DynamoDB Complex

Creating Table with Composite Key

To create a table with composite key, it is the same process, except you will be adding an additional field to the `key-schema` flag:

```
aws dynamodb create-table \
    --table-name UserOrdersTable \
    --attribute-definitions '[
        {
            "AttributeName": "Username",
            "AttributeType": "S"
        },
        {
            "AttributeName": "OrderId",
            "AttributeType": "S"
        }
    ]' \
    --key-schema '[
        {
            "AttributeName": "Username",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "OrderId",
            "KeyType": "RANGE"
        }
    ]' \
    --provisioned-throughput '{
        "ReadCapacityUnits": 1,
        "WriteCapacityUnits": 1
    }' \
    $LOCAL
```

In addition, the KeyType must be of `RANGE`, you cannot have more than one `HASH` KeyType, it must be `RANGE`.

Retrieving All Items | Partition Key

If you want to query a list of items given a partition key for composite primary keys (because one partition key can map to multiple items with composite keys), you would need to use the `query` operation.

```
aws dynamodb query \
  --table-name UserOrdersTable \
  --key-condition-expression "Username = :usr" \
  --expression-attribute-values '{
      ":usr": {"S": "daffyduck"}
  }' \
  $LOCAL
```

`get-item` only works if you have the full key, in this case we are only giving it the partition key not the sorting key, hence it will not work.

Filtering

You can filter your result by your sorting key by using logical and built-in operators like below:

```
aws dynamodb query \
    --table-name UserOrdersTable \
    --key-condition-expression "Username = :usr AND OrderId BETWEEN :startdate AND :enddate" \
    --expression-attribute-values '{
        ":usr": {"S": "daffyduck"},
        ":startdate": {"S": "20170101"},
        ":enddate": {"S": "20180101"}
    }' \
    $LOCAL
```

Select

Every query operation will have the count key to show how many items were returned in the response. If you just want the count then you can include the `--select COUNT` option to return that.

Scanning

> I will never use the Scan operation unless I know what I am doing

This is because scan operation operates on the entire table and if you do this in production it will very quickly use up all of your read capacity.

Only do this if your table is very small, if you are exporting your table's data to another storage system, or if you use global secondary index in a special way, but this is rarely the case.

```
aws dynamodb scan \
    --table-name UserOrdersTable \
    $LOCAL
```

This will just return all of the items within the table, however, DynamoDB's request have 1MB limit and it will return a "NextToken" key for you to use to retrieve the next set of items.

Server side filtering

You can do server side filtering so that you do not need to filter the result when it is returned to the client.

> Why can't you do this in the key-condition-expression? Well that is only used for filtering on primary keys. So if you are filtering on attributes that are not part of the primary key it will not work, you must do it in filter-expression.

This is done by using the `filter-expression` flag where you specify the condition on what you want to filter by. An example would be:

```
aws dynamodb query \
    --table-name UserOrdersTable \
    --key-condition-expression "Username = :usr" \
    --filter-expression "Amount > :amt" \
    --expression-attribute-values '{
        ":usr": {"S": "daffyduck"},
        ":amt": {"N": "100"},
    }' \
    $LOCAL
```

Do keep in mind that the filtering process are applied after the items are retrieved by the primary key. Which means you will be consuming the read capacity of the amount of items you have got, not the number of read capacity of the filtered result.

Say the primary key returns 4 items, but the filtering only return to the client 1 items, you will be consuming 4 items of read capacity.

Secondary Indexes

Using the primary key of the table is the most efficient way to get items without any slow scan operations, but that makes your query inflexible as you are forced to query on the primary key. You would need to do filtering instead after all of the rows have been retrieved on those non-key attributes.

DynamoDB have concept of secondary indexes, which allow you to specify alternative primary key structure that can be used for query or scan, but not GetItem operations.

Again there are two types of secondary index, local and global. Local keeps the same partition key, but allows you to specify a different sorting key. Global allows you to pick a different partition key and sorting key.

Basics

- With secondary indexes, there can be duplicate items. Primary key enforce uniqueness, however, secondary indexes does not because you can point them to attributes that can be duplicates.
- Secondary index attribute are not required. With primary key, you must put
- You can only make 20 global secondary index and 55 local secondary index per table

Projected attributes

When you set up a secondary index, you have to specify which attributes from the base table you want to project (copy) into the secondary index. Basically this is saying "oh which attribute do you want to make it available from the secondary index directly without needing to retrieve from the original table".

You can specify three options:

1. **KEYS_ONLY**: The secondary index will only include keys for THIS index that you setting up and the base table's partition and sorting key value, but nothing else
2. **ALL**: Every attribute from the base table will be available in the secondary index
3. **INCLUDE**: You pick what attribute you want to project onto the secondary index

Local Secondary Index

When you create local secondary index, it must be specified at table creation time. Meaning you cannot add local secondary index to an existing table. In addition, the underlying partition key must store less than 10GB of data, so you would need to be conscience of what your projected attribute are for the secondary index because the 10GB limit combines both the item in the base table and in the local secondary index, if you store the same attribute from the base table, then you are essentially duplicating the storage.

In addition, you can only create local secondary index on composite key, and you should be able to see the reason why.

To create a local secondary index, you would add this to your table creation command:

```
  --local-secondary-indexes '[
    {
        "IndexName": "UserAmountIndex",
        "KeySchema": [
            {
                "AttributeName": "Username",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "Amount",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "KEYS_ONLY"
        }
    }
  ]' \
```

It is specifying a different sorting key compared to the "OrderId" that was specified for the original table.

Now after you created your secondary index you can query it! In addition to the table name that you have to specify for your command you also need to specify the secondary index to query under:

```
aws dynamodb query \
    --table-name UserOrdersTable \
    --index-name UserAmountIndex \
    --key-condition-expression "Username = :usr AND Amount > :amt" \
    --expression-attribute-values '{
        ":usr": {"S": "daffyduck"},
        ":amt": {"N": "100"}
    }' \
    $LOCAL
```

Notice that this is completely different from when we are using server sided filtering, it is straight up querying with the attributes, it doesn't do any filtering which make our query much more efficient and saves reading capacity as well since we are only querying what we need.

Global secondary index

Now with global secondary index you can add either a new simple primary key or composite primary key with a different partition key.

Basics

- You will be provisioning a separate throughput for the global secondary index.
- The writes to the table will be eventual consistency. So if you write to the table, it will be asynchronously replicated to the global secondary index and not immediate. You will get different result when querying the table and global secondary at the same time.
- There is not partition key size limit like with local secondary index
- You can add global secondary index to any table, unlike local secondary index which you can only add to table that has composite keys, and you can add either a simple primary key or composite primary key with the new index

Table creation

When you create global secondary you can either do after you create the table or do it during the table creation. DynamoDB will just fill the global secondary index with data on the existing data in the table after.

> If you create a global secondary index on attributes that don't exist, then when you query the table it will just not give you back anything simple as that. As soon as you start adding items with the matching attribute then global secondary index will spit you back items.

Here is an example of how you can create a global index to an existing table:

```
aws dynamodb update-table \
    --table-name UserOrdersTable \
    --attribute-definitions '[
        {
            "AttributeName": "ReturnDate",
            "AttributeType": "S"
        },
        {
            "AttributeName": "OrderId",
            "AttributeType": "S"
        }
    ]' \
    --global-secondary-index-updates '[
        {
            "Create": {
```

```
                "IndexName": "ReturnDateOrderIdIndex",
                "KeySchema": [
                    {
                        "AttributeName": "ReturnDate",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "OrderId",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        }
    ]' \
    $LOCAL
```

Global secondary index is a good use case for sparse index. A sparse index is when not every item contains the attribute you're indexing in the global secondary index. So you will get fewer items in the index than the underlying table.

Revision #5
Created 2023-07-08 01:24:39 UTC by Tamarine
Updated 2023-07-09 00:12:58 UTC by Tamarine