Writing Efficient Queries in BigQuery

Optimizing SQL for Analytics in the Cloud

NASHVILLE SOFTWARE SCHOOL

Objectives

1. Understand core architectural differences between Postgres and BigQuery
2. Master primary techniques for writing queries that reduce data I/O and optimize compute
3. Utilize BigQuery’s built-in tools for performance analysis and cost tracking

BigQuery vs PostgreSQL: Core Differences

Feature	PostgreSQL (Transactional)	BigQuery (Analytical)
Architecture	Row-oriented	Columnar
Performance driver	Indexes, query plan stability, row-level operations	Data scanned/processed, data shuffling, parallelism
Pricing model	Based on compute/instance hours	Based on data scanned/ processed (on-demand)
Optimization tool	EXPLAIN ANALYZE (detailed, index-focused)	Query execution details (stage-focused, I/O, shuffle)
Key optimization	Creating effective indexes, tuning memory/buffering	Reducing bytes scanned using partitioning, clustering

Minimize Data Scanned

The single most important principle in BigQuery is to read less data.

- Avoid SELECT *
  - Always explicitly list columns you need for the specific analysis
  - BigQuery bills per column read. SELECT * scans the entire table's data, which is expensive and slow
- Leverage table partitioning
  - Filter tables on the partition column (often a DATE or TIMESTAMP) as early as possible
  - Example: WHERE event_date BETWEEN "2025-01-01" AND "2025-01-31"
  - Partitioning physically divides the table. Filtering prunes partitions, eliminating the need to read data blocks entirely

Minimize Data Scanned Continued

- Utilize table clustering
  - When filtering or aggregating data, use columns defined as clustering keys in your WHERE or GROUP BY clauses
  - Clustering organizes data within each partition, allowing BigQuery to skip data blocks that don't match the filter values (called block pruning)
- Trim data early and often (Filter first!)
  - Place restrictive WHERE clauses (especially partition/cluster filters) first in your main query or CTEs
  - Reduces the volume of data that must be processed in subsequent stages (joins, aggregations, functions)
  - Order your filters from most eliminating to least eliminating (BigQuery executes WHERE filters in written order, it does not try to optimize by reordering them)

Optimize Query Computation

Reduce the work that BigQuery’s execution engine has to do.

- Optimize joins
  - Filter tables before joining them
  - Place the largest table on the left side of the JOIN (BigQuery’s optimizer often handles this, but it’s good habit)
  - Prefer integer keys over string keys for joins, string comparisons are more computationally intensive
- Avoid unnecessary repetition
  - Materialize intermediate results as a temp table instead of repeating complex subqueries or CTEs (if it is complex and used multiple times)
  - Use window functions instead of self-joins to calculate things like running totals or rankings (often more efficient)

Optimize Query Computation Continued

- Handle aggregations
  - Consider using approximate aggregate functions (eg APPROX_COUNT_DISTINCT) when a precise count isn't strictly necessary, they're much faster and cheaper
- Push complex operations to the end
  - Put expensive functions like ORDER BY (especially without a LIMIT), and LIMIT clauses at the outermost part of your query
  - You want to compute complex operations like regex on the smallest possible dataset after filtering and aggregating
- Don't use a complex operation where a simpler approach will work
  - But don't over-simplify - sometimes the window function is overkill and sometimes it's the best way to correctly analyze the data

Monitoring and Debugging: The Execution Plan

BigQuery provides a dynamic, stage-based execution plan instead of EXPLAIN ANALYZE.

- The query is broken into stages (parallel work units).
- Input/Output: Look for the Bytes Read and Bytes Written for each stage.
- Shuffle: High Shuffle Output Bytes means a lot of data is being transferred between processing nodes, which can indicate an expensive GROUP BY or JOIN.
- Bottlenecks: Stages with high Slot Time Consumed or large discrepancies between the average and max time for workers indicate data skew (one worker got much more data than the others).

Goal: Identify the most expensive stage (highest I/O or shuffle) and optimize the corresponding part of your SQL.

Monitoring and Debugging: Information Schema

INFORMATION_SCHEMA.JOBS contains metadata about every BigQuery job executed in your project.

Key columns:

- Project_id, user_email: who ran the query and where
- Job_id, creation_time: unique ID and start time of the job
- Total_bytes_billed: amount of data scanned and billed (primary cost metric)
- Total_slot_ms: total processing time consumed across all workers
- State: job status (DONE, RUNNING, FAILED)
- Cache_hit: did the query use cached results?
- Error_result: details on if a query failed and the reason

Example Query: Top 5 Most Expensive Jobs

```
SELECT query,
       total_bytes_billed,
       total_slot_ms,
       user_email,
       creation_time
FROM `region-us`.INFORMATION_SCHEMA.JOBS
WHERE creation_time >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(),
INTERVAL 7 DAY)
       AND job_type = 'QUERY'
       AND state = 'DONE'
ORDER BY total_bytes_billed DESC
LIMIT 5;
```

Note that queries using INFORMATION_SCHEMA.JOBS must use a region qualifier like region-us as in this example.

NASHVILLE SOFTWARE SCHOOL

Using INFORMATION_SCHEMA to Track Usage

```
-- This query analyzes BigQuery job metadata to calculate daily total_bytes_billed
-- and a 30-day rolling sum of billed bytes.

WITH daily_billing AS (
  -- 1. Aggregate total_bytes_billed by day
  SELECT
    -- Use job creation date for billing tracking
    DATE(creation_time) AS query_date,

    -- Sum the bytes billed for all relevant jobs on that day
    SUM(total_bytes_billed) / POW(1024, 4) AS tb_daily
  FROM
    -- Reference the regional INFORMATION_SCHEMA.JOBS view.
    `region-us`.INFORMATION_SCHEMA.JOBS
  GROUP BY 1
)
-- 2. Calculate the 30-day rolling total using a window function
SELECT
  query_date,
  tb_daily,
  -- Window function to sum the daily billed bytes over the current row
  -- and the 29 days preceding it (totaling 30 days).
  SUM(tb_daily)
    OVER (
      ORDER BY UNIX_DATE(query_date)
      RANGE BETWEEN 29 PRECEDING AND CURRENT ROW
    ) AS rolling_30_day_tb
FROM daily_billing
ORDER BY query_date DESC
```