**Big Query Hands-on**

|  |  |
| --- | --- |
| BQ Architecture | Reference Architecture |
| Data Sets | **Creating Datasets** |
| Copying Datasets |
| Controlling Access |
| Managing Datasets |
| Tables | Datatypes, Arrays and Structures |
| Creating tables |
| Usage of tables |
| Managing Tables |
| Copying Tables |
| Export Table/Query Data/Result |
| Creating Partitioned tables |
| Views | Creating view |
| Creating Authorize views |
| Creating Materialized views |
| **Data Load** | **Load data from cloud storage** |
| **Loading data from local file** |
| **Query Data** |
| **Query federated data** |
| **data load on nested and repeated tables** |
| **Performance and Pricing** |
| **Wildcard Tables and Partitioning** |
| **BigQuery API Calls** | **Load/Exporting data from/to cloud storage** |
| **Loading data from local file** |
| **Analytical Queries/Interaction with Public Datasets** |
| **Query Data/Job Creation** |
|  | **BigQuery Lab** |

**Create dataset using BQ CLI**

bq mk -d --data\_location='asia-south1' pract1

**Copying Datasets**

Certainly! To copy datasets in BigQuery using the bq command-line tool, you have a few options. Let’s explore them:

1. **Copying Within the Same Project (Same Region)**:
   * If you want to copy a dataset within the same project and the datasets are in the same region, you can use the bq cp command. This operation has **zero cost**.
   * Example:

**bq cp ali\_23.tb10 pract1.tb11**

1. **Copying Between Different Projects (Same Region)**:
   * To copy a dataset from one project to another within the same region, use the following format:
   * bq cp source\_project:source\_dataset.source\_table dest\_project:dest\_dataset.dest\_table
2. **Copying Across Regions**:
   * If you need to copy a dataset across different regions, consider using the **BigQuery Data Transfer Service**. This service allows you to copy datasets, including partitioned data, without extracting, moving, or reloading data.
   * The bq cp command can also handle this scenario, but both the source and destination datasets must be in the same region.
   * Example:
   * bq cp --transfer\_config --data\_source=projects/source\_project/datasets/source\_dataset tables/dest\_table

**Controlling Access**

Certainly! Let’s walk through the steps to control access in a **Google BigQuery** dataset using the **Identity and Access Management (IAM)** system:

1. **Access the Google Cloud Console**:
   * Open the Google Cloud Console.
   * Make sure you are logged in with the appropriate Google account.
2. **Navigate to BigQuery**:
   * Click on the navigation menu (☰) in the top-left corner.
   * Select **BigQuery** from the list of available services.
3. **Select Your Project and Dataset**:
   * In the BigQuery console, choose your project from the project dropdown (if not already selected).
   * Click on the dataset for which you want to manage access.
4. **Access Control**:
   * Click on the **“Share dataset”** button at the top of the dataset details page.
   * This will take you to the IAM permissions page for the dataset.
5. **Add Members and Assign Roles**:
   * Click on the **“Add”** button to add members (users or groups).
   * Enter the email addresses of the users or groups you want to grant access to.
   * Choose the appropriate roles for each member:
     + **BigQuery Data Viewer**: Allows viewing data in the dataset.
     + **BigQuery Data Editor**: Allows modifying data (including running queries).
     + **BigQuery Data Owner**: Full control over the dataset (including managing access).
   * Click **“Add”** to save the changes.
6. **Custom Roles (Optional)**:
   * If you need more granular control, you can create custom roles with specific permissions.
   * Navigate to the **“IAM & Admin”** section in the Google Cloud Console.
   * Click on **“Roles”** and create a custom role with the desired permissions.
   * Assign this custom role to specific users or groups.
7. **Test Access**:
   * To verify access, log in as one of the users you granted access to.
   * Run queries or interact with the dataset to ensure the permissions are working as expected.

Remember to follow the principle of least privilege: grant only the necessary permissions to users based on their roles and responsibilities. If you encounter any issues or need further assistance, feel free to ask! 😊

**Managing dataset**

**Delete dataset with content**

bq rm -r pract1 (recursively delete with contents)

**delete table from dataset**

bq rm ali\_23.tb9

**Datatypes, Arrays and Structures**

**Array**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb9` (string\_array ARRAY<STRING>);

INSERT INTO `trim-sunlight-418610.karthick18081.tb9` (string\_array) VALUES (["apple", "banana", "orange"]);

INSERT INTO `trim-sunlight-418610.karthick18081.tb9` (string\_array) VALUES (["red", "green", "blue"]);

**Structure**

 Create TABLE `trim-sunlight-418610.karthick18081.tb10` (id INT64,person STRUCT<name STRING,age INT64, address STRUCT<street STRING,city STRING,postal\_code STRING>>);

INSERT INTO `trim-sunlight-418610.karthick18081.tb10` (id, person) VALUES (1, STRUCT("John Doe", 30, STRUCT("123 Main St", "New York", "10001")));

INSERT INTO `trim-sunlight-418610.karthick18081.tb10` (id, person) VALUES (2, STRUCT("Alice Smith", 25, STRUCT("456 Elm St", "Los Angeles", "90002")));

**String**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb1`(name STRING,address STRING,city STRING,postal\_code STRING);

INSERT INTO `trim-sunlight-418610.karthick18081.tb1` (name, address, city, postal\_code) VALUES ('John Doe', '123 Main St', 'Anytown', '12345');

**Int64**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb2`(id INT64,age INT64,quantity INT64);

INSERT INTO `trim-sunlight-418610.karthick18081.tb2` (id, age, quantity) VALUES (1, 25, 10);

INSERT INTO `trim-sunlight-418610.karthick18081.tb2` (id, age, quantity) VALUES (2, 30, 20);

**Float**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb3` (price FLOAT64,temperature FLOAT64,weight FLOAT64);

INSERT INTO `trim-sunlight-418610.karthick18081.tb3` (price, temperature, weight) VALUES (10.5, 25.0, 50.0);

INSERT INTO `trim-sunlight-418610.karthick18081.tb3` (price, temperature, weight) VALUES (15.75, 30.2, 45.8);

**Bool**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb4` (is\_active BOOL,is\_deleted BOOL);

INSERT INTO `trim-sunlight-418610.karthick18081.tb4` (is\_active, is\_deleted) VALUES (TRUE, FALSE);

INSERT INTO `trim-sunlight-418610.karthick18081.tb4` (is\_active, is\_deleted) VALUES (FALSE, TRUE);

**Timestamp**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb5` (created\_at TIMESTAMP,updated\_at TIMESTAMP);

INSERT INTO `trim-sunlight-418610.karthick18081.tb5` (created\_at, updated\_at) VALUES ('2022-01-01 12:00:00', '2022-01-01 12:30:00');

INSERT INTO `trim-sunlight-418610.karthick18081.tb5` (created\_at, updated\_at) VALUES ('2022-01-02 09:15:00', '2022-01-02 10:00:00')

**Date**

CREATE TABLE `trim-sunlight-418610.karthick18081.tb6` (birth\_date DATE,join\_date DATE);

INSERT INTO `trim-sunlight-418610.karthick18081.tb6` (birth\_date, join\_date) VALUES ('1990-01-01', '2010-05-15');

INSERT INTO `trim-sunlight-418610.karthick18081.tb6` (birth\_date, join\_date) VALUES ('1985-03-12', '2012-09-20');

**Creating tables**

**Creating table without schema**

bq mk --table gcp-project-419805:ali\_23.table1

**Creating table with schema**

bq mk --table gcp-project-419805:ali\_23.table2 id:INT64,name:STRING

**Usage of tables**

**Viewing a Table**: To display information about a specific table in your dataset, use the following command:

bq show gcp-project-419805:ali\_23.tb10

1. **Running a Query**: Execute a query using the bq query command. For example:

bq query "SELECT person.age FROM ali\_23.tb10 "

1. **Listing Tables**: To list all tables within a dataset, use:

bq ls

1. **Copying Tables**: To copy a table from one dataset to another, use:
2. **Before that we have to create a table in another dataset**

bq cp gcp-project-419805:ali\_23.tb10 gcp-project- 419805:dest\_dataset.desttb

1. **Exporting Table Data/Query Results**: To export data from a table or query results to a storage location (such as Google Cloud Storage), use:
2. **Export BigQuery**-🡪**GCS**

bq extract gcp-project-419805:ali\_23.table2 gs://bq\_import1/file1.csv

**import gcs-->bq command**

**bq load --source\_format=CSV --autodetect mydataset.mytable gs://mybucket/myfile.csv**

**Integer Range Partitioning:**

Integer range partitioning divides the table into partitions based on ranges of integer values from a specified column.

This type of partitioning is useful when your data has numerical ranges that you want to partition on.

For example, you can partition a table based on age groups, customer IDs, or any other integer column.

**Time-unit Column Partitioning:**

Time-unit column partitioning divides the table into partitions based on time units such as year, month, day, hour, etc.

This type of partitioning is commonly used with DATE or TIMESTAMP columns and allows you to partition data based on specific time intervals.

For example, you can partition a table by day, which creates separate partitions for each day's data, or by hour, which creates separate partitions for each hour's data.

**Ingestion Time Partitioning:**

Ingestion time partitioning automatically partitions data based on the time of data ingestion (i.e., when the data was loaded into BigQuery).

When using ingestion time partitioning, BigQuery automatically creates a new partition for each day's data as it is ingested into the table.

This type of partitioning is useful for scenarios where you want to analyze data based on when it was loaded into BigQuery, regardless of the timestamp within the data itself.

CREATE TABLE `trim-sunlight-418610.karthick18081.time\_unit\_partitioned\_table` (

  event\_id INT64,

  event\_name STRING,

  event\_timestamp TIMESTAMP

)

PARTITION BY DATE(event\_timestamp);

INSERT INTO `trim-sunlight-418610.karthick18081.time\_unit\_partitioned\_table` (event\_id, event\_name, event\_timestamp) VALUES (1, 'Event 1', TIMESTAMP '2023-01-01 08:00:00');

INSERT INTO `trim-sunlight-418610.karthick18081.time\_unit\_partitioned\_table` (event\_id, event\_name, event\_timestamp) VALUES (2, 'Event 2', TIMESTAMP '2023-01-02 10:30:00');

CREATE TABLE `trim-sunlight-418610.karthick18081.ingestion\_time\_partitioned\_table` (

  event\_id INT64,

  event\_name STRING,

  event\_timestamp TIMESTAMP

)

PARTITION BY \_PARTITIONDATE;

INSERT INTO `trim-sunlight-418610.karthick18081.ingestion\_time\_partitioned\_table` (event\_id, event\_name, event\_timestamp) VALUES (1, 'Event 1', TIMESTAMP '2024-01-01 08:00:00');

INSERT INTO `trim-sunlight-418610.karthick18081.ingestion\_time\_partitioned\_table` (event\_id, event\_name, event\_timestamp) VALUES (2, 'Event 2', TIMESTAMP '2024-01-02 10:30:00');

**To see the partitioned table**

SELECT

  table\_name,

  partition\_id,

  total\_rows

FROM

  `trim-sunlight-418610.karthick18081.INFORMATION\_SCHEMA.PARTITIONS`

WHERE

  table\_name = 'ingestion\_time\_partitioned\_table';

### Views

Views are logical tables that are defined by using a SQL query. These include the following types:

* [Views](https://cloud.google.com/bigquery/docs/views-intro), which are logical tables that are defined by using SQL queries. These queries define the view that is run each time the view is queried.

For information about how to create views, see [Create views](https://cloud.google.com/bigquery/docs/views).

* [Materialized views](https://cloud.google.com/bigquery/docs/materialized-views-intro), which are precomputed views that periodically cache the results of the view query. The cached results are stored in BigQuery storage.

In Google BigQuery, Materialized Views provide a way to precompute and store the results of a query, so they can be efficiently queried later. Here's a breakdown of materialized views in BigQuery:

\*\*Automatic Refresh:\*\*

- Materialized views in BigQuery automatically refresh their data based on the underlying query's result set. This means that the materialized view is automatically updated whenever the base data changes, without needing to manually run queries.

\*\*Performance:\*\*

- Materialized views can offer better performance compared to standard views, especially for complex and resource-intensive queries. Since the results are precomputed and stored, querying a materialized view typically involves less computational overhead and can result in faster response times.

\*\*Limitations:\*\*

- BigQuery materialized views have some limitations:

  - Materialized views can't reference other materialized views, only base tables or standard views.

  - There are limitations on the types of queries that can be materialized, including restrictions on certain functions, aggregations, and subqueries.

  - Materialized views have size limitations based on the storage quota of your project.

  - There might be a delay in data freshness, especially for large datasets, as the materialized view needs to be refreshed periodically.

\*\*Join Operations:\*\*

- While materialized views can improve performance for many types of queries, they may introduce complexities when used in join operations, particularly if the joins involve multiple materialized views or if the underlying base data changes frequently.

- Joining materialized views might result in additional processing overhead and potential performance implications, depending on the size and complexity of the joined datasets.

Overall, materialized views in BigQuery offer a powerful mechanism for improving query performance and reducing computational costs, but it's essential to carefully consider their limitations and potential impact on your specific use case, particularly when dealing with join operations and data freshness requirements.

create materialized view `trim-sunlight-418610.karthick18081.student\_db\_m` as

SELECT

  \*

FROM

  `trim-sunlight-418610.karthick18081.StudentData`g

where course="DB";

|  |
| --- |
| Loading data from local file desktop (run this command on our command prompt CLI ..  cd Desktop |

**bq load --source\_format=CSV --autodetect mydataset.mytable mylocalfile.csv**

**Federated queries** let you send a query statement to Spanner or Cloud SQL databases and get the result back as a temporary table. Federated queries use the BigQuery Connection API to establish a connection with Spanner or Cloud SQL. In your query, you use the EXTERNAL\_QUERY function to send a query statement to the external database, using that database's SQL dialect. The results are converted to GoogleSQL data types.

For **Performance and Pricing** related to **Wildcard Tables** and **Partitioning** in BigQuery (BQ), here are the key points:

* **Wildcard Tables**:
  + Use for querying multiple tables with similar names using a wildcard expression.
  + [Can improve query efficiency but has limitations, such as not supporting views or DML statements1](https://cloud.google.com/bigquery/docs/querying-wildcard-tables).
* **Partitioning**:
  + Divides a large table into smaller segments based on a specified column.
  + [Enhances performance and cost control by reducing data scanned per query2](https://medium.com/analytics-vidhya/bigquery-partitioning-clustering-9f84fc201e61).
* **Pricing**:
  + Costs are based on the amount of data processed by queries.
  + [Using wildcard tables and partitioning can lead to cost savings by scanning less data3](https://hatchjs.com/bigquery-partition-by-multiple-columns/).
* **Best Practices**:
  + Avoid excessive use of wildcard tables and ensure tables have identical clustering specifications.
  + [Use partitioning and clustering to optimize query performance and reduce costs4](https://cloud.google.com/bigquery/docs/managing-partitioned-tables).

[For detailed guidance on implementing these features in BQ, you can refer to the official documentation5](https://cloud.google.com/bigquery/docs/best-practices-performance-compute).