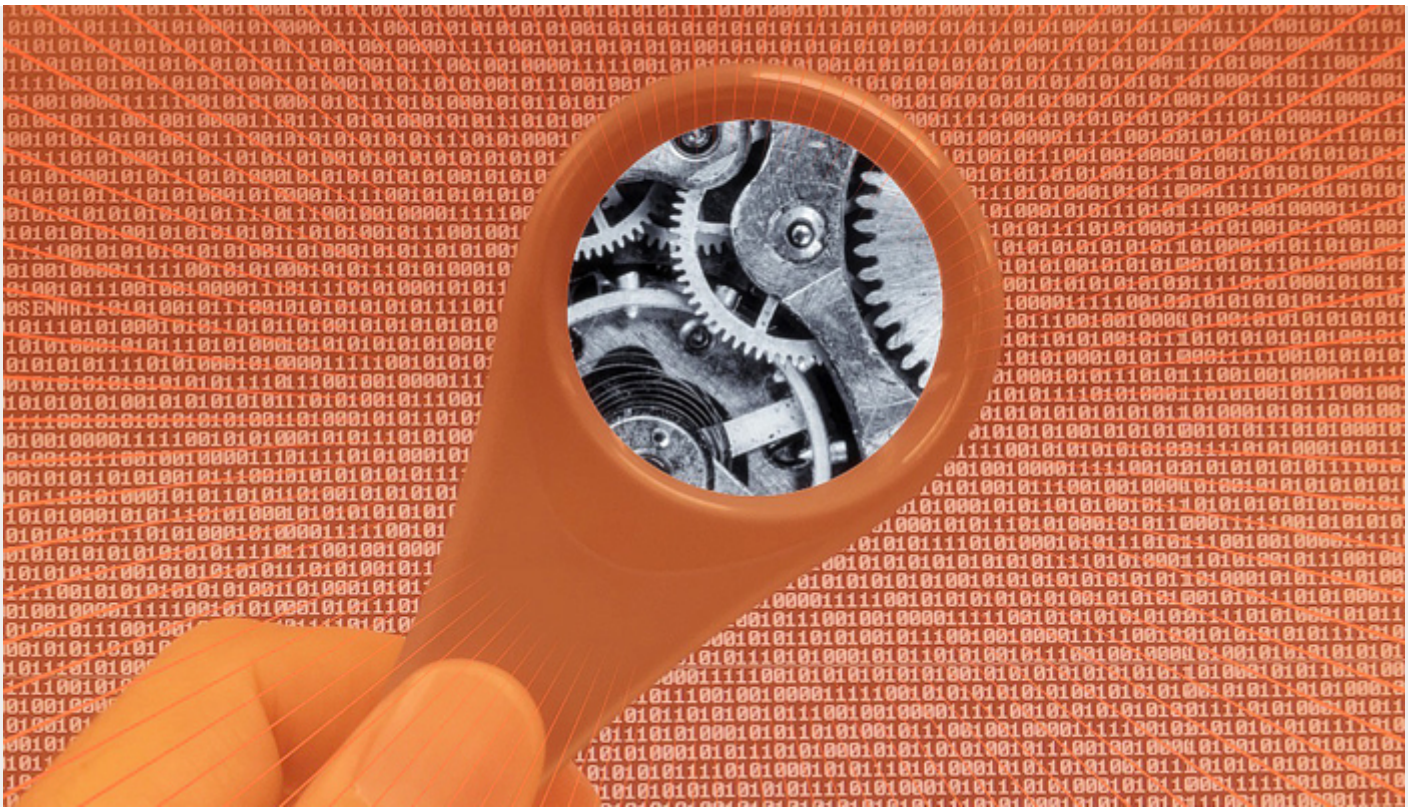
HOW TO QUERY AMAZON DYNAMODB



In this article, we explain the basics of DynamoDB queries. The query syntax is quite different from other databases, so it takes some time to get used to it.

Hash key in DynamoDB

The primary reason for that complexity is that you cannot query DynamoDB without the hash key. So, it's not allowed to query the entire database. That means you cannot do what you would call a **full table scan** in other databases.

However, the primary (partition) key can include a second attribute, which is called a **sort** key. The key query condition must be = (equals). But operators for the sort key can be:

-
-
- _
- =
- =
- **begins_with**
- **between**

Each query has two distinct parts:

1. The key condition query (i.e., the partition key hash) and optionally the sort key

2. The filter expression (whatever query other attribute you want)

Load sample data

We give some examples below, but first we need some data:

1. Install DynamoDB and run it locally, as we explained in How To Add Data in DynamoDB.
2. Install node so we can run some JavaScript code.
3. Install the Amazon SDK using npm, which is part of node:

```
npm install aws-sdk
```

Run these programs from the Amazon JavaScript examples:

1. Create the Movies table by running MoviesCreateTable.js.
2. Download the sample data from here and unzip it.
3. Load some data by running MoviesLoadData.

Inspect the data

Take a look at the top of the data file. It is a movie database that includes nested JSON fields and arrays. So, it is designed to be used as a teaching exercise.

```
{
        "year": 2013,
        "title": "Rush",
        "info": {
            "directors": ,
            "release_date": "2013-09-02T00:00:00Z",
            "rating": 8.3,
            "genres": ,
            "image_url":
"http://ia.media-imdb.com/images/M/MV5BMTQyMDE0MTY0OV5BMl5BanBnXkFtZTcwMjI2OT
I0OQ@@._V1_SX400_.jpg",
            "plot": "A re-creation of the merciless 1970s rivalry between
Formula One rivals James Hunt and Niki Lauda.",
            "rank": 2,
            "running_time_secs": 7380,
            "actors":
        }
    }
```

Describe table

You can verify that the data was loaded using:

```
aws dynamodb describe-table   --table-name Movies   --endpoint-url
http://localhost:8000
```

Here is the first record.

```
{
    "Table": {
        "TableArn": "arn:aws:dynamodb:ddblocal:000000000000:table/Movies",
        "AttributeDefinitions": ,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 10,
            "LastIncreaseDateTime": 0.0,
            "ReadCapacityUnits": 10,
            "LastDecreaseDateTime": 0.0
        },
        "TableSizeBytes": 2095292,
        "TableName": "Movies",
        "TableStatus": "ACTIVE",
        "KeySchema": ,
        "ItemCount": 4608,
        "CreationDateTime": 1595056037.797
    }
}
```


Query structure

We will use the Amazon CLI command line interface to execute queries. If you are using any of the programming language SDKs, the principles are the same. The only part that varies is the syntax.

Queries are composed of two parts:

1. Key condition expression
2. Filter expression

Key condition expression

The key condition expression can contain the partition key and, optionally, the sort key. This primary key is what DynamoDB calls the **partition key**. You can also have a second key, which is called the **sort key**. In the movies database, the partition key name is **year** and the sort key is the **title**. The partition key query expression only allows equals to (=). The sort key allows

-
-
- _
- =
- =
- begins_with
- between

The Amazon sample data includes a reserved word, **year**. So, we have to deal with that complexity up front.

The query is below. We use backslashes (\) after each line because that's the continuation character

in the bash shell. And we put a single quote (') around JSON for the same reason, so that it can span lines.

Here is the query:

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name Movies    \
    --key-condition-expression "#yr = :yyyy"     \
    --expression-attribute-names '{"#yr": "year"}'     \
   --expression-attribute-values  '{ ":yyyy":{"N":"2010"}}'
```

Let's break down each part of the query:

--key-condition-expression #yr = :yyyy	The format of this expression is: **partition key name = placeholder** And you could add a sort key, which for this database is **title**. But we don't want it here as we are looking for a title and not searching by one. The pound (#) sign means that we will redefine that partition key field name in the parameter **expression-attribute-names** because it is a reserved word. Usually you just put the field name. But you cannot use **year** as it is a reserved word. The colon (:) is a placeholder as well. It means we will redefine that below in the **key-condition-expression**
--expression-attribute-names '{"#yr": "year"}'	This is where we provide an alias for the field year as **yea**r is a reserved word, meaning you can't use it as a field name.
--expression-attribute-values '{ ":yyyy":{"N":"2010"}}'	Think of this as expansion or definition of the placeholder used in the key or filter expressions (We will get to filter expressions below). In other words, in the key condition we wrote: **:yyyy .** That's just a temporary placeholder. We define what value that holds here '{ ":yyyy":{"N":"2010"}}' The **N** means it is a number. **S** is string. **B** is binary. **2010** is the targeted value.

Filter expression

Remember that the key condition serves to select a set or records based upon their partition. It's a two-step process of pulling a subset of the database and then querying that subset in the filter expression.

In the example below, we want to show all films in the year 2010 that have a rating higher than 8.5. The (1) key condition gets the year and (2) filter expression lets you query by rating all movies from the year 2010. It's designed this way for speed, by reducing the amount of data to query.

Here is the query:

```
aws dynamodb query  \
```

```
     --endpoint-url http://localhost:8000 \
     --table-name Movies    \
    --key-condition-expression "#yr = :yyyy"     \
    --expression-attribute-names '{"#yr": "year"}'     \
   --expression-attribute-values  '{ ":yyyy":{"N":"2010"}}' \
 --filter-expression 'info.rating > :rating' \
--expression-attribute-values '{
   ":yyyy":{"N":"2010"},
    ":rating": { "N": "8.5" }
}'
```

The filter expression has the same syntax as the key condition, but there are a couple of items to note.

--filter-expression 'info.rating > :rating' \	The filter expression, like the key condition, is just an attribute on the left, operator in the middle, and placeholder on the right. In other words, it's not JSON, whereas we use JSON elsewhere. **rating** is nested underneath **info**. So, we write **info.rating**.
--expression-attribute-values '{ ":yyyy":{"N":"2010"}, ":rating": { "N": "8.5" } }'	In the key condition query above we used the exact same parameter. Here we use JSON syntax: --expression-attribute-values '{ ":yyyy":{"N":"2010"}}' The only different here is we now have two placeholders (:yyyy and :rating) so need two lines in that JSON: --expression-attribute-values '{ ":yyyy":{"N":"2010"}, ":rating": { "N": "8.5" } }' This also illustrates the point that the filter expression must always include the same query key condition. So, you repeat it.

Additional resources

For more on this topic, explore the BMC Big Data & Machine Learning Blog or check out these resources:

- AWS Guide, with 15+ articles and tutorials on AWS
- Availability Regions and Zones for AWS, Azure & GCP
- Databases on AWS: How Cloud Databases Fit in a Multi-Cloud World
- An Introduction to Database Reliability