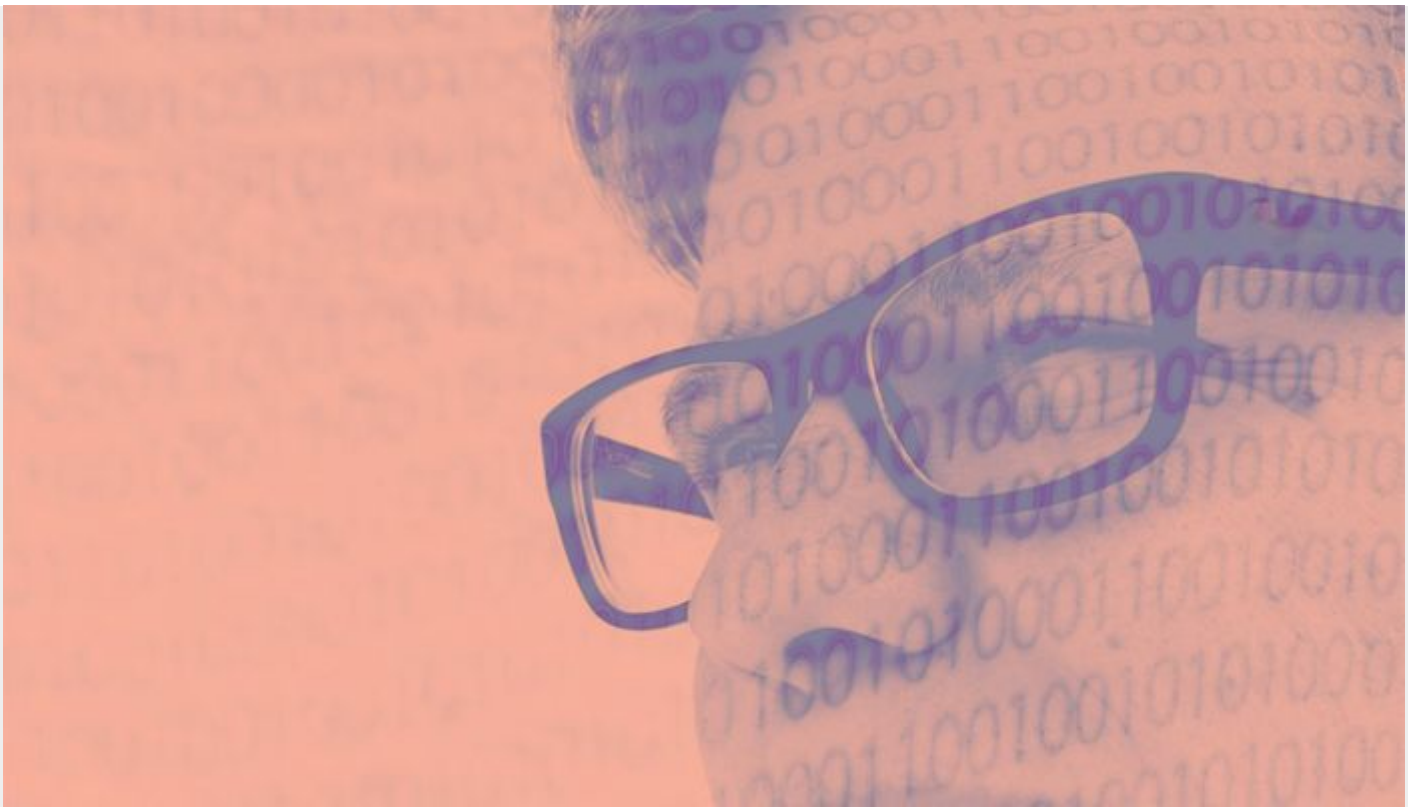
DYNAMODB COMPLEX QUERIES CHEAT SHEET



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

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

Getting Started

Remember the basic rules for querying in DynamoDB:

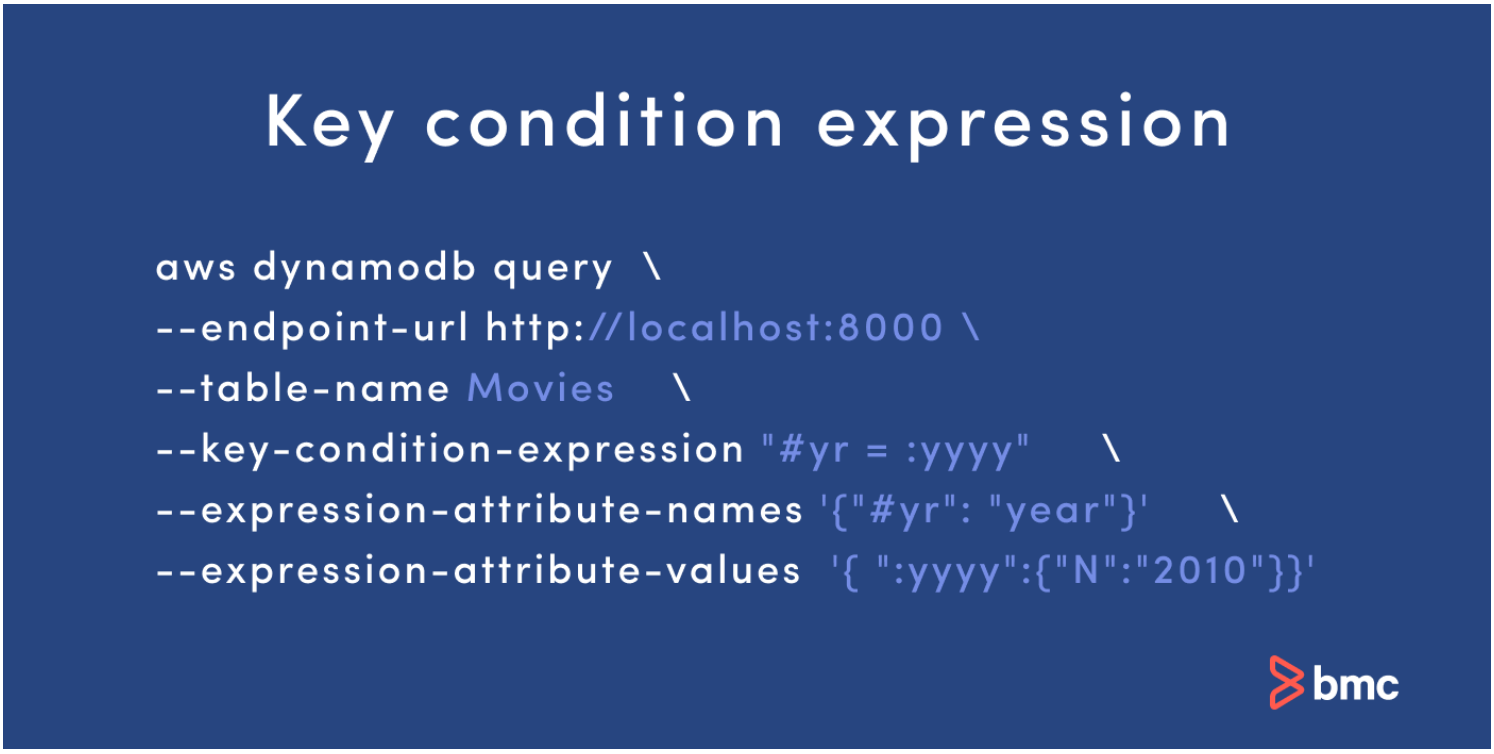
- The query includes a key condition and filter expression.
- The key condition selects the partition key and, optionally, a sort key.
- The partition key query can only be equals to (=). Thus, if you want a compound primary key, then add a sort key so you can use other operators than strict equality.
- 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.

Using DynamoDB filter expressions to refine query results

You can apply a DynamoDB filter expression after a query finishes, but before it returns results. That way, your query consumes the same amount of read capacity, which is limited to 1 MB of data.

DynamoDB filter expressions do not contain partition keys or sort key attributes—those belong in the key condition expression. The syntax, however, is like a key condition expression, using the same comparators, functions and logical operators.

What are DynamoDB key condition expressions?



Use a key condition expression, a specification of the items to be read from a table or index, to indicate search criteria. You will need to include the partition key name and value as an equality condition. You also have the option of using a second sort key condition. You cannot, however, use non-key attributes.

In writing key condition expressions, you can use the following comparison operators:

a = b — true if the attribute a is equal to the value b.

a < b — true if a is less than b.

a <= b — true if a is less than or equal to b.

a > b — true if a is greater than b.

a >= b — true if a is greater than or equal to b.

a BETWEEN b AND c — true if a is greater than or equal to b, and less than or equal to c.

You can also use this function:

begins_with (a, substr)— true if the value of attribute a begins with a particular substring.

Load sample data

To perform these complex DynamoDB 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
```

Advanced Queries in DynamoDB

Between query

Below is a DynamoDB between query example. 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

A DynamoDB begins_with query example is below.

```
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

In the DynamoDB contains query example below, we write a filter expression instead of a key condition. This is just to show how to write a DynamoDB 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

To find out whether an attribute exists, use the DynamoDB 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

To verify that an attribute doesn't exist, use the DynamoDB 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

To test a value for membership in a set, you can write an IN query or IN condition. The syntax of arguments includes a numeric, character, or datetime expression, with an expr_ist of parenthesis-bound, comma-delimited expressions, a table subquery, and, of course, an IN or NOT 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

A string set refers to specific attributes attached to items in a set of data. If you want to query items in a table of data, for example, to find all people in a table that are identified as "actors," you would write a string set query. Our query 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 DynamoDB 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 example:

```
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

When you want to read some attributes versus all attributes in a table, you can use a projection expression. It allows you to specify the items you want, according to attributes that you specify by name. You can specify multiple attributes if you separate the attribute names with a comma.

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