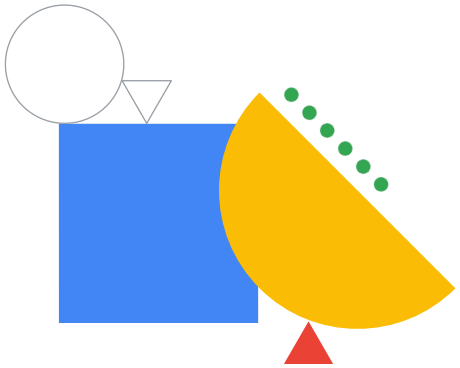
Google Cloud

Optimizing Queries for Performance



Agenda

01 BigQuery Performance Pitfalls

02 Prevent Data Hotspots

03 Diagnose Performance Issues with the Query Explanation Map

This is one of the modules that everyone loves - how to make your queries run faster and save on data processing costs. In this performance optimization module we will cover the key types of work that BigQuery does on the back-end and how you can avoid falling into performance traps.

Then we'll cover one of the most common data traps which is having too much data skewed to a few values.

Last up is your best tool for analyzing and debugging performance issues -- the query explanation colored map.

Let's get started.

BigQuery Performance Pitfalls

Four key elements of work

- **I/O** — How many bytes did you read?
- **Shuffle** — How many bytes did you pass to the next stage?
  - Grouping — How many bytes do you pass to each group?
- **Materialization** — How many bytes did you write to storage?
- **CPU work** — User-defined functions (UDFs), functions

Avoid input / output wastefulness

- Do not SELECT *, use only the columns you need
- Filter using WHERE as early as possible in your queries
- Do not use ORDER BY without a LIMIT

Prevent Data Hotspots

Shuffle wisely: **Be aware of data skew** in your dataset

- **Filter your dataset** as early as possible (this avoids overloading workers on JOINs)
- Hint: Use the Query Explanation map and compare the Max vs the Avg times to highlight skew
- BigQuery will automatically attempt to reshuffle workers that are overloaded with data

Skewed data creates an imbalance between BigQuery worker slots (uneven data partition sizes)

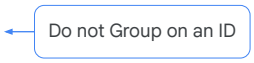
https://cloud.google.com/bigquery/docs/best-practices-performance-patterns

Walk through example using the Query Explanation map:
https://cloud.google.com/bigquery/query-plan-explanation#data_skew_in_stage_1

Careful use of **GROUP BY**

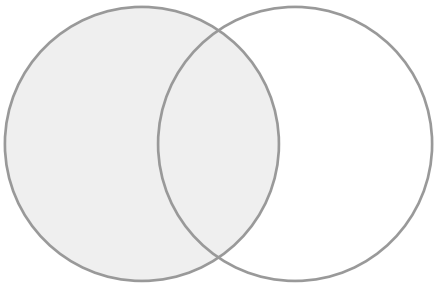
- Best when the number of distinct groups is small (fewer shuffles of data).
- Grouping by a high-cardinality unique ID is a bad idea.

Row	contributor_id	LogEdits
1	2221364	4
2	104574	4
3	73576	4
4	311307	4
5	291919	4
6	140178	4
7	181636	4
8	3661553	4
9	3600820	4
10	4737290	4
11	938404	4
12	295955	4
13	183812	4
14	1811786	4
15	8918196	4
16	561624	4
17	5338406	4



Joins and Unions

- Know your join keys and if they're unique -- no accidental cross joins
- LIMIT Wildcard UNIONS with _TABLE_SUFFIX filter
- Do not use self-joins (consider window functions instead)



Limit UDFs to reduce computational load

- Use native SQL functions whenever possible
- Concurrent rate limits:
  - for non-UDF queries: 50
  - for UDF-queries: 6

```
CREATE TEMP FUNCTION SumFieldsNamedFoo(json_row STRING)
  RETURNS FLOAT64
  LANGUAGE js AS """
function SumFoo(obj) {
  var sum = 0;
  for (var field in obj) {
    if (obj.hasOwnProperty(field) && obj[field] != null) {
      if (typeof obj[field] == "object") {
        sum += SumFoo(obj[field]);
      } else if (field == "foo") {
        sum += obj[field];
      }
    }
  }
  return sum;
}
var row = JSON.parse(json_row);
return SumFoo(row);
""";
```


https://cloud.google.com/bigquery/docs/reference/standard-sql/user-defined-functions

Diagnose Performance Issues with the Query Explanation Map

Diagnose performance issues with the **Query Explanation** map

Elapsed time	Slot time consumed	Bytes shuffled	Bytes spilled to disk
4.6 sec	1 min 6.665 sec	64.7 MB	0 B

		Worker timing					
	Stages		Wait	Read	Compute	Write	Rows
✓	S00: Input	Avg:	667 ms	162 ms	1013 ms	26 ms	Input: 313,797,035
		Max:	914 ms	373 ms	1311 ms	117 ms	Output: 27,773,742
✓	S01: Aggregate+	Avg:	0 ms	0 ms	3504 ms	5 ms	Input: 27,773,742
		Max:	1 ms	0 ms	4324 ms	15 ms	Output: 50
✓	S02: Aggregate	Avg:	4 ms	0 ms	5 ms	4 ms	Input: 50
		Max:	4 ms	0 ms	5 ms	4 ms	Output: 8
✓	S03: Output	Avg:	2 ms	0 ms	6 ms	20 ms	Input: 8
		Max:	2 ms	0 ms	6 ms	20 ms	Output: 8

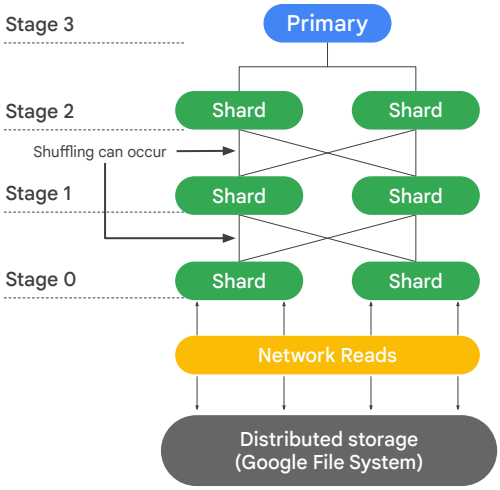
Example: Large GROUP BY

```
SELECT
  LogEdits, COUNT(contributor_id) AS Contributors
FROM (
  SELECT
    contributor_id,
    CAST(LOG10(COUNT(*)) AS INT64) LogEdits # Buckets user edits
  FROM publicdata.samples.wikipedia
  GROUP BY contributor_id)
GROUP BY LogEdits
ORDER BY LogEdits DESC
```

Large **GROUP BY** query

This is a more complicated query to find out how many authors contribute to wikipedia by the number of edits that they made. The LOG10(COUNT(*)) is a way to broadly categorize the number of edits into buckets. For example, if a user contributed 120 edits, then INTEGER(LOG10(120)) will yield 2. Similarly, if a user contributed 185 edits, he will be grouped into the same bucket. If a user contributed 1500 edits, then INTEGER(LOG10(1500) will yield 3. So, the inner SELECT statement groups each contributor into buckets and the final SELECT statement counts the number of contributors for each bucket.

Large GROUP BY means **many forced shuffles**




```
STAGE 3
READ  $10, $90 FROM __stage02_output
SORT  $90 DESC
WRITE  $100, $101 TO __stage03_output
```

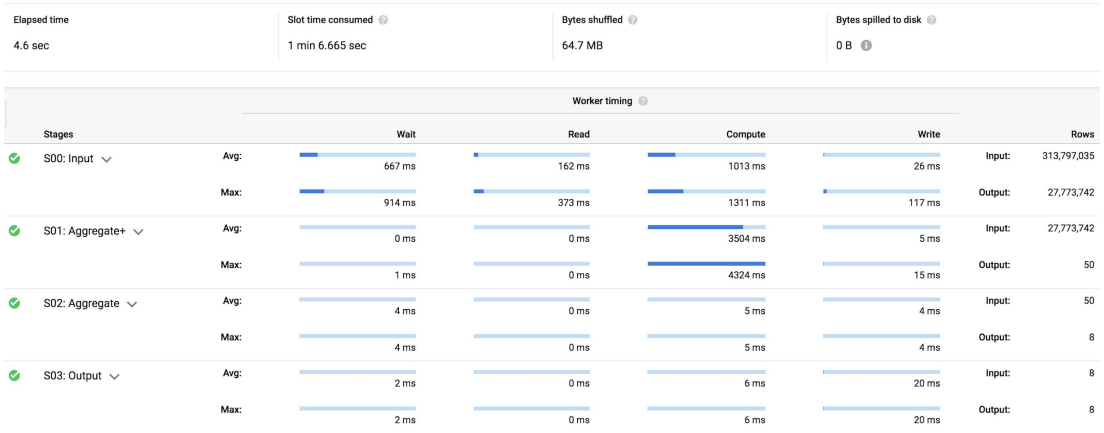
```
STAGE 2
READ  $20, $80 FROM __stage01_output
AGGREGATE  GROUP BY $90 := $80 $10 := SUM_OF_COUNTS($20)
WRITE  $10, $90 TO __stage02_output
```

```
STAGE 1
READ  $60, $50 FROM __stage00_output
AGGREGATE  GROUP BY $80 := $30 $20 := COUNT($70)
COMPUTE  $30 := CAST(log10(CAST($40 AS DOUBLE)) AS INT64)
AGGREGATE  GROUP BY $70 := $60 $40 := SUM_OF_COUNTS($50)
WRITE  $20, $80 TO __stage01_output BY HASH($80)
```

```
STAGE 0
READ  $1:contributor_id
FROM publicdata.samples.wikipedia
AGGREGATE  GROUP BY $60 := $1 $50 := COUNT_STAR()
WRITE  $60, $50 TO __stage00_output BY HASH($60)
```


Because the number of contributors to Wikipedia is huge, the INNER SELECT used ‘GROUP BY’. The large GROUP BY triggered shuffling between stages. Shuffling can be viewed as ‘partitioning’. Shuffling guarantees that all records which have the same value will be stored in the same shard. This allows those operations to be performed efficiently. In the example, each contributor will be shuffled to the same shard. Assuming the distribution is quite even, the implication is that each shard will have fewer contributors to process, hence less memory consumption.

Diagnose performance issues with the Query Explanation map



Elapsed time	Slot time consumed	Bytes shuffled	Bytes spilled to disk
4.6 sec	1 min 6.665 sec	64.7 MB	0 B

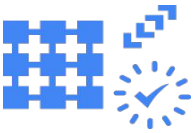
Stages		Worker timing: Wait	Read	Compute	Write	Rows	
S00: Input	Avg:	667 ms	162 ms	1013 ms	26 ms	Input:	313,797,035
	Max:	914 ms	373 ms	1311 ms	117 ms	Output:	27,773,742
S01: Aggregate+	Avg:	0 ms	0 ms	3504 ms	5 ms	Input:	27,773,742
	Max:	1 ms	0 ms	4324 ms	15 ms	Output:	50
S02: Aggregate	Avg:	4 ms	0 ms	5 ms	4 ms	Input:	50
	Max:	4 ms	0 ms	5 ms	4 ms	Output:	8
S03: Output	Avg:	2 ms	0 ms	6 ms	20 ms	Input:	8
	Max:	2 ms	0 ms	6 ms	20 ms	Output:	8

Walk through example using the Query Explanation map:
https://cloud.google.com/bigquery/query-plan-explanation#data_skew_in_stage_1

Table sharding - Then and now

Traditional databases get performance boost by partitioning very large tables.

Usually requires an administrator to pre-allocate space, define partitions, and maintain them.



Manual **Table sharding** divides big table into smaller tables with new suffix of YYYYMMDD.

Queries use Table Wildcard functions.

Date Partition a single table based on specified DAY or a Date Column.

Creating Partitioned Tables

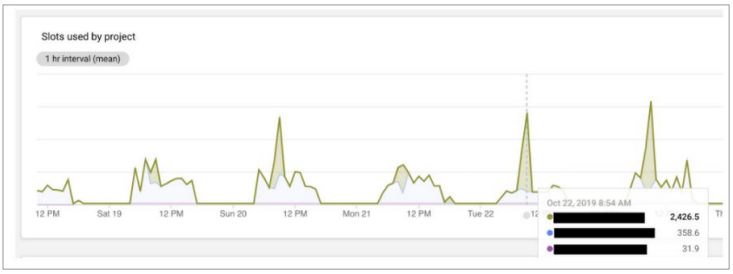
Dividing a dataset into daily tables helped to reduce the amount of data scanned when querying a specific date range. For example, if you have a year's worth of data in a single table, a query that involves the last seven days of data still requires a full scan of the entire table to determine which data to return. However, if your table is divided into daily tables, you can restrict the query to the seven most recent daily tables.

Daily tables, however, have several disadvantages. You must manually, or programmatically, create the daily tables. SQL queries are often more complex because your data can be spread across hundreds of tables. Performance degrades as the number of referenced tables increases. There is also a limit of 1,000 tables that can be referenced in a single query.

How to create partitioned tables:
https://cloud.google.com/bigquery/docs/creating-partitioned-tables

Monitor performance with Cloud Monitoring

- Available for all BigQuery customers
- Fully interactive GUI. Customers can create custom dashboards displaying up to 13 BigQuery metrics, including:
  - Slots utilization
  - Queries in flight
  - Uploaded bytes
  - Stored bytes



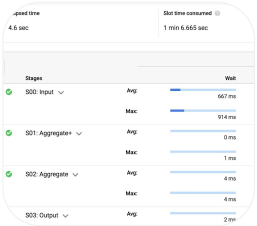
This chart shows Slot Utilization, Slots available and queries in flight for a 1 hr period.

The Cloud Monitoring charting tools offer

- Graphical User Interface to create custom dashboards for multiple Google Cloud Products
- virtually real time data on many parameters (the lag on slot utilization for example is less than 5 minutes)
- Interactive graphical controls (zooming, creating new charts, selecting display modes, etc)

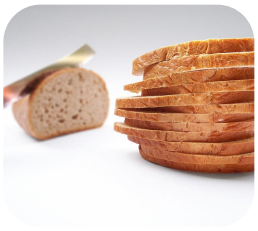
Summary: Query and data model design has a significant impact on performance

Query only the rows and columns you need to reduce bytes processed.



Investigate the query explanation map to see if data skew is bottlenecking your query.

Avoid SQL anti-patterns like ORDER BY without a LIMIT or a GROUP BY on high-cardinality fields.



Use table partitioning to reduce the volume of data scanned.

Writing fast queries may not come naturally at first. Since you're paying for the total bytes processed, you want to first limit the columns of data you want returned and second consider using WHERE filters wherever possible. This doesn't mean you can't write queries that process your entire dataset, BigQuery was built for to be petabyte-scale, but just be mindful with your resources.

Next up we covered the Query Explanation map where you get visual cues of what types of work is most demanded by your queries. Note that a large difference between the average (dark bar graph color) and the max (lighter color) could indicate heavy skew inside of your dataset which can partially be solved by hashing or shuffling your identifying fields.

Then we covered SQL bad behavior like SELECTing every record and ordering it without a limit clause.

Finally, if you're only accessing recent data like the most recent events consider using table partitioning to reduce the bytes scanned by your query.