Google Cloud

BigQuery Best Practices

01
Query Best Practices

Query Complexity

Query Complexity is an indicator of when a query is considered internally as "complex" and therefore would normally run inefficiently. Each time a query is run, BigQuery analyze the query and assign and measure its Query Complexity (QC). If QC exceed a predefined internal threshold, BQ will return the error: "Too many subqueries or query is too complex":

How to estimate the QC value?

- VIEWs, WITH clauses and SQL UDFs are inlined/expanded in the query
- Query complexity is measured as approximately the number of relational operations (like joins, aggregations, subqueries) multiplied by number of columns involved.

Main causes for the error - "Too many subqueries or query is too complex":

1. Deeply nested and used repeatedly **WITH** clauses
2. Deeply nested and used repeatedly **VIEWs**
3. Lots of **UNION ALL**

Google Cloud

General Best practices

- Price your queries before running them using:
  - The query validator in the web UI
  - The `--dry_run` flag in the CLI
  - The `dryRun` parameter when submitting a query job using the API
- Limit query costs by restricting the number of bytes billed
- Do not treat **WITH** clauses as prepared statements. **WITH** clauses are used primarily for readability because they are not **materialized**. If a query appears in more than one WITH clause, it executes in each clause.

SELECT

Optimization: Necessary columns only

Original code

```
SELECT
 *
FROM
`bigquery-public-data.
github_repos.contents`
LIMIT
 100
```

Bytes processed
2.44 TB

Optimized

```
SELECT
 * EXCEPT(content)
FROM
`bigquery-public-data.
github_repos.contents`
LIMIT
 100
```

Bytes processed
15.45 GB

Reasoning

Only select the columns necessary, especially in inner queries. **SELECT *** is cost inefficient and may also hurt performance.

If the number of columns to return is large, consider using **SELECT * EXCEPT** to exclude unneeded columns.

In some use cases, **SELECT * EXCEPT** may be necessary.

Use Preview, it is free.

Console

Optimization: Multiple WITH-clause References

Original code

```
with a as (
  select ...
),
b as (
  select ... from a ...
),
c as (
  select ... from a ...
)
select
  b.dim1, c.dim2
from
  b, c;
```

Optimized

```
create temp table a as
select ...;

with b as (
  select ... from a ...
),
c as (
  select ... from a ...
)
select
  b.dim1, c.dim2
from
  b, c;
```

Reasoning

The **WITH** statement in BigQuery is like a Macro. At runtime the contents of **the subquery will be inlined every place the alias is referenced**. This can lead to query plan explosion as seen by the plan executing the same query stages multiple times.

Refactor to remove multiple references if possible or refactor to use **TEMP** table if the query cannot be refactored and duplicate stages are expensive.

WHERE / ORDER BY

Optimization: WHERE Clause Order & Simplicity

Original code

```
SELECT text
FROM `stackoverflow.comments`
WHERE
  REGEXP_CONTAINS(text, '.*pandas.*')
  AND user_name = 'WesMcKinney'
```

Optimized

```
SELECT text
FROM `stackoverflow.comments`
WHERE
  user_name = 'WesMcKinney'
  AND text LIKE '%pandas%'
```

Rationale

It's better to assume the query engine trusts the user to provide the best order of expressions in the WHERE clause, and does not attempt to reorder expressions. The username expression filters more content first, then the entire text column (with high does not need to be searched.

Also, Regular Expressions (RegExp) have unique optimization patterns, so simpler expressions should be favored.

➢ In logical `AND` clauses, expressions should be ordered with the **most selective expression first** to allow for short circuiting. Logical `OR` clauses are the opposite.

➢ Then prefer to use **simple equalities before complex** conditionals.

- Operations on `BOOL`, `INT`, `FLOAT`, and `DATE` columns are typically faster than operations on `STRING` or `BYTE` columns.
- `BOOL` and `INT` columns are faster for `JOIN`s too.

WHERE clause: Expression order matters!

Original code

```
SELECT
 text
FROM
`bigquery-public-data.stackoverflow.comments`
WHERE
 text LIKE '%java%'
 AND user_display_name = 'anon'
```

1s
12.81 GB
1min 54s slots

Wait ms	avg: 252 max: 537
Read ms	avg: 59 max: 417
Compute ms	**avg: 1527** max: 1970
Write ms	avg: 6 max: 24

Optimized

```
SELECT
 text
FROM
`bigquery-public-data.stackoverflow.comments`
WHERE
 user_display_name = 'anon'
 AND text LIKE '%java%'
```

0.7s
12.81 GB
5s slots

Wait ms	avg: 184 max: 343
Read ms	avg: 22 max: 79
Compute ms	**avg: 32** max: 83
Write ms	avg: 2 max: 7

Reasoning

BigQuery assumes that the user has provided the best order of expressions in the WHERE clause, and does not attempt to reorder expressions.

Expressions in your WHERE clauses should be ordered with the most selective expression first.

The optimized example is **faster** because it doesn't execute the expensive LIKE expression on the entire column content, but rather only on the content from user, 'anon'.

The expression: `user_display_name = 'anon'` filters out much more data than the expression: `text LIKE '%java%'`

2024 update: order does not matter in some cases

Optimization: ORDER BY with LIMIT

Original code

```
SELECT
 post_id, user_id, score, text
FROM
`bigquery-public-data.stackoverflow
.comments`
WHERE
 text LIKE '%java%'
ORDER BY
 score DESC
```

13s
5min 4s slots
1.2 GB shuffled

Optimized

```
SELECT
 post_id, user_id, score, text
FROM
`bigquery-public-data.stackoverflow.
comments`
WHERE
 text LIKE '%java%'
ORDER BY
 score DESC
LIMIT
 100
```

3s
3min 57s slots
1.71 MB shuffled

Reasoning

Writing results for a query with an **ORDER BY** clause can result in Resources Exceeded errors.

The final sorting must be done on a single slot, if you are attempting to order a very large result set, the final sorting can overwhelm the slot that is processing the data.

If you are sorting a very large number of values use a **LIMIT** clause.

\+ Very interesting execution graph comparison

Filter Columns by Native Column Types

Original code

```
SELECT *
FROM  table_name
WHERE
CAST(a.date as STRING) =
STRING(current_date() - 1)
```

Optimized

```
SELECT *
FROM  table_name
WHERE
a.date = current_date() - 1
```

Reasoning

Filter columns using their native types instead of filtering using a different casted type.

Keeping columns as their **native types allows cluster and partition pruning to take effect** while reading from disk instead of having the slot worker do the filtering.

JOINs Optimizations

Optimization: Reduce data before JOINs

Original code

```
SELECT
 user_id,
 COUNT(text) AS comments_count
FROM
`bigquery-public-data`.stackoverflow.comments AS c
JOIN
`bigquery-public-data`.stackoverflow.users AS u
ON
 c.user_id = u.id
GROUP BY
 user_id
```

9s
8min slots
17 GB shuffled

Optimized

```
SELECT
 user_id,
 comments_count
FROM (
 SELECT
   user_id,
   COUNT(text) AS comments_count
 FROM
`bigquery-public-data`.stackoverflow.comments
 GROUP BY
   user_id ) AS c
JOIN
 `bigquery-public-data`.stackoverflow.users AS u
ON
 c.user_id = u.id
```

6s
3min 45s slots
555 MB shuffled

Reasoning

Performing aggregations early reduces the amount of data that is shuffled by a subsequent JOIN.

If the join is using non unique keys on both sides then the result will be exponentially bigger than the input. This is, the join will cause an explosion.

\+ interesting execution graph comparison (records to join)

Optimization: Large table first

Original code

```
select
  t1.dim1,
  sum(t1.metric1),
  sum(t2.metric2)
from
  small_table t1
join
  large_table t2
on
  t1.dim1 = t2.dim1
where t1.dim1 = 'abc'
group by 1;
```

Optimized

```
select
  t1.dim1,
  sum(t1.metric1),
  sum(t2.metric2)
from
  large_table t2
join
  small_table t1
on
  t1.dim1 = t2.dim1
where t1.dim1 = 'abc'
group by 1;
```

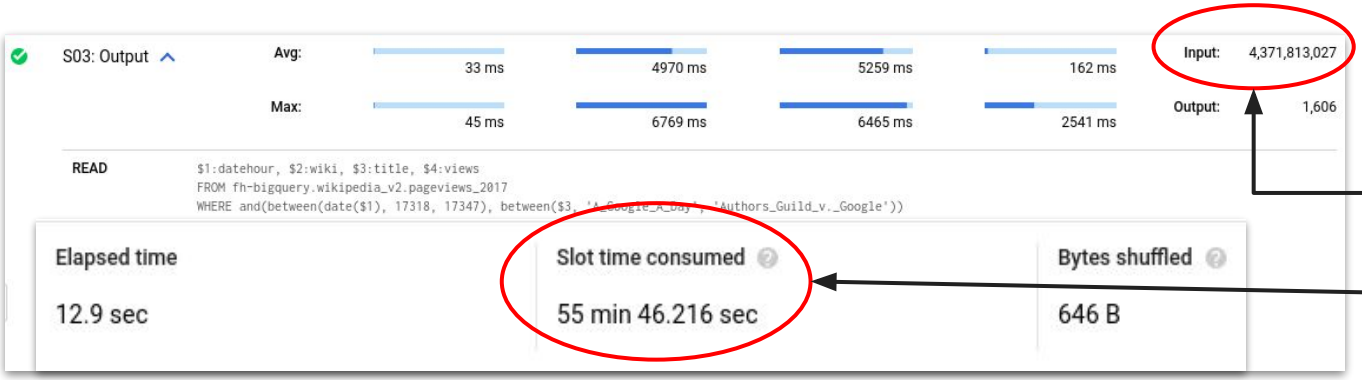
Reasoning

When you create a query by using a JOIN, consider the order in which you are merging the data. The standard SQL query optimizer can determine which table should be on which side of the join, but it is still recommended to order your joined tables appropriately.

The best practice is to **manually place the largest table first**, followed by the smallest, and then by decreasing size.

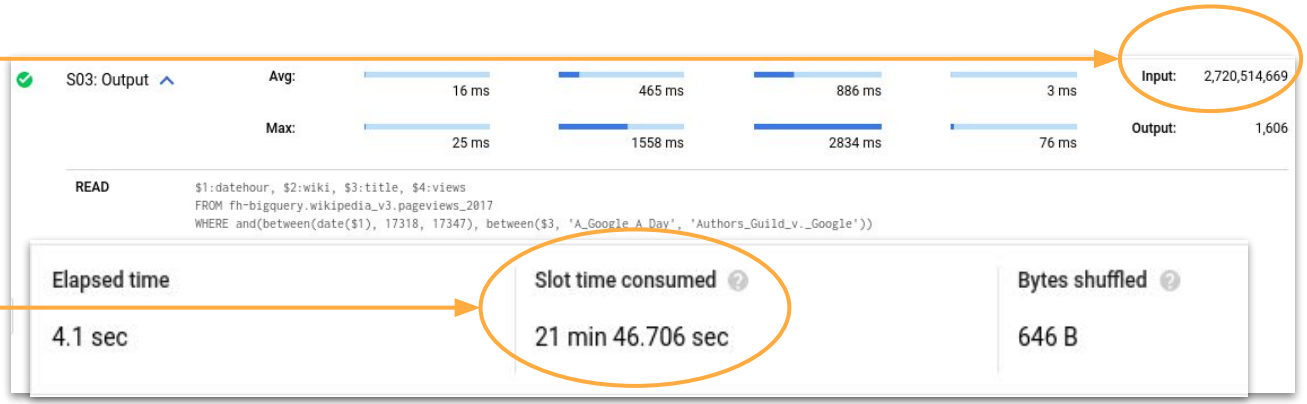
Only under specific table conditions does BigQuery automatically reorder/optimize based on table size.

Most Benefit: Fact Table Clustered on Join Key



Partitioned but **not clustered** fact table results in **scanning more data**, consuming **more slots**.

Clustered fact table results in **scanning less data**, consuming **less slots**.



Optimization: Filter before JOINs

Original code

```
select
  t1.dim1,
  sum(t1.metric1)
from
  `dataset.table1` t1
left join
  `dataset.table2` t2
on
  t1.dim1 = t2.dim1
where t2.dim2 = 'abc'
group by 1;
```

Optimized

```
select
  t1.dim1,
  sum(t1.metric1)
from
  `dataset.table1` t1
left join
  `dataset.table2` t2
on
  t1.dim1 = t2.dim1
where t1.dim2 = 'abc' AND t2.dim2 = 'abc'
group by 1;
```

Reasoning

WHERE clauses should be executed as soon as possible, especially within joins, so the tables to be joined are as small as possible.

WHERE clauses may not always be necessary, as standard SQL will do its best to push down filters. Review the explanation plan to see if filtering is happening as early as possible, and either fix the condition or use a subquery to filter in advance.

Optimization: Avoid Functions on JOIN columns

Original code

```
SELECT fact.*
FROM pageviews_2017 fact
JOIN just_latest_rows dim
-- Trim whitespace before joining
ON TRIM(fact.wiki) = TRIM(dim.wiki)
AND TRIM(fact.title) = TRIM(dim.title)
WHERE
DATE(fact.datehour) BETWEEN
  '2017-06-01' AND '2017-06-30'
AND REGEXP_CONTAINS(dim.title, "Google")
```

Optimized

```
-- Perform TRIM() and another data
-- cleansing inside ELT pipeline.
SELECT fact.*
FROM pageviews_2017 fact
JOIN just_latest_rows dim
-- Join tables using only column
-- names, no functions added.
ON fact.wiki = dim.wiki
AND fact.title = dim.title
WHERE
DATE(fact.datehour) BETWEEN
  '2017-06-01' AND '2017-06-30'
AND REGEXP_CONTAINS(dim.title, "Google")
```

Reasoning

BigQuery **cannot perform join optimizations** (such as snowflake join optimization) between two tables if the join columns are modified by functions.

The best practice is to **standardize your table data** as part of your ingestion process or post-ingestion process via ELT SQL pipelines. This then allows you to join tables without having to modify their join columns via native functions.

Slot time consumed	Bytes shuffled
41 min 7 sec	62.6 KB

Slot time consumed	Bytes shuffled
1 min 6 sec	62.65 KB

97% IMPROVEMENT!

Optimization: Self-Join

Original code

```
select
  t1.*,
  t2.value as prev_value
from
  table t1
left join
  table t2
  on
    t1.id = t2.id
    and t1.date - 1 = t2.date
```

Optimized

```
select
  t.*,
  lag(t.value) over
    (order by t.id) as prev_value
from
  table t;
```

Reasoning

Typically, self-joins are used to compute row-dependent relationships. The result of using a self-join is that it potentially squares the number of output rows. This increase in output data can cause poor performance.

Instead of using a self-join, use a window (analytic) function to reduce the number of additional bytes that are generated by the query.

Optimization: Late Aggregation

Original Code

```
select
  t1.dim1,
  sum(t1.m1)
  sum(t1.m2)
from (select
    dim1,
    sum(metric1) m1
  from `dataset.table1` group by 1) t1
join (select
    dim1,
    sum(metric2) m2
  from `dataset.table2` group by 1) t2
on t1.dim1 = t2.dim1
group by 1;
```

Optimized

```
select
  t1.dim1,
  sum(t1.m1)
  sum(t2.m2)
from (select
    dim1,
    metric1 m1
  from `dataset.table1`) t1
join (select
    dim1,
    metric2 m2
  from `dataset.table2`) t2
on t1.dim1 = t2.dim1
group by 1;
```

Reasoning

Aggregate as late and seldom as possible, as aggregation is very costly.

BUT: *If a table can be reduced drastically by aggregation to then be joined, aggregate it early*

Caution: *With JOINS, this only works if the two tables are already aggregated to the same level (i.e. there is only one row for every join key value)*

01 BigQuery Optimization

Follow SQL best practices

As described in the documentation

- Use clustering as first query optimization step
- Use partitioning to optimize updates
- Select only columns that you need and curate filtering, ordering, and sharding
- Denormalize where needed - use nested and repeated columns
- Choose the right function and pay attention to Javascript user defined functions
- Choose the right data types
- Optimize join and common table expressions
- Look for anti-patterns

Optimiser: I want to optimize performance&costs

Visibility

Admin monitoring tab
INFORMATION_SCHEMA
Top tables
Top queries
Top users

Enforce Control

Per project per user quota

Egress controls

Mandatory where filters

Queries & Schemas

Partition & cluster

Materialized views

Nested & repeated fields

Editions

Workload level

Small granularity with slots

For specific use cases

Partitioning and clustering

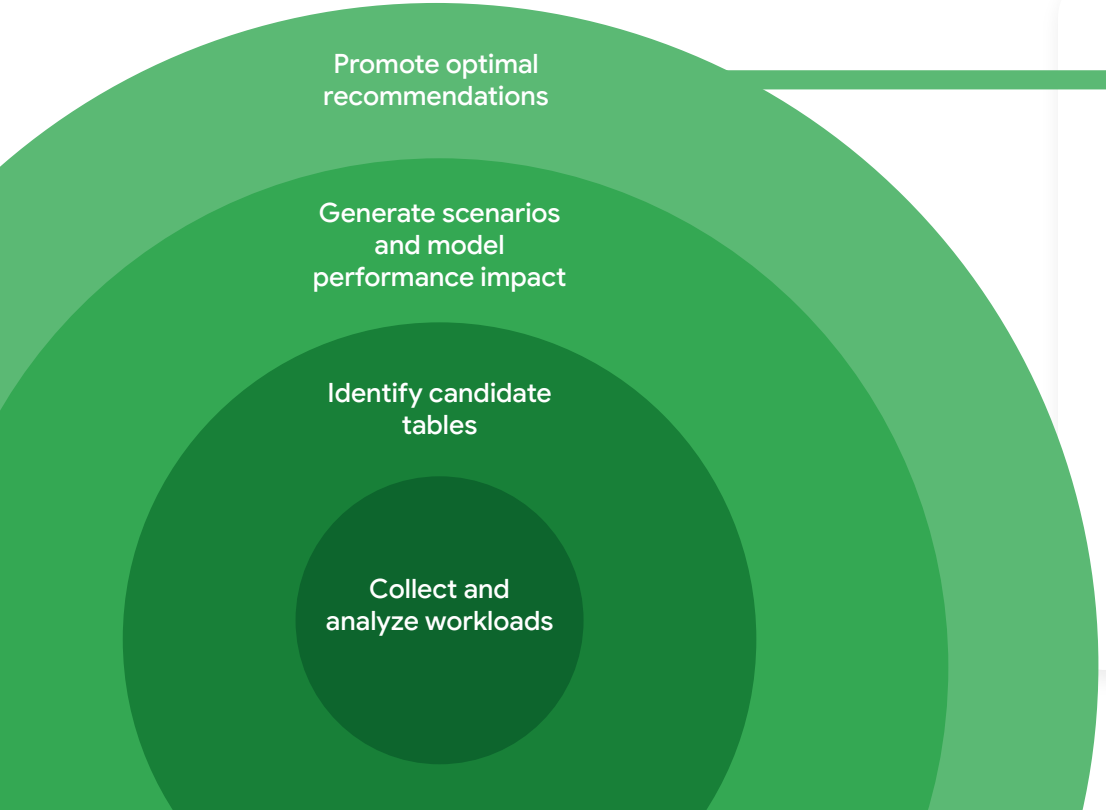
Partitioning

A partitioned table is a special table that is divided into segments, called partitions, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance and control costs by reducing the number of bytes read by a query (link).

Clustering

When you create a clustered table in BigQuery, the table data is automatically organized based on the contents of one or more columns in the table's schema. Clustering can improve the performance of certain types of queries, e.g. queries that use filter clauses and aggregate data (link).

Partitioning and clustering recommender



- **Analyzes** per-table workloads (queries) to find slow and expensive queries with selective filters
- **Simulates** scenarios for alternate partitioning and clustering schemes that can optimize queries
- **Generates** recommendations for partitioning and clustering changes that are predicted to save significant resources

Optimiser: I want to optimize performance&costs

Orders table
Not Clustered; Not partitioned

Order_Date	Country	Status
2022-08-02	US	Shipped
2022-08-04	JP	Shipped
2022-08-05	UK	Canceled
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-05	US	Processing
2022-08-04	JP	Processing
2022-08-04	KE	Shipped
2022-08-06	UK	Canceled
2022-08-02	UK	Processing
2022-08-05	JP	Canceled
2022-08-06	UK	Processing
2022-08-05	US	Shipped
2022-08-06	JP	Processing
2022-08-02	KE	Shipped
2022-08-04	US	Shipped

Orders table
Clustered by Country; Not partitioned

Order_Date	Country	Status
2022-08-04	JP	Shipped
2022-08-04	JP	Processing
2022-08-05	JP	Canceled
2022-08-06	JP	Processing
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-04	KE	Shipped
2022-08-02	KE	Shipped
2022-08-05	UK	Processing
2022-08-06	UK	Canceled
2022-08-02	UK	Canceled
2022-08-06	UK	Processing
2022-08-02	US	Shipped
2022-08-05	US	Processing
2022-08-05	US	Shipped
2022-08-04	US	Shipped

Orders table
Clustered by Country; Partitioned by Order_Date (Daily)

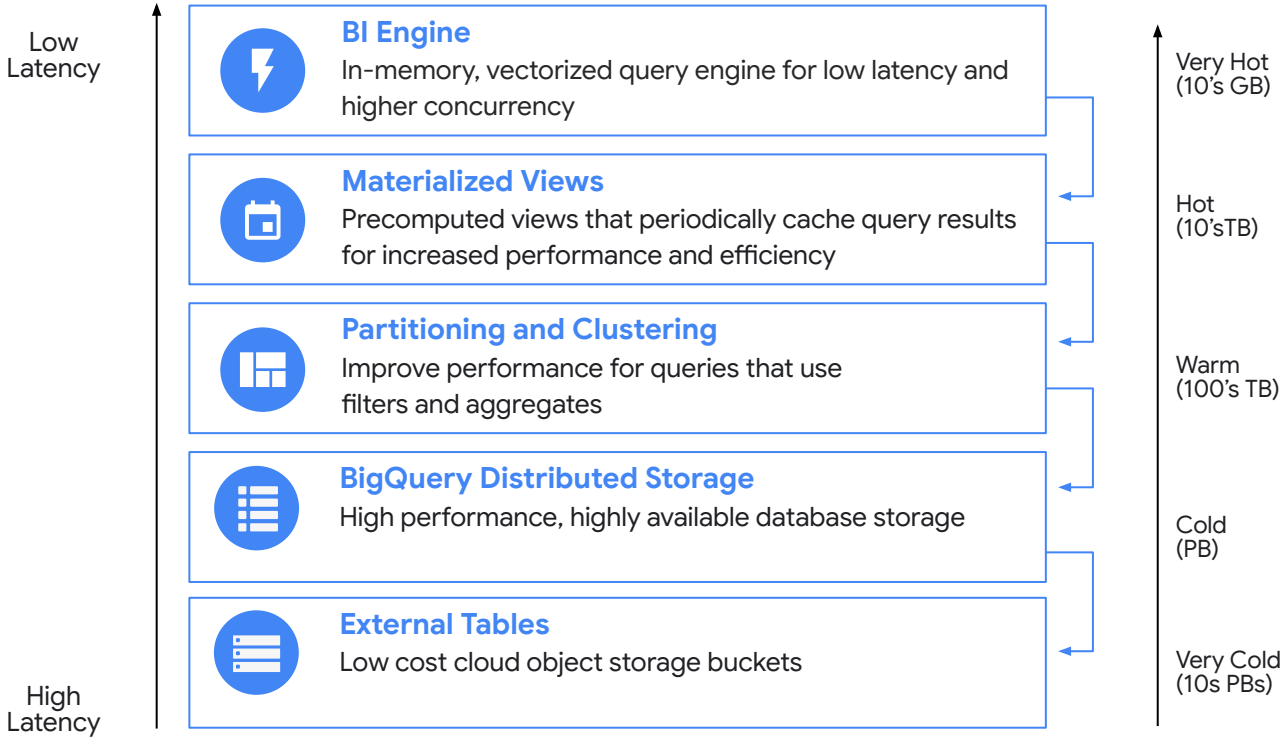
	Order_Date	Country	Status
Partition: 2022-08-02	2022-08-02	KE	Shipped
	2022-08-02	KE	Canceled
Clusters: Country	2022-08-02	UK	Processing
	2022-08-02	US	Shipped

	Order_Date	Country	Status
Partition: 2022-08-04	2022-08-04	JP	Shipped
	2022-08-04	JP	Processing
Cluster: Country	2022-08-04	KE	Shipped
	2022-08-04	US	Shipped

	Order_Date	Country	Status
Partition: 2022-08-05	2022-08-05	JP	Canceled
	2022-08-05	UK	Canceled
Cluster: Country	2022-08-05	US	Shipped
	2022-08-05	US	Processing

	Order_Date	Country	Status
Partition: 2022-08-06	2022-08-06	JP	Processing
	2022-08-06	KE	Shipped
Cluster: Country	2022-08-06	UK	Canceled
	2022-08-06	UK	Processing

Optimiser: I want to optimize performance&costs



Use materialized views to reduce costs

Pros (compared to a table)

- No need to write an incremental update pipeline
- No need to schedule/trigger the update
- Smart query tuning can leverage the MV

Pros (compared to a view)

- Faster than a view, if up-to-date
- Costs less than a view, if up-to-date

Optimiser: I want to optimize&predict costs

Query processing (compute)

The cost to process queries in BigQuery

- **On-demand pricing**
- **Flat-rate pricing: BigQuery Editions and autoscaling**

Storage

The cost to store data that you load into BigQuery, calculated:

- **uncompressed (logical)**
- **compressed (physical) volume of data**.

You can check the documentation to see the compression rate of your data here.

Optimiser: I want to optimize&predict costs

Standard

$0.04/slot hour

Low-cost option for standard SQL analysis

- Autoscaling slots
- Capped at 1,600 slots per reservation
- Max 5 reservations per admin project
- 99.9% SLA
- Zonal High Availability
- Google Cloud Platform-wide certifications [1]
- HIPAA compliance
- Google managed encryption keys

[1] All GCP platform wide certifications including ISO 9001, ISO 27001, SOC 1-3, PCI

Enterprise

$0.06/slot hour

Advanced enterprise workloads

Standard features +

- Unlimited reservation size
- 99.99% SLA
- BI query acceleration
- Integrated ML models
- Full-text search
- Object tables
- VPC Service Controls to prevent data exfiltration
- Data masking, column security and row security

Optional 1 yr commitment ($0.048/slot hour)
Optional 3 yr commitment ($0.036/slot hour)

Enterprise Plus

$0.10/slot hour

Business critical enterprise workloads

Enterprise features +

- Region-level disaster recovery[2]
- Customer-managed encryption keys
- Support for FedRAMP, ITAR and other enhanced compliance regimes available through Assured Workloads

Optional 1 yr commitment ($0.08/slot hour)
Optional 3 yr commitment ($0.06/slot hour)

[2] Roadmap functionality

On-Demand (pay-as-you-go for data processed)

$6.25/TB scanned. First 1 TB/month is free

Includes all capabilities of Enterprise Plus