 out of 4 conceptualized, let's keep going.

Part 2 - Step 3: Return the dragons in range attack order

Hmm, this one is a little more interesting as to brings something back in order (or a reverse order). For this we really need to be using number attributes on the Sort Key.

The problem is our Sort key is taken, *Doh* however we can create a GSI with `range` as the Sort Key

Proposed `range_index`

SK(PK)	**range (sk)**	**`<attributes>`**
bonus	5	PK : `<UUID>`

Note that the PK is the old SK. This is a reverse index, and super useful for look-ups in a single table.

Although you could just scan the table, that won't give you the range order. You need to be able to specify what we call the `ScanIndexForward` key to true (default - ascending) or false (defending).

As we want descending, we need to specific `ScanIndexForward` as `false` and you need to use a query for that, as it is not an option for a scan. Which means proving a Primary key, which is luckily the value "bonus".

I hope you are starting to see the value of single tables already.

So query 3 is super easy, we just need a simple `range index`.

Part 2 - Step 4: Give me all the dragons that live in Arizona, USA

And finally (thank goodness I hear you say) for query number **4 - Give me all the dragons that live in Arizona, USA**

We recently removed all the location info, so you can't do the following anymore

```
aws dynamodb scan --index-name dragon_stats_index --table-name single_dragon_tables --filter
location_state = "arizona"
```

This would be a horribly inefficient scan anyway, as there are many dragons *not* in Arizona.

You might be thinking that you could have set up a query on location_state..had we not removed it ;)

However good news!

Remember earlier we created a **composite** attribute like this.

```
usa#arizona#flagstaff#lake mary rd
```

All we need to do is create a **reverse GSI** called `location_index` with `SK` as the Primary Key, and just like you did with the stats index, *project* the dragon info that you need returning, like the following:

Proposed location_index

SK (Pk)	Location (SK)	
stats	<Location_country_value>: <Location_city_value>: <Location_neighborhood_value>	dragon_name:, protection:, damage:, description:, family: *[uuid-PK added by default]*

Note you made the Location the Sort Key, and this is KEY excuse the pun!

It is critical that the new Sort Key is present so we can leverage the features of DynamoDB that allow us to place **functional conditions** in our queries.

Look how cool this query is using the single table concept.

Super efficient!

```
aws dynamodb query --index-name location_index --table-name single_dragon_tables --
keycondition location = starts_with("usa#arizona")
```

Not powerful enough? How about give me every dragon spotted (near me) in Spring Valley in Las Vegas?

```
aws dynamodb query --index-name location_index --table-name single_dragon_tables --
keycondition location = starts_with("usa#nevada#las vegas#spring valley")
```

#win

Wow, you got through all of the concepts, now it's time to code it!

Before you go to Marry with the new `index5.html` to show her the base functionality can be easily replicated and simplified using singe table pattern, you really want to be able to show here these queries and not just give her an explanation of how it would work.

In this final part of this lab *(and the course actually)* you will do the following:

1. Create a **new** single table (singe_table_improved) based on the schema we just discussed, ensuring we have all the GSIs we need for all our potential queries.
2. Seed it correctly with data, similar to before, just accommodating the new schema.
3. Create a few `.js` files that you can use to show example queries.
4. Hand everything over to her devs and let them take it from here, so you can get your chocolate cake off Mary for all your hard work .

Part 2 - Step 5: Create a new single improved table

1. Open `create_improved_single_table_and_all_indexes.js`
2. Edit the s like normal.
3. Run it

```
node create_improved_single_table_and_all_indexes.js
```

You should see:

```
null { TableDescription:
   { AttributeDefinitions: [ [Object], [Object], [Object], [Object], [Object], [Object] ],
     TableName: 'improved_single_dragon_table',
     KeySchema: [ [Object], [Object] ],
     TableStatus: 'CREATING',
     CreationDateTime: 2019-06-05T17:03:57.138Z,
     ProvisionedThroughput:
      { NumberOfDecreasesToday: 0,
        ReadCapacityUnits: 0,
        WriteCapacityUnits: 0 },
     TableSizeBytes: 0,
     ItemCount: 0,
     TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/improved_single_dragon_table',
     TableId: 'c28b30ad-8ef9-4b95-a1e5-e83a5299fc81',
```

```
    BillingModeSummary: { BillingMode: 'PAY_PER_REQUEST' },
    GlobalSecondaryIndexes: [ [Object], [Object], [Object], [Object] ] } }
```

⚠ This may take some time, do not move on until that table is ACTIVE, or part 2 step 6 will fail. You might want to grab some coffee, it's annoyingly slow.

Part 2 - Step 6: Seed it correctly with data

1. Open `seed_improved_table.js`

2. Edit the s like normal.

3. Run it:

```
node seed_improved_table.js #just once please
```

You should see this:

```
seeded
```

Part 2 - Step 7: Create a few `.js` files that you can use to show example queries.

1. Open `query_1.js` (Return dragons that can [spew acid]).

2. Edit the s like normal.

3. Run it:

```
node query_1.js test "spews acid"
```

You should see this. *We just projected the dragon name for brevity.*

```
24 dragons found that spews acid
null [ { dragon_name: { S: 'Nightingale' } },
  { dragon_name: { S: 'Castral' } },
  { dragon_name: { S: 'Bahamethut' } },
  { dragon_name: { S: 'Magnum' } },
  { dragon_name: { S: 'Ragnorl' } },
  { dragon_name: { S: 'Pradumo' } },
  { dragon_name: { S: 'Sheblonguh' } },
  { dragon_name: { S: 'Blackhole' } },
  { dragon_name: { S: 'Midnight' } },
  { dragon_name: { S: 'Dexler' } },
  { dragon_name: { S: 'Shadow' } },
  { dragon_name: { S: 'Sonic' } },
  { dragon_name: { S: 'Cassidiuma' } },
  { dragon_name: { S: 'Mino' } },
```

```
  { dragon_name: { S: 'Smolder' } },
  { dragon_name: { S: 'Amaron' } },
  { dragon_name: { S: 'Samurilio' } },
  { dragon_name: { S: 'Prythus' } },
  { dragon_name: { S: 'Shulmi' } },
  { dragon_name: { S: 'Warcumer' } },
  { dragon_name: { S: 'Tornado' } },
  { dragon_name: { S: 'Lucian' } },
  { dragon_name: { S: 'Havarth' } },
  { dragon_name: { S: 'Jerichombur' } } ]
```

1. Open `query_2.js` *(Return all dragons that are [green])*.

2. Edit the s like normal.

3. Run it.

```
node query_2.js test "green"
```

You should see this: *We just projected the dragon name for brevity.*

```
null [ { dragon_name: { S: 'Ragnorl' } },
  { dragon_name: { S: 'Pradumo' } },
  { dragon_name: { S: 'Cassidiuma' } },
  { dragon_name: { S: 'Amaron' } },
  { dragon_name: { S: 'Samurilio' } },
  { dragon_name: { S: 'Prythus' } },
  { dragon_name: { S: 'Shulmi' } },
  { dragon_name: { S: 'Warcumer' } },
  { dragon_name: { S: 'Havarth' } },
  { dragon_name: { S: 'Jerichombur' } } ]
```

1. Open `query_3.js` *(Return the dragons in range attack order - highest first)*.

2. Edit the s like normal.

3. Run it.

```
node query_3.js test
```

You should see this: *(note we added `range` to each item sent back).*

```
....more items...
....
dragon_name: { S: 'Isilier' } },
  { pk: { S: '2bf1f72f-90b9-4695-a9ad-c582dfae252e' },
    range: { N: '4' },
    damage: { N: '7' },
    protection: { N: '9' },
    family: { S: 'red' },
    description:
     { S: 'Ruby has a skin and coat that\'s as hard as gems. This gives her extra defense
against her enemies.' },
    dragon_name: { S: 'Ruby' } } ]
Dragons in range order:
```

1. Open `query_4.js` (*Give me all the dragons that live in Arizona, USA*).

2. Edit the s like normal.

3. Run it.

```
node query_4.js test "usa#arizona"
```

You should see this:

```
null [ { location: { S: 'usa#arizona#chandler#w german rd' },
    damage: { N: '7' },
    sk: { S: 'stats' },
    family: { S: 'green' },
    description:
     { S: 'Ragnorl is a rogue dragon, disowned from his own tribe. He can change colors to
blend with the earth around him.' },
    pk: { S: '04e39412-ed8e-41ea-b7f8-e5d6ff73c002' },
    protection: { N: '7' },
    dragon_name: { S: 'Ragnorl' } },
  { location: { S: 'usa#arizona#flagstaff#lake mary rd' },
    damage: { N: '9' },
    sk: { S: 'stats' },
    family: { S: 'black' },
    description:
     { S: 'Sonic has black spikes that can penetrate his enemies. He has a spiked tail that
can attack his opponents.' },
    pk: { S: 'dbe13c27-2e85-4480-aae8-e003e00cad31' },
    protection: { N: '8' },
    dragon_name: { S: 'Sonic' } },
  { location: { S: 'usa#arizona#mesa#e adobe st' },
    damage: { N: '6' },
    sk: { S: 'stats' },
    family: { S: 'blue' },
    description:
     { S: 'Frealu has an ice breath that can freeze her enemies into a paralyzed state. She
is from the souther water tribe.' },
```

```
    pk: { S: 'e21bb786-d240-416a-ac12-7f65fced089b' },
    protection: { N: '6' },
    dragon_name: { S: 'Frealu' } },
  { location: { S: 'usa#arizona#page#morgan rd' },
    damage: { N: '8' },
    sk: { S: 'stats' },
    family: { S: 'blue' },
    description: { S: 'Always hiding in shadows' },
    pk: { S: '3c1ce221-6df1-417c-bb17-8b2172d37407' },
    protection: { N: '6' },
    dragon_name: { S: 'Sanguia' } },
  { location: { S: 'usa#arizona#tempe#e laguna dr' },
    damage: { N: '4' },
    sk: { S: 'stats' },
    family: { S: 'red' },
    description:
     { S: 'Firestorm can summon a fire storm of hail and rain, that burns his opponents.' },
    pk: { S: 'a60085d4-84dd-4859-9bf2-2b27c3519751' },
    protection: { N: '9' },
    dragon_name: { S: 'Firestorm' } } ]
```

Congrats you can now show Mary these new queries from your laptop, and convince her about the benefits of using the single table pattern.

You're' done!

Oh wait, there's one more **critical** task to complete...Text Mary and get her to bring over that chocolate cake!

We hope you had lots of fun working with DynamoDB in a web application setting, and learned a few new things too.

Congrats on sticking with it, there was a lot in here!

....See you next time