# BigQuery Learning Guide – Table of Contents

## Page 1: BigQuery Architecture Overview

### 🏗️ BigQuery Architecture Overview (Verified)

BigQuery is a serverless, highly scalable, and cost-effective multi-cloud data warehouse designed for business agility. Its architecture is based on a separation of storage and compute, enabling powerful and flexible analytics.

🔗 Official Docs Reference:
BigQuery Introduction – Google Cloud Docs
BigQuery Storage Overview
BigQuery Compute with Dremel

### 1. Storage Layer – Colossus + Capacitor

BigQuery stores data in Colossus, Google's global storage system.

Data is stored in a columnar format using the Capacitor engine, which optimizes compression and minimizes disk I/O.

Enables high-throughput access and fast analytical queries by scanning only relevant columns.

### 2. Compute Layer – Dremel

Dremel is the distributed query execution engine powering BigQuery.

It uses a multi-level tree architecture for query processing.

Queries are compiled into execution trees, where intermediate results are aggregated in parallel across thousands of workers.

This architecture delivers low-latency interactive SQL queries, even at petabyte scale.

3. Serverless Model
No infrastructure to manage: no servers, clusters, or tuning.

BigQuery auto-scales based on workload — whether it's 10 rows or 10 billion.

You pay only for what you use (on-demand model) or can opt for flat-rate pricing.

4. Separation of Storage and Compute
You can store data in BigQuery’s managed storage or query external data (e.g., in Google Cloud Storage, Drive, or other BigQuery regions).

This decoupling allows independent scaling, reduced costs, and more flexibility.

5. Ingestion & Federated Queries
Supports streaming data ingestion with near real-time availability.

Can query external sources (federated queries) like Google Sheets, Cloud SQL, or Parquet files in Cloud Storage — without moving the data.

6. Security & Access Control
Uses Google Cloud IAM for granular access control at the project, dataset, or table level.

Data is encrypted at rest and in transit using Google-managed encryption keys (or customer-managed keys).

Offers audit logging, row-level security, and column-level access policies.

7. Performance Optimizations
Partitioning and Clustering enable efficient query execution by minimizing scanned data.

Uses materialized views, caching, and BI Engine for sub-second dashboard performance.

Query plans are automatically optimized — and you can inspect them with EXPLAIN.

Real-time analytics with streaming insert.

Page 2: Partitioning and Clustering in BigQuery

🔹 What Are Partitioning and Clustering?

Partitioning and clustering are optimization techniques that improve performance and reduce cost when working with large tables:

Partitioning: Divides a table into segments (partitions) based on a column like a date or an integer.

Clustering: Organizes data within each partition (or entire table) by sorting it based on one or more columns.

🔹 Partitioning

✅ Types of Partitioning

Date or Timestamp Column Partitioning

Partitions the table by a date or timestamp field.

Useful for log data, transaction records, and time-series analysis.

Example:

```
CREATE TABLE my_dataset.sales (
  transaction_id STRING,
  transaction_date DATE,
  amount NUMERIC
)
PARTITION BY DATE(transaction_date);
```

Ingestion-Time Partitioning

Automatically partitions data by the time it was added to the table.

Useful when your data doesn’t include a natural date column.

Example:

```
CREATE TABLE my_dataset.logs (
  log_id STRING,
  log_message STRING
)
PARTITION BY _PARTITIONTIME;
```

Integer-Range Partitioning

Partitions data based on ranges of an integer column.

Good for cases where time isn't the best dimension, like age groups or customer IDs.

Example:

```
CREATE TABLE my_dataset.user_data (
  user_id INT64,
  user_name STRING
)
PARTITION BY RANGE_BUCKET(user_id, GENERATE_ARRAY(0, 100, 10));
```

✅ Benefits of Partitioning

Improves Query Speed: Queries that filter by the partition column scan only relevant segments.

Reduces Cost: Less data is scanned, so query costs go down.

Manages Data Lifecycle: You can expire old partitions automatically.

✅ Additional Partition Features

Set partition expiration to delete old data automatically:

```
CREATE TABLE my_dataset.sales (
  transaction_id STRING,
  transaction_date DATE,
  amount NUMERIC
)
PARTITION BY DATE(transaction_date)
OPTIONS (
  partition_expiration_days = 30
);
```

Query a specific partition:

```
SELECT *
FROM my_dataset.sales
WHERE transaction_date = '2025-01-01';
```

🔹 Clustering

✅ What is Clustering?

Clustering sorts data in a table (or within partitions) based on one or more columns. It improves the speed of filtering, grouping, and joining operations on those columns.

✅ Creating a Clustered Table

```
CREATE TABLE my_dataset.sales (
  transaction_id STRING,
  transaction_date DATE,
  customer_id STRING,
  amount NUMERIC
)
CLUSTER BY customer_id;
```

You can combine clustering and partitioning for even better performance:

```
CREATE TABLE my_dataset.sales (
  transaction_id STRING,
  transaction_date DATE,
  customer_id STRING,
  amount NUMERIC
)
PARTITION BY DATE(transaction_date)
CLUSTER BY customer_id;
```

✅ Benefits of Clustering

Faster Filtering and Aggregations: Especially when the query filters by the clustered column.

Reduced Cost: Clustering helps reduce the scanned data, saving on resources.

Works with Up to 4 Columns: You can cluster a table using up to four columns.

✅ Things to Keep in Mind

Order of Columns Matters: Place the most commonly filtered column first.

Automatic Re-clustering: Happens in the background as new data is added.

Works Best with Repeated Query Patterns: If your queries often filter or group by the same fields, clustering can drastically help.

🔸 Best Practices

Use partitioning for broad separation (like by day or region).

Use clustering to organize rows inside partitions or the table itself.

Choose columns based on how you filter and analyze data most often.

Combine both techniques for large, high-traffic tables.

Page 3: BigQuery Web UI – Productivity Features

## 🖥️ BigQuery Web UI – Productivity Features (In-Depth)

The BigQuery Web UI, accessed through the Google Cloud Console, provides a feature-rich environment for managing data, writing SQL, and performing advanced analytics — all in a fully serverless and browser-based interface.

### 🔹 1. Navigation Menu

The left-hand navigation panel offers quick access to:

Studio: A workspace to write, save, and run SQL queries or collaborate using notebooks (Colab Enterprise).

Data Transfers: Configure and monitor scheduled data transfers using BigQuery Data Transfer Service.

Dataform: Manage SQL-based data transformation pipelines and Git-integrated workflows within BigQuery.

### 🔹 2. Explorer Pane

This pane displays all your available projects and datasets and allows you to:

Browse datasets and tables by expanding your project.

Access public datasets via the "Add Data" button.

Preview data and inspect schemas before writing queries.

### 🔹 3. SQL Workspace

Designed to help analysts and engineers be productive:

Multi-tab Editor: Work on multiple queries simultaneously with independent tabs.

SQL Editor: Offers syntax highlighting, intelligent autocompletion, and inline error checking.

Query History: Keeps track of previously run queries for easy reuse or debugging.

### 🔹 4. Schema Management

You can manage your data structures directly in the UI:

Edit table schemas: Rename columns, change data types, or add descriptions.

Add or delete columns on demand.

View metadata and schema evolution details.

🔹 5. Data Integration
The UI makes it easy to connect to internal and external sources:

Access over 100 curated public datasets instantly.

Query external sources like Google Drive, Cloud Storage, and Cloud SQL without moving the data.

🔹 6. Collaboration Features
BigQuery is built with collaboration in mind:

Save and Share Queries: Share queries across your team with link-based access.

Notebook Integration: Launch queries in Colab Enterprise notebooks for more advanced workflows.

🔹 7. Productivity Enhancements
A set of built-in tools streamline everyday workflows:

Keyboard shortcuts for running queries, opening tabs, and formatting code.

Query results are cached for 24 hours, reducing redundant compute cost and time for reruns.

Use the Execution Plan view to identify performance bottlenecks and slot usage.

Page 3: Core SQL Syntax in BigQuery (Beginner Level)
📘 Page 3: Core SQL Syntax in BigQuery (Beginner Level)
1. SELECT Statement
The SELECT statement is used to retrieve data from one or more columns in a table.

Syntax:

```
SELECT column1, column2, ...
FROM `project.dataset.table`;
```
Example:

```
SELECT name, age
FROM `my_project.my_dataset.employees`;
```

2. WHERE Clause

The WHERE clause filters records based on specified conditions.

Syntax:

```
SELECT column1, column2, ...
FROM `project.dataset.table`
WHERE condition;
```

Example:

```
SELECT name, department
FROM `my_project.my_dataset.employees`
WHERE department = 'Sales';
```

3. ORDER BY Clause

The ORDER BY clause sorts the result set based on one or more columns.

Syntax:

```
SELECT column1, column2, ...
FROM `project.dataset.table`
ORDER BY column1 [ASC|DESC];
```

Example:

```
SELECT name, salary
FROM `my_project.my_dataset.employees`
ORDER BY salary DESC;
```

4. LIMIT Clause

The LIMIT clause restricts the number of rows returned in the result set.

Syntax:

```
SELECT column1, column2, ...
FROM `project.dataset.table`
LIMIT number;
```

Example:

```
SELECT name
FROM `my_project.my_dataset.employees`
LIMIT 5;
```

5. GROUP BY Clause
The GROUP BY clause groups rows that have the same values in specified columns into summary rows.

Syntax:

```
SELECT column1, COUNT(*)
FROM `project.dataset.table`
GROUP BY column1;
```

Example:

```
SELECT department, COUNT(*) AS employee_count
FROM `my_project.my_dataset.employees`
GROUP BY department;
```

6. HAVING Clause
The HAVING clause filters groups based on aggregate functions.

Syntax:

```
SELECT column1, COUNT(*)
FROM `project.dataset.table`
GROUP BY column1
HAVING COUNT(*) > value;
```

Example:

```
SELECT department, COUNT(*) AS employee_count
FROM `my_project.my_dataset.employees`
GROUP BY department
HAVING COUNT(*) > 10;
```

7. JOIN Operations
Joins combine rows from two or more tables based on related columns.

Syntax:

```
SELECT a.column1, b.column2
FROM `project.dataset.table1` AS a
```

```
JOIN `project.dataset.table2` AS b
ON a.common_column = b.common_column;
```

Example:

```
SELECT e.name, d.department_name
FROM `my_project.my_dataset.employees` AS e
JOIN `my_project.my_dataset.departments` AS d
ON e.department_id = d.department_id;
```

8. Aliases

Aliases assign temporary names to columns or tables for readability.

Syntax:

```
SELECT column_name AS alias_name
FROM `project.dataset.table` AS alias_table;
```

Example:

```
SELECT name AS employee_name
FROM `my_project.my_dataset.employees` AS e;
```

9. Aggregate Functions

Aggregate functions perform calculations on multiple rows and return a single value.

COUNT(): Counts the number of rows.

SUM(): Calculates the total sum.

AVG(): Calculates the average value.

MIN(): Finds the minimum value.

MAX(): Finds the maximum value.
OWOX Reports

Example:

```
SELECT department, AVG(salary) AS average_salary
FROM `my_project.my_dataset.employees`
GROUP BY department;
```

10. DISTINCT Keyword

The DISTINCT keyword returns unique values, eliminating duplicates.

Syntax:

```
SELECT DISTINCT column1
FROM `project.dataset.table`;
```

Example:

```
SELECT DISTINCT department
FROM `my_project.my_dataset.employees`;
```

11. IN Clause
The IN clause filters records based on a list of values.

Syntax:

```
SELECT column1
FROM `project.dataset.table`
WHERE column1 IN (value1, value2, ...);
```

Example:

```
SELECT name
FROM `my_project.my_dataset.employees`
WHERE department IN ('Sales', 'Marketing');
```

12. BETWEEN Clause
The BETWEEN clause filters records within a range.

Syntax:

```
SELECT column1
FROM `project.dataset.table`
WHERE column1 BETWEEN value1 AND value2;
```

Example:

```
SELECT name, salary
FROM `my_project.my_dataset.employees`
WHERE salary BETWEEN 50000 AND 70000;
```

13. LIKE Clause
The LIKE clause filters records based on pattern matching.

Syntax:

```
SELECT column1
FROM `project.dataset.table`
WHERE column1 LIKE pattern;
```

Example:

```
SELECT name
FROM `my_project.my_dataset.employees`
WHERE name LIKE 'A%';
```

14. IS NULL / IS NOT NULL

These clauses filter records with null or non-null values.

Syntax:

```
SELECT column1
FROM `project.dataset.table`
WHERE column1 IS NULL;

SELECT column1
FROM `project.dataset.table`
WHERE column1 IS NOT NULL;
```

Example:

```
SELECT name
FROM `my_project.my_dataset.employees`
WHERE manager_id IS NULL;
```

15. CASE Statement

The CASE statement allows conditional logic in queries.

Syntax:

```
SELECT column1,
  CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE result3
  END AS alias_name
FROM `project.dataset.table`;
```

Example:

```
SELECT name,
```

::contentReference[oaicite:39]{index=39}

Page 4: Window Functions in BigQuery

📘 Page 4: Window Functions in BigQuery (Beginner to Intermediate Level)
🔹 What Are Window Functions?
Window functions (also called analytic functions) perform calculations across a set of rows that are related to the current row — without collapsing rows into groups like regular aggregate functions.

Why use them?

Assign row numbers or rankings

Access previous or next rows' values

Calculate running totals or moving averages

Perform cumulative aggregates across partitions

🔹 General Syntax

```
function_name([arguments]) OVER (
  [PARTITION BY column]
  [ORDER BY column]
  [ROWS BETWEEN ...]
)
```

function_name – What you want to calculate (like RANK(), SUM())

PARTITION BY – Divides data into groups (like GROUP BY)

ORDER BY – Sorts rows within each partition

ROWS BETWEEN – Optional frame definition, useful for moving averages

🔹 Numbering Functions
✅ ROW_NUMBER()
Gives each row a unique number within its partition.

```
SELECT name, salary,
      ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS
row_num
FROM employees;
```

✅ RANK()

Ranks rows; ties get the same rank, but skips numbers after.

```
SELECT name, salary,
      RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank
FROM employees;
```

✅ DENSE_RANK()

Like RANK(), but doesn't skip values after ties.

```
SELECT name, salary,
      DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
dense_rank
FROM employees;
```

🔹 Navigation Functions

✅ LAG()

Accesses a previous row’s value.

```
SELECT name, hire_date,
      LAG(hire_date) OVER (PARTITION BY department ORDER BY hire_date) AS
prev_hire_date
FROM employees;
```

✅ LEAD()

Accesses the next row’s value.

```
SELECT name, hire_date,
      LEAD(hire_date) OVER (PARTITION BY department ORDER BY hire_date) AS
next_hire_date
FROM employees;
```

✅ FIRST_VALUE() / LAST_VALUE()

Returns the first/last value in the ordered set of a partition.

```
SELECT name, salary,
      FIRST_VALUE(salary) OVER (PARTITION BY department ORDER BY hire_date) AS
first_salary
FROM employees;

SELECT name, salary,
      LAST_VALUE(salary) OVER (
        PARTITION BY department ORDER BY hire_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
      ) AS last_salary
FROM employees;
```

🔹 Aggregate Functions as Window Functions

You can use SUM(), AVG(), MIN(), MAX(), etc. across a rolling or full partition.

```
SELECT name, salary,
      SUM(salary) OVER (PARTITION BY department ORDER BY hire_date) AS
cumulative_salary
FROM employees;
```

🔹 Window Frame (ROWS BETWEEN...)

Defines the window for aggregate calculations — like rolling totals or averages.

```
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

Example: Moving Average of Salary (last 3 rows)

```
SELECT name, salary,
      AVG(salary) OVER (
        PARTITION BY department
        ORDER BY hire_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
      ) AS moving_avg_salary
FROM employees;
```

🔹 Real Example: 3-Month Rolling Sales Sum

```
SELECT sale_date, customer_id, amount,
      SUM(amount) OVER (
        PARTITION BY customer_id
        ORDER BY sale_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
      ) AS rolling_3_month_total
FROM sales;
```

## Page 5: Set Operators, CASE, and NULL Handling

Set operators allow you to combine the results of two or more queries. BigQuery supports the following set operators:

1. UNION ALL
Combines the result sets of two queries and includes all rows, including duplicates.

```
SELECT * FROM table1
UNION ALL
SELECT * FROM table2;
```

2. UNION DISTINCT (or simply UNION)
Combines the result sets of two queries and removes duplicate rows.

```
SELECT * FROM table1
UNION DISTINCT
SELECT * FROM table2;
```

Note: In BigQuery, UNION is equivalent to UNION DISTINCT.

3. INTERSECT DISTINCT
Returns rows that are present in both queries.

```
SELECT column_name FROM table1
INTERSECT DISTINCT
SELECT column_name FROM table2;
```

Note: BigQuery requires the DISTINCT keyword with INTERSECT. INTERSECT ALL is not supported.

4. EXCEPT DISTINCT
Returns rows from the first query that are not present in the second query.

```
SELECT column_name FROM table1
EXCEPT DISTINCT
SELECT column_name FROM table2;
```

Note: BigQuery requires the DISTINCT keyword with EXCEPT. EXCEPT ALL is not supported.

🔸 Important Notes

Column Alignment: All queries combined using set operators must have the same number of columns, and corresponding columns must have compatible data types.

Ordering: To order the final result set, use an ORDER BY clause after the last query.

```
SELECT column_name FROM table1
UNION ALL
```

SELECT column_name FROM table2
ORDER BY column_name;
Parentheses: Use parentheses to control the order of operations when combining multiple set operators.

```
(SELECT column_name FROM table1
 UNION ALL
 SELECT column_name FROM table2)
EXCEPT DISTINCT
SELECT column_name FROM table3;
```

Page 5:
STRUCTs, ARRAYs, and UNNEST
🔹 STRUCTs in BigQuery
A STRUCT is a container of ordered fields, each with a name and a type — think of it like a record or object.

✅ Create a STRUCT:

```
SELECT STRUCT("Alice" AS name, 28 AS age) AS person;
```
Output:

```
person.name   person.age
Alice    28
```
You can also extract fields:

```
SELECT person.name, person.age
FROM (SELECT STRUCT("Bob" AS name, 32 AS age) AS person);
```
🔹 ARRAYs in BigQuery
An ARRAY is an ordered list of values — it can hold primitive types or STRUCTs.

✅ Create an ARRAY:

```
SELECT [1, 2, 3, 4] AS numbers;
```
✅ ARRAY of STRUCTs:

```
SELECT [
  STRUCT("John" AS name, 25 AS age),
  STRUCT("Jane" AS name, 27 AS age)
] AS people;
```

🔹 UNNEST in BigQuery

UNNEST() is used to flatten an array into rows — essential when working with repeated data.

✅ UNNEST an array of integers:

```
SELECT number
FROM UNNEST([10, 20, 30]) AS number;
```

Output:

```
number
10
20
30
```

✅ UNNEST with CROSS JOIN:

```
WITH data AS (
  SELECT "Team A" AS team, [1, 2, 3] AS scores
)
SELECT team, score
FROM data, UNNEST(scores) AS score;
```

Output:

```
team    score
Team A        1
Team A        2
Team A        3
```

🔸 Tips

STRUCTs are great for hierarchical or grouped data.

ARRAYs are handy for lists and multi-value fields.

UNNEST must be used properly — avoid creating Cartesian joins unintentionally.

Page 7: BigQuery Optimization Techniques

1. Time Travel

Default: 7 days

Reduce to 1–2 days for temp tables.

```
CREATE TABLE `dataset.temp_table`
OPTIONS (
  enable_time_travel = TRUE,
  max_staleness = INTERVAL 2 DAY
);
```

2. Partitioning

```
CREATE TABLE `dataset.sales`
PARTITION BY DATE(order_date) AS
SELECT * FROM `raw.sales_data`;
```

3. Clustering

```
CREATE TABLE `dataset.sales`
PARTITION BY order_date
CLUSTER BY customer_id, product_id AS
SELECT * FROM `raw.sales_data`;
```

4. SELECT Optimization
Avoid SELECT *.

Select only required columns.

5. Early Filtering

```
SELECT * FROM table WHERE date = '2024-01-01';
```

6. Use UNNEST

```
SELECT id
FROM (
  SELECT UNNEST([id1, id2, id3]) AS id
  FROM table
)
WHERE id = '123';
```

7. LIMIT for Sampling

```
SELECT * FROM table LIMIT 100;
```

8. TABLESAMPLE / RAND()

```
SELECT * FROM table TABLESAMPLE SYSTEM (10 PERCENT);
```

9. INNER JOIN vs LEFT JOIN
Prefer INNER JOIN when possible.
10. APPROX Functions

```
SELECT APPROX_COUNT_DISTINCT(user_id) FROM table;
```

11. Avoid DISTINCT where unnecessary
12. Check Execution Plans & Metadata