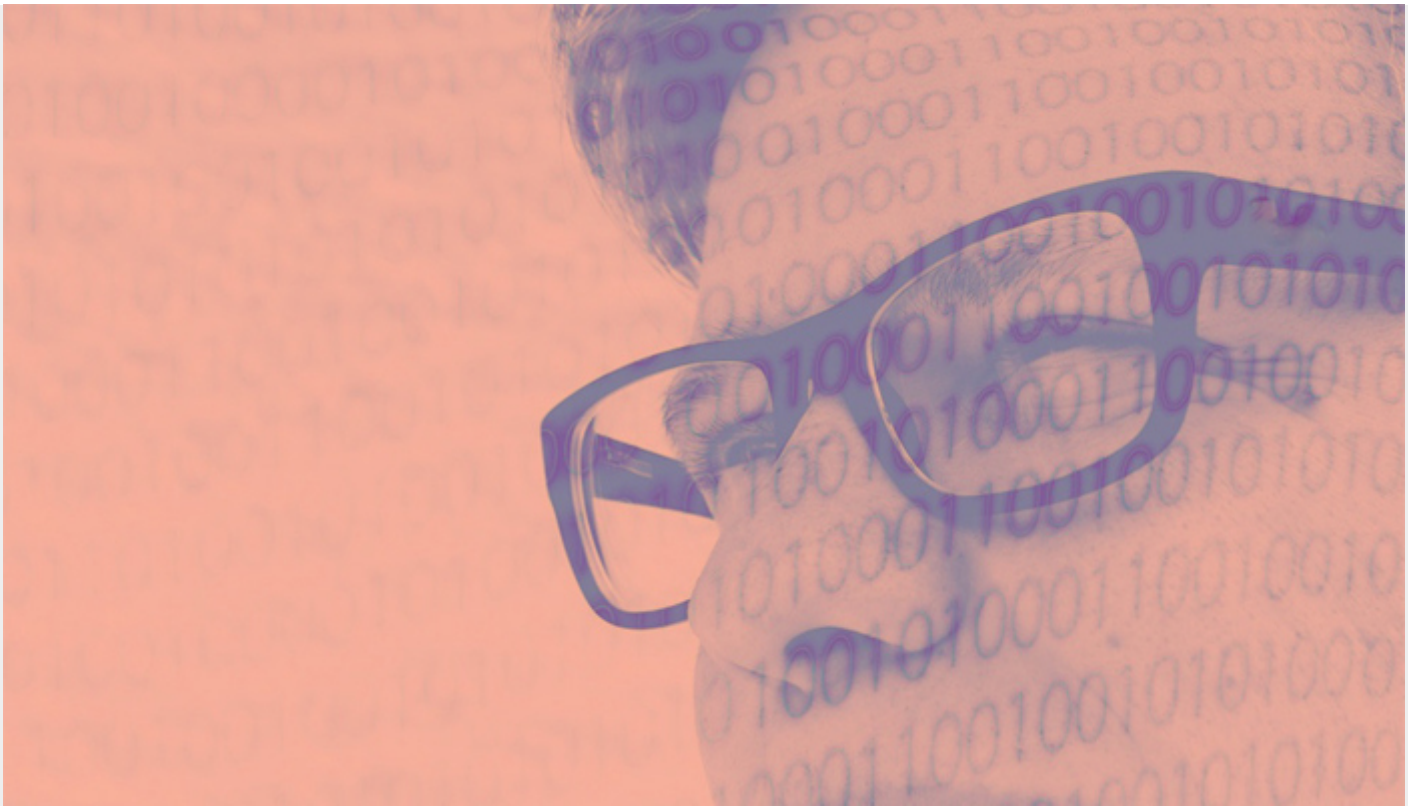
DYNAMODB ADVANCED QUERIES: A CHEAT SHEET



This is an article on advanced queries in Amazon DynamoDB and it builds upon DynamoDB basic queries.

(This tutorial is part of our DynamoDB Guide. Use the right-hand menu to navigate.)

DynamoDB Query Rules

Remember the basic rules for querying in DynamoDB:

- The query includes a key condition and filter expression.
- The key condition selects the partition key and, optionally, a sort key.
- The partition key query can only be equals to (=). Thus, if you want a compound primary key, then add a sort key so you can use other operators than strict equality.
- Having selected a subset of the database with the key condition, you can narrow that down by writing a filter expression. That can run against any attribute.
- Logical operators (>, <, begins_with, etc.) are the same for key conditions and filter expressions, except you cannot use **contains** as a key condition.

Load sample data

To perform these advanced queries, we need some data to work with. Download this sample data from GitHub, which is data from IMDB that I've slightly modified.

Create a table

In this document we are using DynamoDB on a local machine. So, we specify **--endpoint-url http://localhost:8000**.

Create the **title** table like this:

```
aws dynamodb create-table \
--endpoint-url http://localhost:8000 \
    --table-name title \
    --attribute-definitions AttributeName=tconst,AttributeType=S \
                                    AttributeName=primaryTitle,AttributeType=S
\
    --key-schema AttributeName=tconst,KeyType=HASH \
       AttributeName=primaryTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

Notice that the primary key is the combination of the attributes **tconst** (partition key) and **primaryTitle** (sort key).

For our sample data we have data like shown below. All the partition keys are set to the same value **movie**. Then the movie **primaryTitle** is the sort key.

```
{
          "tconst": {
            "S": "movie"
          },
           "primaryTitle": {
            "S": "Travel Daze"
          },
```

Then load the data like this:

```
aws dynamodb batch-write-item \
         --endpoint-url http://localhost:8000 \
          --request-items
file:////Users/walkerrowe/Documents/imdb/movies.json \
         --return-consumed-capacity  TOTAL \
        --return-item-collection-metrics  SIZE
```

Between query

Here we use the first (space) and last (ÿ) characters in the UTF-8 character set to select all titles.

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst and primaryTitle BETWEEN
:fromTitle AND :toTitle" \
   --expression-attribute-values  '{
```

```
                    ":tconst":{"S":"movie"},
                     ":fromTitle":{"S":" "},
                     ":toTitle":{"S":"ÿ"}
      }'
```

Begins with query

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst and
begins_with(primaryTitle, :beginsWith)" \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"},
                     ":beginsWith":{"S":"A"}
      }'
```

Contains query

Here we write a filter expression instead of a key condition just to show how to write a filter expression as opposed to a key condition. As we mentioned above, the operators are the same, except you cannot use the operator **contains** as a key condition.

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression 'contains(originalTitle, :containsStr)' \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"},
                     ":containsStr":{"S":"Brooklyn"}
      }'
```

Attribute exists query

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression 'attribute_exists(genres)' \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"}
       }'
```

Attribute not exists query

```
aws dynamodb query  \
```

```
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression 'attribute_not_exists(genres)' \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"}
       }'
```

In query

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression "genres IN (:inDrama, :inComedy)" \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"},
                   ":inDrama":{"S":"Drama"},
                    ":inComedy":{"S":"Comedy"}
       }'
```

String set query

Our data contains data like this:

```
"actors": {
        "SS":
},
```

So, an equality condition on that string set (SS) element would necessarily contain all those strings.

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression "actors =  :actors" \
   --expression-attribute-values  '{
        ":tconst":{"S":"movie"},
         ":actors":{"SS": }
       }'
```

Boolean query

Boolean data is stored like this:

```
"isComedy": {
        "BOOL": false
}
```

So, you query it like this:

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression "isComedy = :isComedy" \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"},
                    ":isComedy":{"BOOL":true}
       }'
```

Query map type

The map query is similar to the nested query (see the next item).

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression "aditionalInfo = :aditionalInfo" \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"},
                     ":aditionalInfo": {
                  "M": {"Location": {"S": "Bost"}, "Language": {"S": "FR"}}
        }
}'
```

Nested query

A nested DynamoDB object is one that contains a map. You refer to the element using the dot notation **parent.child**, like for this data you would write **aditionalInfo.Location**.

```
"aditionalInfo": {
            "M": {"Location": {"S": "Bost"}, "Language": {"S": "FR"}}
        },
```

Location is a reserved word, so we have to give it an alias using:

```
--expression-attribute-names '{"#loc": "Location"}'
```

And here is the nested DynamoDB query:

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression "aditionalInfo.#loc = :loc" \
--expression-attribute-names '{"#loc": "Location"}'     \
   --expression-attribute-values  '{
```

```
                    ":tconst":{"S":"movie"},
                     ":loc": {"S": "Bost"}
}'
```

Projection Expression

Use this projection expression to limit the attributes returned by DynamoDB, as it returns all attributes by default.

```
aws dynamodb query  \
     --endpoint-url http://localhost:8000 \
     --table-name title    \
    --key-condition-expression "tconst = :tconst" \
   --filter-expression "aditionalInfo = :aditionalInfo" \
   --expression-attribute-values  '{
                    ":tconst":{"S":"movie"},
                     ":aditionalInfo": {
                  "M": {"Location": {"S": "Bost"}, "Language": {"S": "FR"}}
        }
}' \
--projection-expression "originalTitle, runtimeMinutes"
```

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