Optimizing your BigQuery Queries for Performance 2.5

## Overview

Performance tuning of BigQuery is usually carried out because we wish to reduce query execution times or cost. In this lab, we will look at a number of performance optimizations that might work for your use case. Performance tuning should be carried out only at the end of the development stage, and only if it is observed that typical queries take too long. It is far better to have flexible table schemas and elegant, readable, and maintainable queries than to obfuscate table layouts and queries in search of a tiny bit of performance. However, there will be instances where you do need to improve the performance of your queries, perhaps because they are carried out so often that small improvements are meaningful. Another aspect is that knowledge of performance tradeoffs can help you in deciding between alternative designs.

### **Objectives**

In this lab, you learn about the following techniques for reducing BigQuery execution times and costs:

* Minimizing I/O
* Caching results of previous queries
* Performing efficient joins
* Avoid over-whelming single workers
* Using approximate aggregation functions

### **Open BigQuery Console**

1. In the Google Cloud Console, select **Navigation menu** > **BigQuery**.

The **Welcome to BigQuery in the Cloud Console** message box opens. This message box provides a link to the quickstart guide and lists UI updates.

1. Click **Done**.

## Minimize I/O

A query that computes the sum of three columns will be slower than a query that computes the sum of two columns, but most of the performance difference will be due to reading more data, not the extra addition. Therefore, a query that computes the mean of a column will be nearly as fast as a query whose aggregation method is to compute the variance of the data (even though computing variance requires BigQuery to keep track of both the sum and the sum of the squares) because most of the overhead of simple queries is caused by I/O, not by computation.

### **Be purposeful in SELECT**

Because BigQuery uses columnar file formats, the fewer the columns that are read in a SELECT, the less the amount of data that needs to be read. In particular, doing a SELECT \* reads every column of every row in the table, making it quite slow and expensive. The exception is when you use a SELECT \* in a subquery, then only reference a few fields in an outer query; the BigQuery optimizer will be smart enough to only read the columns that are absolutely required.

1. Execute the following query in the [BigQuery EDITOR window](https://console.cloud.google.com/bigquery" \t "_blank):

SELECT

bike\_id,

duration

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

ORDER BY

duration DESC

LIMIT

1

Copied!

content\_copy

In the **Query results** window notice that the query completed in ~1.2s and processed ~372MB of data.

1. Execute the following query in the BigQuery EDITOR window:

SELECT

\*

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

ORDER BY

duration DESC

LIMIT

1

Copied!

content\_copy

In the **Query results** window notice that this query completed in ~4.5s and consumed ~2.6GB of data. Much longer!

If you require nearly all the columns in a table, consider using SELECT \* EXCEPT so as to not read the ones you don’t require.

BigQuery will cache query results to speed up repeat queries. Turn off this cache to see actual query processing performance by clicking **More -> Query settings** and un-checking **Use cached results**

### **Reduce data being read**

When tuning a query, it is important to start with the data that is being read and consider whether it is possible to reduce this. Suppose we wish to find the typical duration of the most common one-way rentals.

1. Execute the following query into the BigQuery editor window:

SELECT

MIN(start\_station\_name) AS start\_station\_name,

MIN(end\_station\_name) AS end\_station\_name,

APPROX\_QUANTILES(duration, 10)[OFFSET (5)] AS typical\_duration,

COUNT(duration) AS num\_trips

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

WHERE

start\_station\_id != end\_station\_id

GROUP BY

start\_station\_id,

end\_station\_id

ORDER BY

num\_trips DESC

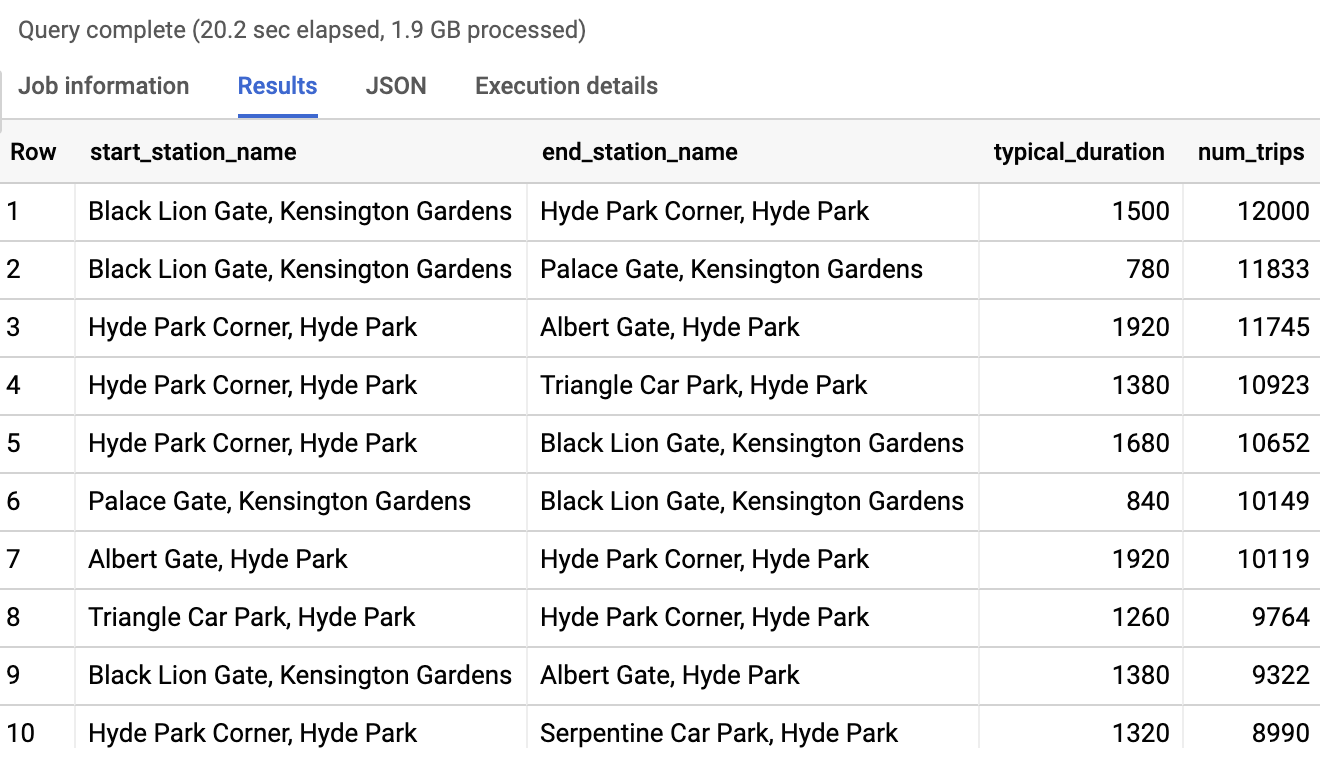
LIMIT

10

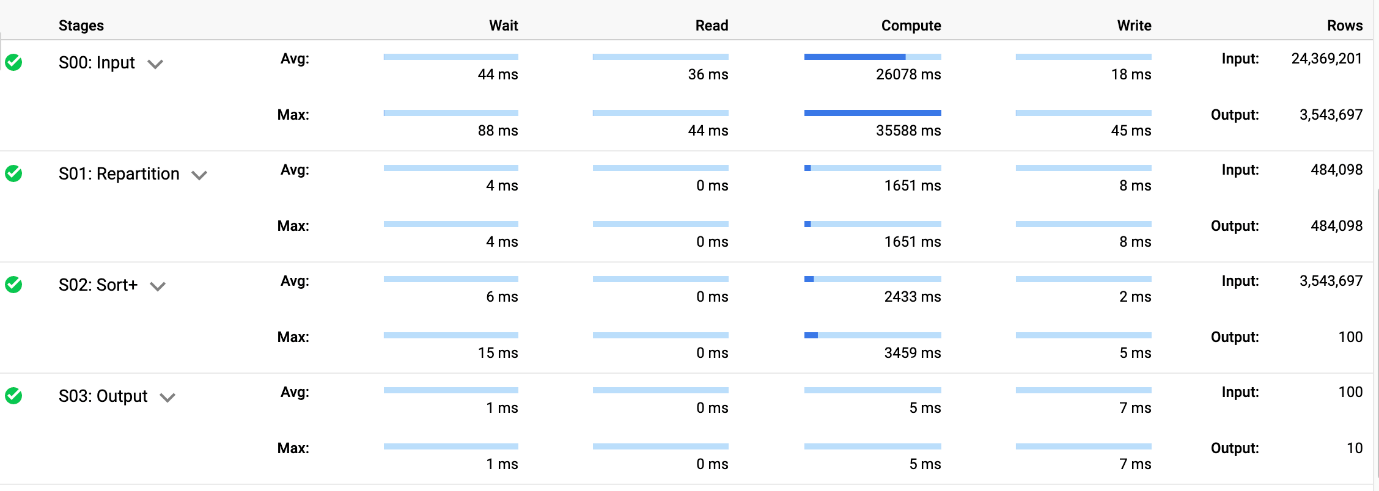
Copied!

content\_copy

The output of your query should look similar to the following:



1. Click on the **Execution details** tab of the **Query results** window.



The details of the query indicate that the sorting (for the approximate quantiles for every station pair) required a repartition of the outputs of the input stage but most of the time is spent during computation.

1. We can reduce the I/O overhead of the query if we do the filtering and grouping using the station name rather than the station id since we will need to read fewer columns. Execute the following query:

SELECT

start\_station\_name,

end\_station\_name,

APPROX\_QUANTILES(duration, 10)[OFFSET(5)] AS typical\_duration,

COUNT(duration) AS num\_trips

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

WHERE

start\_station\_name != end\_station\_name

GROUP BY

start\_station\_name,

end\_station\_name

ORDER BY

num\_trips DESC

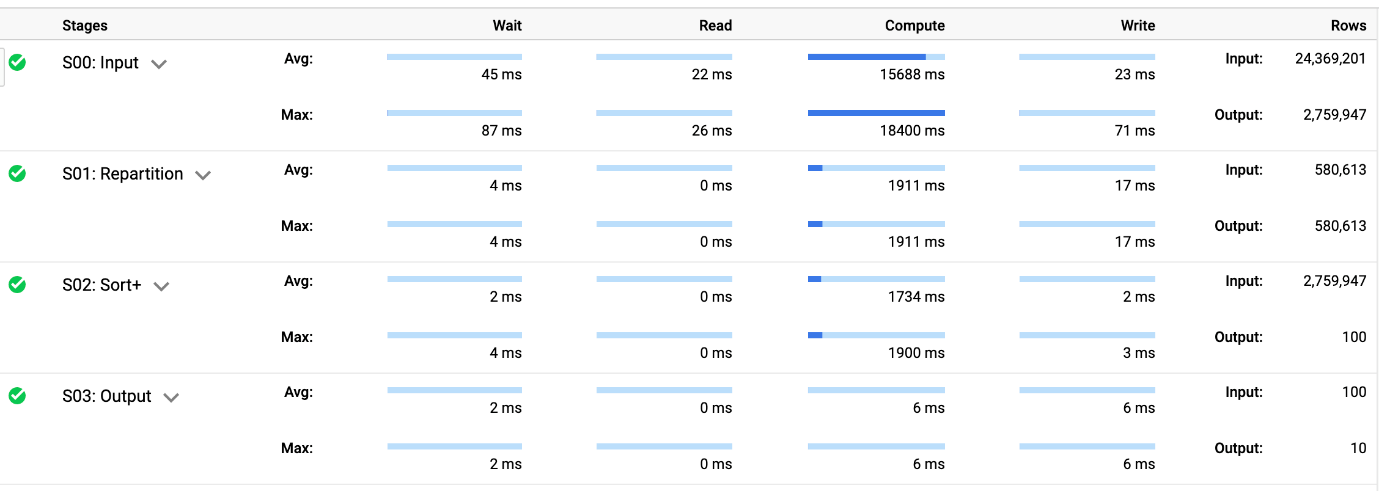
LIMIT

10

Copied!

content\_copy

The above query avoids the need to read the two id columns and finishes in 10.8 seconds. This speedup is caused by the downstream effects of reading less data.



The query result remains the same since there is a 1:1 relationship between the station name and the station id.

### **Reduce number of expensive computations**

Suppose we wish to find the total distance traveled by each bicycle in our dataset.

1. A naive way to do this would be to find the distance traveled in each trip undertaken by each bicycle and sum them up:

WITH

trip\_distance AS (

SELECT

bike\_id,

ST\_Distance(ST\_GeogPoint(s.longitude,

s.latitude),

ST\_GeogPoint(e.longitude,

e.latitude)) AS distance

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire,

`bigquery-public-data`.london\_bicycles.cycle\_stations s,

`bigquery-public-data`.london\_bicycles.cycle\_stations e

WHERE

start\_station\_id = s.id

AND end\_station\_id = e.id )

SELECT

bike\_id,

SUM(distance)/1000 AS total\_distance

FROM

trip\_distance

GROUP BY

bike\_id

ORDER BY

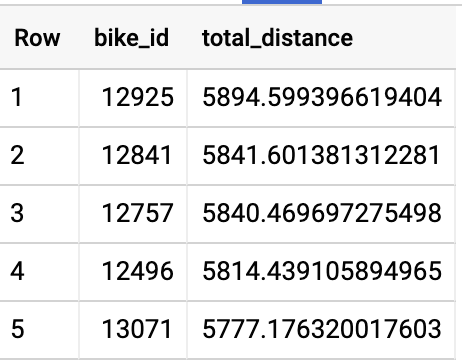
total\_distance DESC

LIMIT

5

Copied!

content\_copy



The above query takes 9.8 seconds (55 seconds of slot time) and shuffles 1.22 MB. The result is that some bicycles have been ridden nearly 6000 kilometers.

1. Computing the distance is a pretty expensive operation and we can avoid joining the cycle\_stations table against the cycle\_hire table if we precompute the distances between all pairs of stations:

WITH

stations AS (

SELECT

s.id AS start\_id,

e.id AS end\_id,

ST\_Distance(ST\_GeogPoint(s.longitude,

s.latitude),

ST\_GeogPoint(e.longitude,

e.latitude)) AS distance

FROM

`bigquery-public-data`.london\_bicycles.cycle\_stations s,

`bigquery-public-data`.london\_bicycles.cycle\_stations e ),

trip\_distance AS (

SELECT

bike\_id,

distance

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire,

stations

WHERE

start\_station\_id = start\_id

AND end\_station\_id = end\_id )

SELECT

bike\_id,

SUM(distance)/1000 AS total\_distance

FROM

trip\_distance

GROUP BY

bike\_id

ORDER BY

total\_distance DESC

LIMIT

5

Copied!

content\_copy

This query only makes 600k geo-distance calculations vs. 24M previously. Now it takes 31.5 seconds of slot time (a 30% speedup), despite shuffling 33.05MB of data.

Click Check my progress to verify the objective.

Minimize I/O

Check my progress

## Cache results of previous queries

The BigQuery service automatically caches query results in a temporary table. If the identical query is submitted within approximately 24 hours, the results are served from this temporary table without any recomputation. Cached results are extremely fast and do not incur charges.

There are, however, a few caveats to be aware of. Query caching is based on exact string comparison. So even whitespaces can cause a cache miss. Queries are never cached if they exhibit non-deterministic behavior (for example, they use CURRENT\_TIMESTAMP or RAND), if the table or view being queried has changed (even if the columns/rows of interest to the query are unchanged), if the table is associated with a streaming buffer (even if there are no new rows), if the query uses DML statements, or queries external data sources.

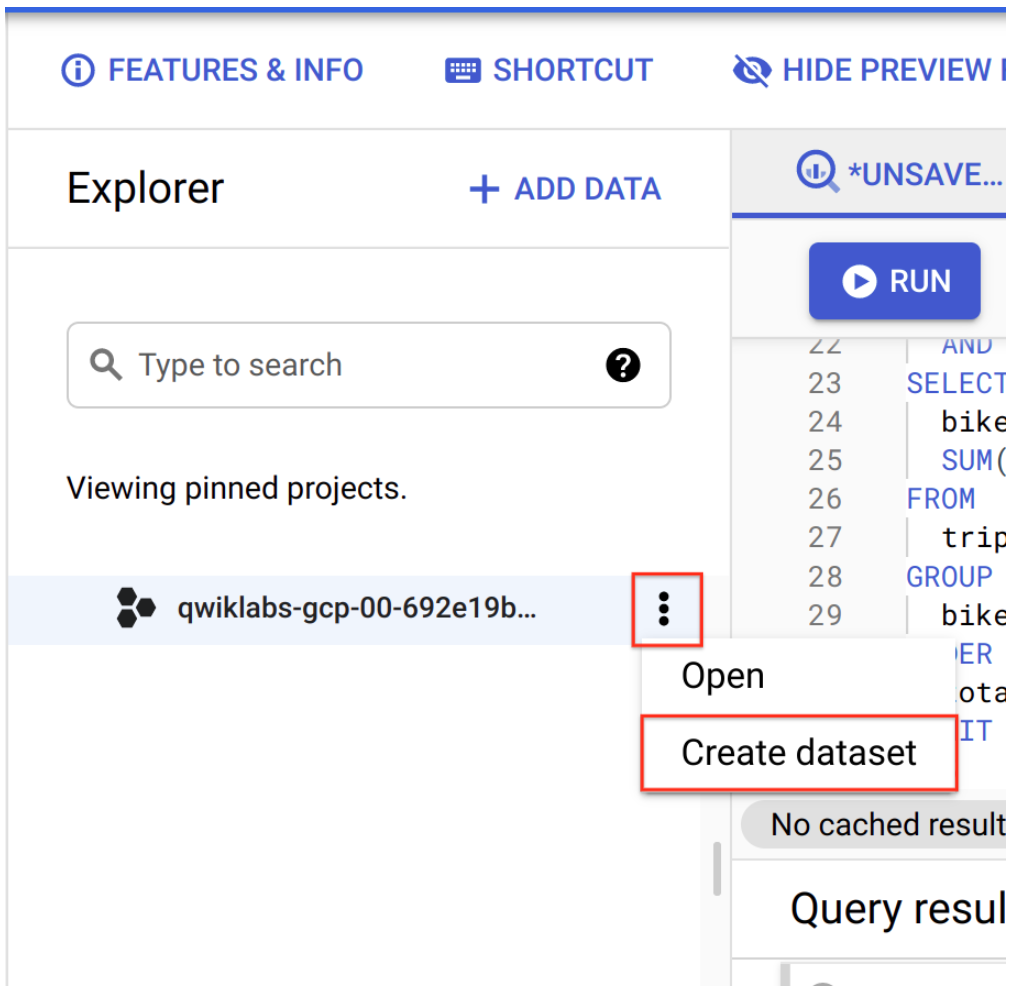
### **Cache intermediate results**

It is possible to improve overall performance at the expense of increased I/O by taking advantage of temporary tables and materialized views.

1. For example, suppose you have a number of queries that start out by finding the typical duration of trips between a pair of stations. The WITH clause (also called a Common Table Expression) improves readability but does not improve query speed or cost since results are not cached. The same holds for views and subqueries as well. If you find yourself using a WITH clause, view, or a subquery often, one way to potentially improve performance is to store the result into a table (or materialized view).

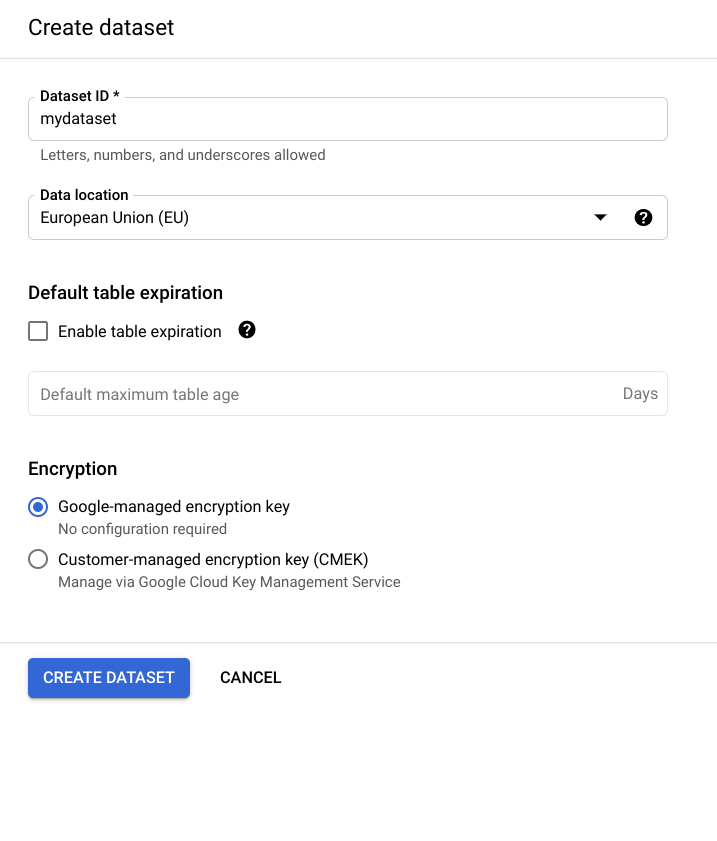
First you will need to create a dataset named mydataset in the EU region (where the bicycle data resides) under your project in BigQuery.

* In the left pane in the **Explorer** section, click on the **View action** icon (three dots) near your BigQuery project (qwiklabs-gcp-xxxx) and select **Create dataset**.



In the **Create dataset** dialog:

* Set the Dataset ID to mydataset.
* Set the Data location to European Union (EU).
* Leave all other options at their default values.
* To finish, click the blue **CREATE DATASET** button.



Now you may execute the following query:

CREATE OR REPLACE TABLE

mydataset.typical\_trip AS

SELECT

start\_station\_name,

end\_station\_name,

APPROX\_QUANTILES(duration, 10)[OFFSET (5)] AS typical\_duration,

COUNT(duration) AS num\_trips

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

GROUP BY

start\_station\_name,

end\_station\_name

Copied!

content\_copy

1. Use the table created to find days when bicycle trips are much longer than usual:

SELECT

EXTRACT (DATE

FROM

start\_date) AS trip\_date,

APPROX\_QUANTILES(duration / typical\_duration, 10)[OFFSET(5)] AS ratio,

COUNT(\*) AS num\_trips\_on\_day

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire AS hire

JOIN

mydataset.typical\_trip AS trip

ON

hire.start\_station\_name = trip.start\_station\_name

AND hire.end\_station\_name = trip.end\_station\_name

AND num\_trips > 10

GROUP BY

trip\_date

HAVING

num\_trips\_on\_day > 10

ORDER BY

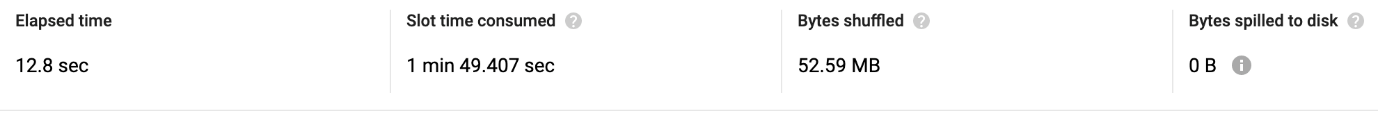
ratio DESC

LIMIT

10

Copied!

content\_copy



1. Use the WITH clause to find days when bicycle trips are much longer than usual:

WITH

typical\_trip AS (

SELECT

start\_station\_name,

end\_station\_name,

APPROX\_QUANTILES(duration, 10)[OFFSET (5)] AS typical\_duration,

COUNT(duration) AS num\_trips

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

GROUP BY

start\_station\_name,

end\_station\_name )

SELECT

EXTRACT (DATE

FROM

start\_date) AS trip\_date,

APPROX\_QUANTILES(duration / typical\_duration, 10)[

OFFSET

(5)] AS ratio,

COUNT(\*) AS num\_trips\_on\_day

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire AS hire

JOIN

typical\_trip AS trip

ON

hire.start\_station\_name = trip.start\_station\_name

AND hire.end\_station\_name = trip.end\_station\_name

AND num\_trips > 10

GROUP BY

trip\_date

HAVING

num\_trips\_on\_day > 10

ORDER BY

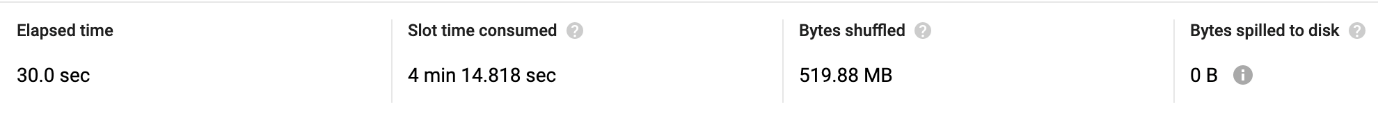
ratio DESC

LIMIT

10

Copied!

content\_copy



Notice the ~50% speedup since the average trip duration computation is avoided. Both queries return the same result, that trips on Christmas take longer than usual. Note, the table mydataset.typical\_trip is not refreshed when new data is added to the cycle\_hire table. One way to solve this problem of stale data is to use a materialized view or to schedule queries to update the table periodically. You should measure the cost of such updates to see whether the improvement in query performance makes up for the extra cost of maintaining the table or materialized view up-to-date.

### **Accelerate queries with BI Engine**

If there are tables that you access frequently in Business Intelligence (BI) settings such as dashboards with aggregations and filters, one way to speed up your queries is to employ **BI Engine**. It will automatically store relevant pieces of data in memory (either actual columns from the table or derived results), and will use a specialized query processor tuned for working with mostly in-memory data. You can reserve the amount of memory (up to a current maximum of 10 GB) that BigQuery should use for its cache from the BigQuery Admin Console, under **BI Engine**.

Make sure to reserve this memory in the same region as the dataset you are querying. Then, BigQuery will start to cache tables, parts of tables, and aggregations in memory and serve results faster.

A primary use case for BI Engine is for tables that are accessed from dashboard tools such as Google Data Studio. By providing memory allocation for a BI Engine reservation, we can make dashboards that rely on a BigQuery backend much more responsive.

Click Check my progress to verify the objective.

Cache results of previous queries

Check my progress

## Efficient joins

Joining two tables requires data coordination and is subject to limitations imposed by the communication bandwidth between slots. If it is possible to avoid a join, or reduce the amount of data being joined, do so.

### **Denormalization**

One way to improve the read performance and avoid joins is to give up on storing data efficiently, and instead add redundant copies of data. This is called denormalization.

1. Thus, instead of storing the bicycle station latitudes and longitudes separately from the cycle hire information, we could create a denormalized table:

CREATE OR REPLACE TABLE

mydataset.london\_bicycles\_denorm AS

SELECT

start\_station\_id,

s.latitude AS start\_latitude,

s.longitude AS start\_longitude,

end\_station\_id,

e.latitude AS end\_latitude,

e.longitude AS end\_longitude

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire AS h

JOIN

`bigquery-public-data`.london\_bicycles.cycle\_stations AS s

ON

h.start\_station\_id = s.id

JOIN

`bigquery-public-data`.london\_bicycles.cycle\_stations AS e

ON

h.end\_station\_id = e.id

Copied!

content\_copy

Then, all subsequent queries will not need to carry out the join because the table will contain the necessary location information for all trips. In this case, you are trading off storage and reading more data against the computational expense of a join. It is quite possible that the cost of reading more data from disk will outweigh the cost of the join -- you should measure whether denormalization brings performance benefits.

Click Check my progress to verify the objective.

Denormalization

Check my progress

### **Avoid self-joins of large tables**

Self-joins happen when a table is joined with itself. While BigQuery supports self-joins, they can lead to performance degradation if the table being joined with itself is very large. In many cases, you can avoid the self-join by taking advantage of SQL features such as aggregation and window functions.

1. Let’s look at an example. One of the BigQuery public datasets is the dataset of baby names published by the US Social Security Administration. It is possible to query the dataset to find the most common male names in 2015 in the state of Massachusetts (Make sure your query is running in the US region by selecting **More** > **Query settings** > **Processing location**):

SELECT

name,

number AS num\_babies

FROM

`bigquery-public-data`.usa\_names.usa\_1910\_current

WHERE

gender = 'M'

AND year = 2015

AND state = 'MA'

ORDER BY

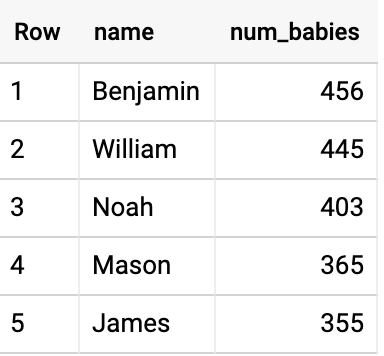
num\_babies DESC

LIMIT

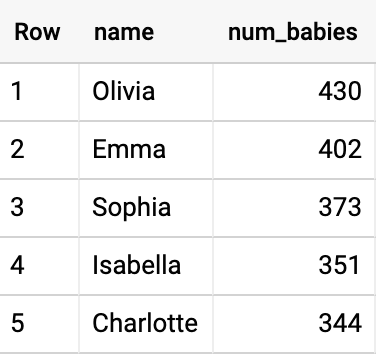
5

Copied!

content\_copy



1. Similarly, query the dataset to find the most common female names in 2015 in the state of Massachusetts:



1. What are the most common names assigned to both male and female babies in the country over all the years in the dataset? A naive way to solve this problem involves reading the input table twice and doing a self-join:

WITH

male\_babies AS (

SELECT

name,

number AS num\_babies

FROM

`bigquery-public-data`.usa\_names.usa\_1910\_current

WHERE

gender = 'M' ),

female\_babies AS (

SELECT

name,

number AS num\_babies

FROM

`bigquery-public-data`.usa\_names.usa\_1910\_current

WHERE

gender = 'F' ),

both\_genders AS (

SELECT

name,

SUM(m.num\_babies) + SUM(f.num\_babies) AS num\_babies,

SUM(m.num\_babies) / (SUM(m.num\_babies) + SUM(f.num\_babies)) AS frac\_male

FROM

male\_babies AS m

JOIN

female\_babies AS f

USING

(name)

GROUP BY

name )

SELECT

\*

FROM

both\_genders

WHERE

frac\_male BETWEEN 0.3

AND 0.7

ORDER BY

num\_babies DESC

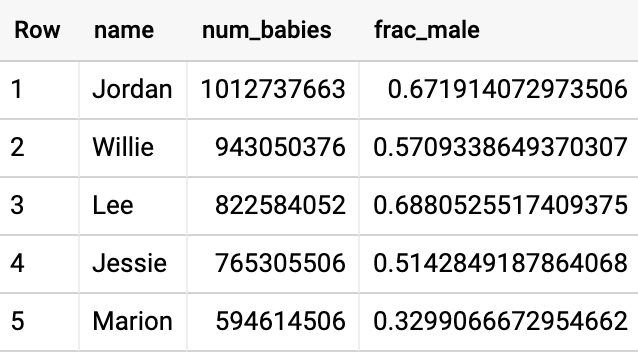
LIMIT

5

Copied!

content\_copy

This took 74 seconds and yielded:



To add insult to injury, the answer is also wrong -- as much as we like the name Jordan, the entire US population is only 300 million, so there cannot have been 982 million babies with that name. The self-JOIN unfortunately joins across state and year boundaries.

1. A faster, more elegant (and correct!) solution is to recast the query to read the input only once and avoid the self-join completely.

WITH

all\_babies AS (

SELECT

name,

SUM(

IF

(gender = 'M',

number,

0)) AS male\_babies,

SUM(

IF

(gender = 'F',

number,

0)) AS female\_babies

FROM

`bigquery-public-data.usa\_names.usa\_1910\_current`

GROUP BY

name ),

both\_genders AS (

SELECT

name,

(male\_babies + female\_babies) AS num\_babies,

SAFE\_DIVIDE(male\_babies,

male\_babies + female\_babies) AS frac\_male

FROM

all\_babies

WHERE

male\_babies > 0

AND female\_babies > 0 )

SELECT

\*

FROM

both\_genders

WHERE

frac\_male BETWEEN 0.3

AND 0.7

ORDER BY

num\_babies DESC

LIMIT

5

Copied!

content\_copy

This took only 2.4 seconds, a 30x speedup.

### **Reduce data being joined**

It is possible to carry out the query above with an efficient join as long as we reduce the amount of data being joined by grouping the data by name and gender early on:

1. Try the following query:

WITH

all\_names AS (

SELECT

name,

gender,

SUM(number) AS num\_babies

FROM

`bigquery-public-data`.usa\_names.usa\_1910\_current

GROUP BY

name,

gender ),

male\_names AS (

SELECT

name,

num\_babies

FROM

all\_names

WHERE

gender = 'M' ),

female\_names AS (

SELECT

name,

num\_babies

FROM

all\_names

WHERE

gender = 'F' ),

ratio AS (

SELECT

name,

(f.num\_babies + m.num\_babies) AS num\_babies,

m.num\_babies / (f.num\_babies + m.num\_babies) AS frac\_male

FROM

male\_names AS m

JOIN

female\_names AS f

USING

(name) )

SELECT

\*

FROM

ratio

WHERE

frac\_male BETWEEN 0.3

AND 0.7

ORDER BY

num\_babies DESC

LIMIT

5

Copied!

content\_copy

The early grouping served to trim the data early in the query, before the query performs a JOIN. That way, shuffling and other complex operations only executed on the much smaller data and remain quite efficient. The query above finished in 2 seconds and returned the correct result.

### **Use a window function instead of a self-join**

Suppose you wish to find the duration between a bike being dropped off and it being rented again, i.e., the duration that a bicycle stays at the station. This is an example of a dependent relationship between rows. It might appear that the only way to solve this is to join the table with itself, matching the end\_date of one trip against the start\_date of the next. (Make sure your query is running in the EU region by selecting **More** > **Query settings** > **Processing location**)

1. You can, however, avoid a self-join by using a window function:

SELECT

bike\_id,

start\_date,

end\_date,

TIMESTAMP\_DIFF( start\_date, LAG(end\_date) OVER (PARTITION BY bike\_id ORDER BY start\_date), SECOND) AS time\_at\_station

FROM

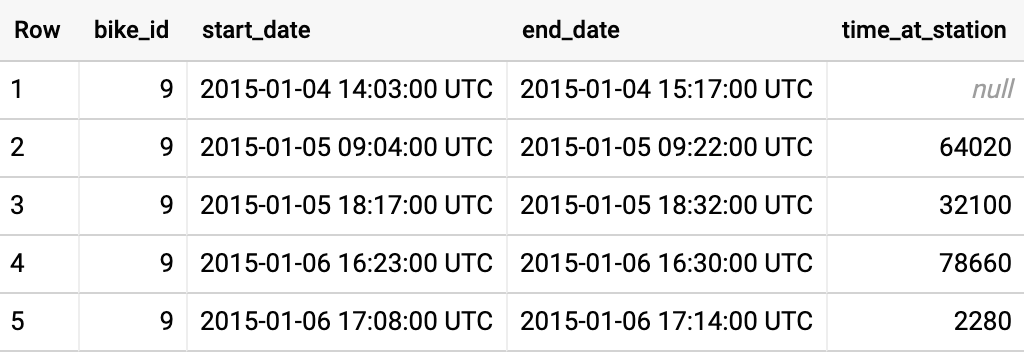
`bigquery-public-data`.london\_bicycles.cycle\_hire

LIMIT

5

Copied!

content\_copy



Notice that the first row has a null for time\_at\_station since we don’t have a timestamp for the previous dropoff. After that, the time\_at\_station tracks the difference between the previous dropoff and the current pickup.

1. Using this, we can compute the average time that a bicycle is unused at each station and rank stations by that measure:

WITH

unused AS (

SELECT

bike\_id,

start\_station\_name,

start\_date,

end\_date,

TIMESTAMP\_DIFF(start\_date, LAG(end\_date) OVER (PARTITION BY bike\_id ORDER BY start\_date), SECOND) AS time\_at\_station

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire )

SELECT

start\_station\_name,

AVG(time\_at\_station) AS unused\_seconds

FROM

unused

GROUP BY

start\_station\_name

ORDER BY

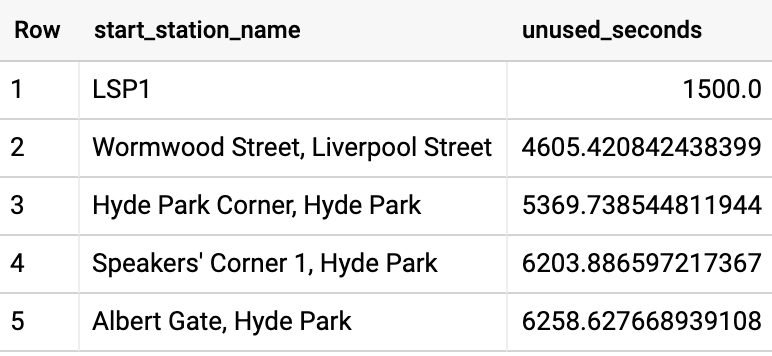
unused\_seconds ASC

LIMIT

5

Copied!

content\_copy



### **Join with precomputed values**

Sometimes, it can be helpful to precompute functions on smaller tables, and then join with the precomputed values rather than repeat an expensive calculation each time.

For example, suppose we wish to find the pair of stations between which our customers ride bicycles at the fastest pace. To compute the pace (minutes per kilometer) at which they ride, we need to divide the duration of the ride by the distance between stations.

1. We could create a denormalized table with distances between stations and then compute the average pace:

WITH

denormalized\_table AS (

SELECT

start\_station\_name,

end\_station\_name,

ST\_DISTANCE(ST\_GeogPoint(s1.longitude,

s1.latitude),

ST\_GeogPoint(s2.longitude,

s2.latitude)) AS distance,

duration

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire AS h

JOIN

`bigquery-public-data`.london\_bicycles.cycle\_stations AS s1

ON

h.start\_station\_id = s1.id

JOIN

`bigquery-public-data`.london\_bicycles.cycle\_stations AS s2

ON

h.end\_station\_id = s2.id ),

durations AS (

SELECT

start\_station\_name,

end\_station\_name,

MIN(distance) AS distance,

AVG(duration) AS duration,

COUNT(\*) AS num\_rides

FROM

denormalized\_table

WHERE

duration > 0

AND distance > 0

GROUP BY

start\_station\_name,

end\_station\_name

HAVING

num\_rides > 100 )

SELECT

start\_station\_name,

end\_station\_name,

distance,

duration,

duration/distance AS pace

FROM

durations

ORDER BY

pace ASC

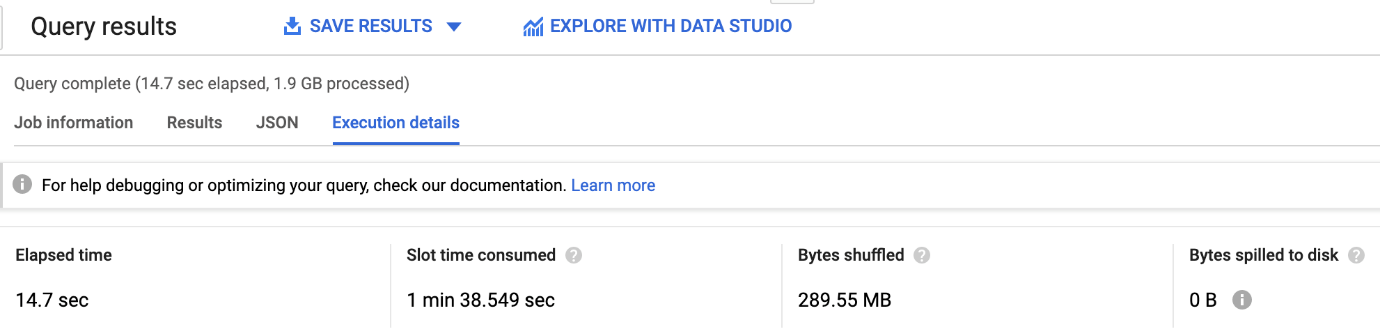
LIMIT

5

Copied!

content\_copy

The above query invokes the geospatial function ST\_DISTANCE once for each row in the cycle\_hire table (24 million times), takes 14.7 seconds and processes 1.9 GB.



1. Alternately, we can use the cycle\_stations table to precompute the distance between every pair of stations (this is a self-join) and then join it with the reduced-size table of average duration between stations:

WITH

distances AS (

SELECT

a.id AS start\_station\_id,

a.name AS start\_station\_name,

b.id AS end\_station\_id,

b.name AS end\_station\_name,

ST\_DISTANCE(ST\_GeogPoint(a.longitude,

a.latitude),

ST\_GeogPoint(b.longitude,

b.latitude)) AS distance

FROM

`bigquery-public-data`.london\_bicycles.cycle\_stations a

CROSS JOIN

`bigquery-public-data`.london\_bicycles.cycle\_stations b

WHERE

a.id != b.id ),

durations AS (

SELECT

start\_station\_id,

end\_station\_id,

AVG(duration) AS duration,

COUNT(\*) AS num\_rides

FROM

`bigquery-public-data`.london\_bicycles.cycle\_hire

WHERE

duration > 0

GROUP BY

start\_station\_id,

end\_station\_id

HAVING

num\_rides > 100 )

SELECT

start\_station\_name,

end\_station\_name,

distance,

duration,

duration/distance AS pace

FROM

distances

JOIN

durations

USING

(start\_station\_id,

end\_station\_id)

ORDER BY

pace ASC

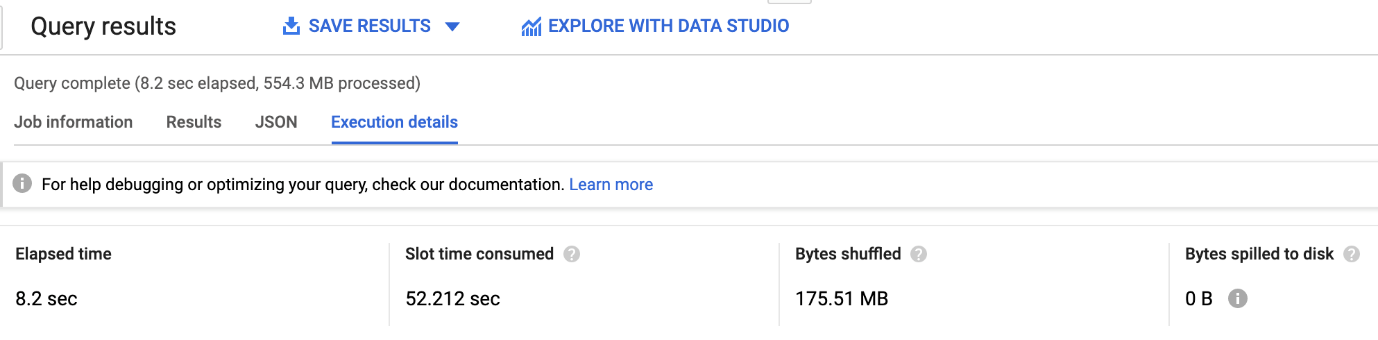
LIMIT

5

Copied!

content\_copy

The recast query with the more efficient joins takes only 8.2 seconds, a 1.8x speedup and processes 554 MB, a nearly 4x reduction in cost.



Click Check my progress to verify the objective.

Joins

Check my progress

## Avoid overwhelming a worker

Some operations (e.g. ordering) have to be carried out on a single worker. Having to sort too much data can overwhelm a worker’s memory and result in a “resources exceeded” error. Avoid overwhelming the worker with too much data. As the hardware in Google data centers is upgraded, what “too much” means in this context expands over time. Currently, this is on the order of one GB.

### **Limiting large sorts**

1. Let’s say that we wish to go through the rentals and number them 1, 2, 3, etc. in the order that the rental ended. We could do that using the ROW\_NUMBER() function

SELECT

rental\_id,

ROW\_NUMBER() OVER(ORDER BY end\_date) AS rental\_number

FROM

`bigquery-public-data.london\_bicycles.cycle\_hire`

ORDER BY

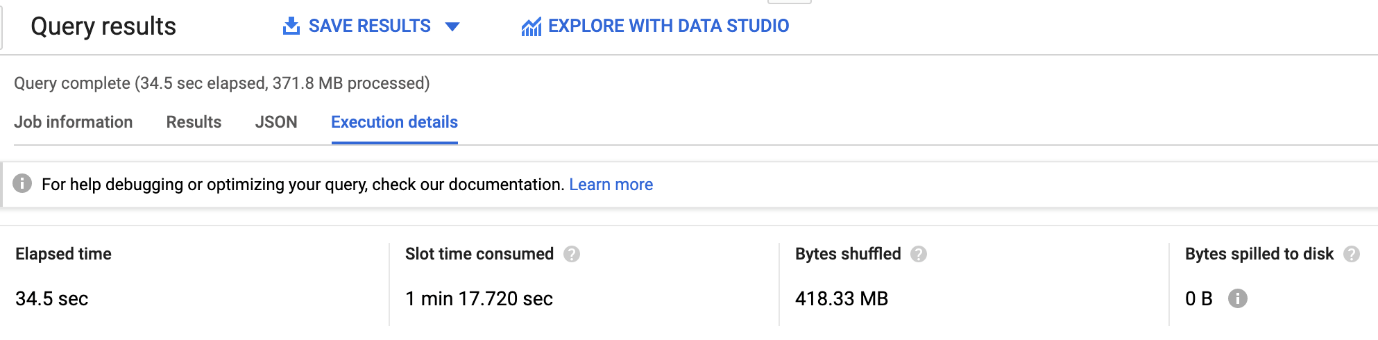
rental\_number ASC

LIMIT

5

Copied!

content\_copy



It takes 34.5 seconds to process just 372 MB because it needs to sort the entirety of the London bicycles dataset on a single worker. Had we processed a larger dataset, it would have overwhelmed that worker.

1. We might want to consider whether it is possible to limit the large sorts and distribute them. Indeed, it is possible to extract the date from the rentals and then sort trips within each day:

WITH

rentals\_on\_day AS (

SELECT

rental\_id,

end\_date,

EXTRACT(DATE

FROM

end\_date) AS rental\_date

FROM

`bigquery-public-data.london\_bicycles.cycle\_hire` )

SELECT

rental\_id,

rental\_date,

ROW\_NUMBER() OVER(PARTITION BY rental\_date ORDER BY end\_date) AS rental\_number\_on\_day

FROM

rentals\_on\_day

ORDER BY

rental\_date ASC,

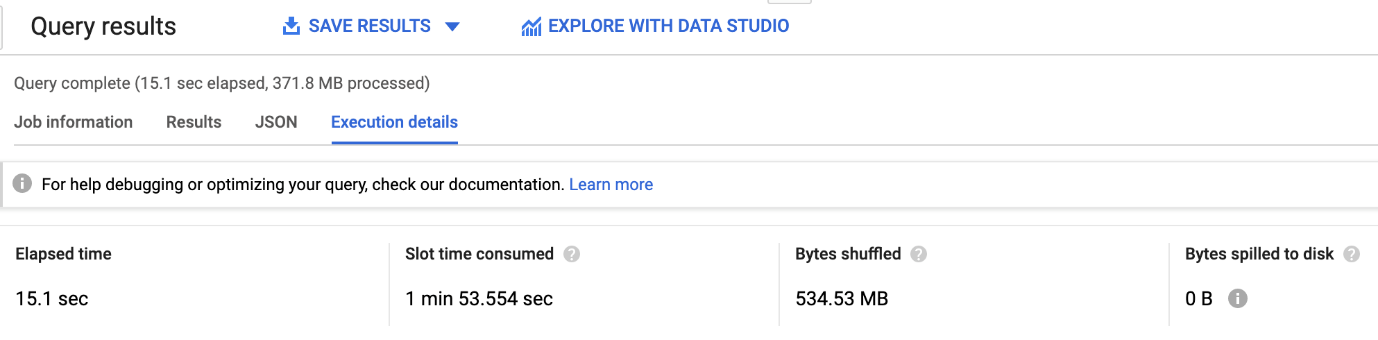
rental\_number\_on\_day ASC

LIMIT

5

Copied!

content\_copy



This takes 15.1 seconds (a 2x speedup) because the sorting can be done on just a single day of data at a time.

Click Check my progress to verify the objective.

Avoid overwhelming a worker

Check my progress

### **Data skew**

The same problem of overwhelming a worker (in this case, overwhelm the memory of the worker) can happen during an ARRAY\_AGG with GROUP BY if one of the keys is much more common than the others.

1. Because there are more than 3 million GitHub repositories and the commits are well distributed among them, this query succeeds (make sure you execute the query in the US processing center):

SELECT

repo\_name,

ARRAY\_AGG(STRUCT(author,

committer,

subject,

message,

trailer,

difference,

encoding)

ORDER BY

author.date.seconds)

FROM

`bigquery-public-data.github\_repos.commits`,

UNNEST(repo\_name) AS repo\_name

GROUP BY

repo\_name

Copied!

content\_copy

Note, while this query will succeed, it can take upwards of 15 minutes to do so. If you understand the query, move on in the lab.

1. Most of the people using GitHub live in only a few time zones, so grouping by the timezone fails -- we are asking a single worker to sort a significant fraction of 750GB:

SELECT

author.tz\_offset,

ARRAY\_AGG(STRUCT(author,

committer,

subject,

message,

trailer,

difference,

encoding)

ORDER BY

author.date.seconds)

FROM

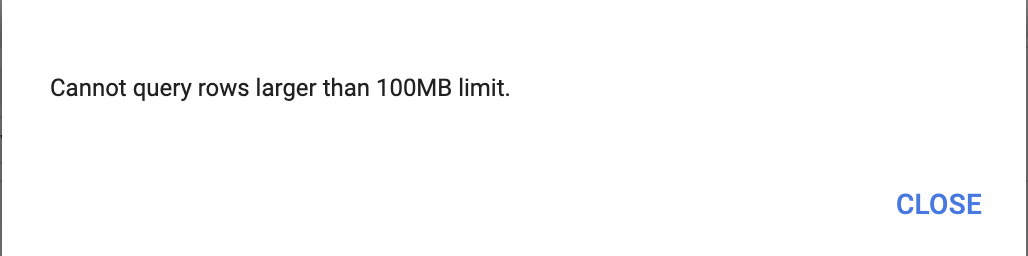
`bigquery-public-data.github\_repos.commits`

GROUP BY

author.tz\_offset

Copied!

content\_copy



1. If you do require sorting all the data, use more granular keys (i.e. distribute the group’s data over more workers) and then aggregate the results corresponding to the desired key. For example, instead of grouping only by the time zone, it is possible to group by both timezone and repo\_name and then aggregate across repos to get the actual answer for each timezone:

SELECT

repo\_name,

author.tz\_offset,

ARRAY\_AGG(STRUCT(author,

committer,

subject,

message,

trailer,

difference,

encoding)

ORDER BY

author.date.seconds)

FROM

`bigquery-public-data.github\_repos.commits`,

UNNEST(repo\_name) AS repo\_name

GROUP BY

repo\_name,

author.tz\_offset

Copied!

content\_copy

Note, while this query will succeed, it can take upwards of 15 minutes to do so. If you understand the query, move on in the lab.

## Approximate aggregation functions

BigQuery provides fast, low-memory approximations of aggregate functions. Instead of using COUNT(DISTINCT …), we can use APPROX\_COUNT\_DISTINCT on large data streams when a small statistical uncertainty in the result is tolerable.

### **Approximate count**

1. We can find the number of unique GitHub repositories using:

SELECT

COUNT(DISTINCT repo\_name) AS num\_repos

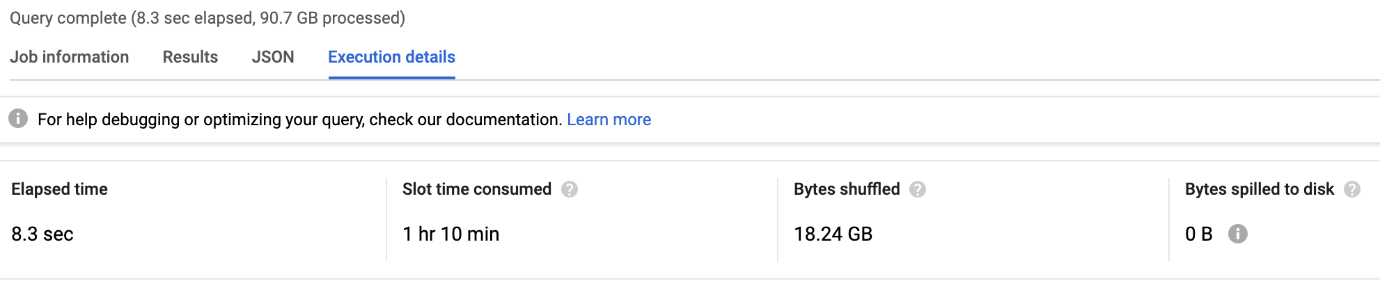
FROM

`bigquery-public-data`.github\_repos.commits,

UNNEST(repo\_name) AS repo\_name

Copied!

content\_copy



The above query takes 8.3 seconds to compute the correct result of 3347770.

1. Using the approximate function:

SELECT

APPROX\_COUNT\_DISTINCT(repo\_name) AS num\_repos

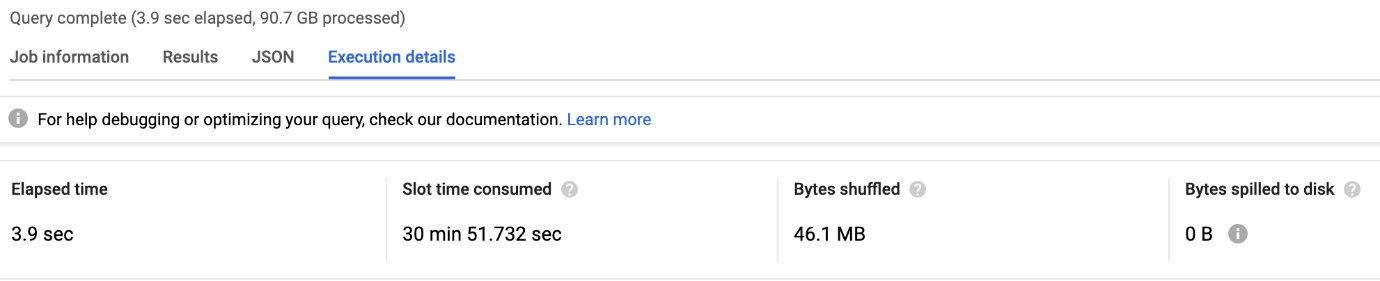
FROM

`bigquery-public-data`.github\_repos.commits,

UNNEST(repo\_name) AS repo\_name

Copied!

content\_copy



takes 3.9 seconds (a 2x speedup) and returns an approximate result of 3399473, which overestimates the correct answer by 1.5%.

The approximate algorithm is much more efficient than the exact algorithm only on large datasets and is recommended in use-cases where errors of approximately 1% are tolerable. Before using the approximate function, measure on your use case!

Other available approximate functions include APPROX\_QUANTILES to compute percentiles, APPROX\_TOP\_COUNT to find the top elements and APPROX\_TOP\_SUM to compute top elements based on the sum of an element.

Click Check my progress to verify the objective.

Approximate aggregation functions

Check my progress

## Congratulations!

You've learned about a number of techniques to potentially improve your query performance. While considering some of these techniques, remember the legendary computer scientist Donald Knuth's quote, "Premature optimization is the root of all evil."