SQL Query Optimization

How to write efficient SQL queries?

Throughout this lecture, we'll be using the following datasets from BigQuery's public data repository.

- ***crypto_ethereum*** dataset
- ***bikeshare*** tables from ***san_francisco*** dataset

Before you start writing your queries :

1. Click on more tab on your SQL workspace,
2. Select the Query settings option,
3. Go to Cache preference and
4. Uncheck Use cached results.

Remember that you'll have to repeat this process every time you open a new query editor.

Some common terminology definitions required in BigQuery :

- **Elapsed Time:**
    - Elapsed time, also known as query execution time, is the total time it takes for a query to run from the moment it is submitted until the results are returned.
    - It includes the time spent on processing, reading data, shuffling data (if necessary), and any other operations required to execute the query.

- **Slot Time Consumed:**
    - Slot time consumed refers to the amount of computational resources (slots) that a query consumes during its execution.
    - The more slots a query consumes, the faster it is likely to execute, but it also means higher costs, as you pay for the slots used.

- **Bytes Shuffled:**
  - The redistribution of data across different slots for processing is called shuffling, and it involves moving data between different parts of the distributed system.
  - Bytes shuffled represents the volume of data that is moved or shuffled between different computational nodes during query execution.

- **Bytes Spilled to Disk:**
  - When a query's intermediate results or data exceed the memory capacity of the allocated slots, some of that data may be written to disk temporarily for storage. This is referred to as spilling to disk.

Lower elapsed times, efficient slot usage, minimized bytes shuffled, and reduced bytes split to disk are all indicative of well-optimized and performant queries.

#1: Busting a common myth about the COUNT() function

Let us compare **COUNT(*)** vs. **COUNT(1)** vs. **COUNT(col_name)**

Which one do you think will take less time to execute?

1. **COUNT(*)** - When * is used as an argument, it simply counts the total number of rows including the NULLs.

2. **COUNT(1)** - With COUNT(1), there is a misconception that it counts records from the first column. What COUNT(1) really does is that it replaces all the records you get from query result with the value 1 and then counts the rows meaning it even replaces a NULL with 1 meaning it takes NULLs into consideration while counting.

3. **COUNT(col_name)** - When a column name is used as an argument, it simply counts the total number of rows excluding the NULLs meaning it will not take NULLs into consideration.

star_query =

```
SELECT COUNT(*) FROM
`bigquery-public-data.crypto_ethereum.blocks`
```

basic_query =

```
SELECT COUNT(1) FROM
`bigquery-public-data.crypto_ethereum.blocks`
```

```
------------------------
Elapsed Time: 8 sec
Slot Time: 2 min 30 sec
Bytes Shuffled: 26 KB
------------------------
```

No significant difference in performance is observed between the two queries.

#2: Only select the columns that you really need

It is tempting to start queries with SELECT * FROM ... It's convenient because you don't need to think about which columns you need. But it can be very inefficient.

basic_query =

```
SELECT * FROM
`bigquery-public-data.crypto_ethereum.blocks`
```

```
------------------------
Elapsed Time: 25 sec
Slot Time: 1 hr
Bytes Processed: 20 GB
Bytes Shuffled: 70 GB
------------------------
```

improved_query =

```
    SELECT
        timestamp,
        number,
        transactions_root, state_root, receipts_root,
        miner,
        difficulty, total_difficulty,
        size,
        extra_data,
        gas_limit, gas_used,
        transaction_count,
        base_fee_per_gas
    FROM `bigquery-public-data.crypto_ethereum.blocks`
```

```
-------------------------
Elapsed Time: 28 sec
Slot Time: 30 min
Bytes Processed: 6 GB
Bytes Shuffled: 20 GB
-------------------------
```

Best practice: It's better to select as few columns as possible.

#3: LIMIT is a trap

LIMIT speeds up performance, but doesn't reduce costs.

The row restriction of the LIMIT clause is applied after SQL databases scan the full range of data.

LIMIT clause speeds up performance by reducing shuffle time.

Since LIMIT puts a cap on the output rows, we need to move around less data on BigQuery's network. This reduction in bytes shuffled significantly improves query performance.

basic_query =

```
    SELECT miner
```

```
        FROM `bigquery-public-data.crypto_ethereum.blocks`
```

```
-------------------------
Elapsed Time : 8 sec
Slot Time : 5 min
Bytes Shuffled : 3 GB
-------------------------
```

improved_query =

```
        SELECT miner
        FROM `bigquery-public-data.crypto_ethereum.blocks`
        LIMIT 1000
```

```
-------------------------
Elapsed Time : 3 sec
Slot Time : 50 ms
Bytes Shuffled : 90 KB
-------------------------
```

Best practice: If your intention is purely to explore the table, you can consider using BigQuery's table preview option instead.

#4: Use EXISTS() instead of COUNT()

When exploring a dataset, sometimes we need to check for the existence of a specific value.

We have two choices, either to compute the frequency of the value with COUNT() , or to check if the value EXISTS().

Now, EXISTS() will exit its processing cycle as soon as it locates the first matching row, returning True if the target value is found, or False if the target value doesn't exist in the table.

On the contrary, COUNT() will continue to search through the entire table in order to return the exact number of occurrences for the target value, wasting unnecessary computing resources.

Question: We want to know if the value 6857606 exists in the number column.

basic_query =
```
        SELECT
                COUNT(number) AS count
        FROM `bigquery-public-data.crypto_ethereum.blocks`
        WHERE
                timestamp BETWEEN '2018-12-01' AND '2019-12-31'
                AND number = 6857606
```

Output: 1
```
-------------------------
Elapsed Time : 3 sec
Slot Time : 15 sec
Bytes Shuffled : 36 B
-------------------------
```

improved_query =
```
        SELECT EXISTS (
                SELECT
                        number
                FROM `bigquery-public-data.crypto_ethereum.blocks`
                WHERE
                        timestamp BETWEEN "2018-12-01" AND "2019-12-31"
                        AND number = 6857606 )
```

Output: True
```
-------------------------
Elapsed Time : 1 sec
Slot Time : 150 ms
Bytes Shuffled : 11 B
-------------------------
```

Best practice: If we don't need to know how frequently the value occurs, always use EXISTS() instead of COUNT().

#5: Use APPROX_COUNT_DISTINCT instead of COUNT(DISTINCT) for large datasets.

COUNT() scans the entire table to determine the number of occurrences. COUNT(DISTINCT) will need a massive amount of computer memory to keep count of unique values in a column.

APPROX_COUNT_DISTINCT() uses statistics to produce an approximate result instead of an exact result.

Question: We're interested in the number of unique Ethereum miners for the 2.2 million blocks.

basic_query =

```
    SELECT
        COUNT(DISTINCT miner)
    FROM `bigquery-public-data.crypto_ethereum.blocks`
    WHERE
        timestamp BETWEEN '2015-01-01' AND '2023-12-31'
```

```
-------------------------
Elapsed Time : 6 sec
Slot Time : 12 min
Bytes Shuffled : 14 MB
-------------------------
```

improved_query =

```
    SELECT
        APPROX_COUNT_DISTINCT(miner)
    FROM `bigquery-public-data.crypto_ethereum.blocks`
    WHERE
        timestamp BETWEEN '2015-01-01' AND '2023-12-31'
```

Elapsed Time : 5 sec
Slot Time : 2 min
Bytes Shuffled : 750 KB

Question: Find out in which major cities does the company 'Austin Bikes' operate and provide bike trips.

basic_query =

```
SELECT COUNT(DISTINCT landmark) as unique_cities
FROM `bigquery-public-data.san_francisco.bikeshare_stations`
```

Elapsed Time : 300 ms
Slot Time : 200 ms
Bytes Shuffled : 98 KB
----

improved_query =

```
SELECT APPROX_COUNT_DISTINCT(landmark) as unique_cities
FROM `bigquery-public-data.san_francisco.bikeshare_stations`
```

Elapsed Time : 200 ms
Slot Time : 100 ms
Bytes Shuffled : 48 KB

Best practice: In cases when data volumes are significant, do consider the option of trading accuracy for performance by utilizing the approximate aggregate functions.

#6: Replace Self-Join with Windows Function

Self-join usually requires more reads than windows function, therefore slower.

Question: We want to know the difference between the number of Ethereum blocks mined today and yesterday by each miner.

basic_query =

```
        WITH cte_table AS (
                SELECT
                        DATE(timestamp) AS date,
                        miner,
                        COUNT(DISTINCT number) AS block_count
                FROM `bigquery-public-data.crypto_ethereum.blocks`
                WHERE
                        DATE(timestamp) BETWEEN "2015-01-01" AND "2023-12-31"
                GROUP BY 1,2 )
        SELECT
                a.miner,
                a.date AS today,
                a.block_count AS today_count,
                b.date AS tmr,
                b.block_count AS tmr_count,
                b.block_count - a.block_count AS diff
        FROM cte_table a
        LEFT JOIN cte_table b
        ON
                DATE_ADD(a.date, INTERVAL 1 DAY) = b.date
                AND a.miner = b.miner
        ORDER BY
                a.miner, a.date
```

Elapsed Time : 5 sec

Slot Time : 22 sec

Bytes Shuffled : 12 MB

----

improved_query =

```
        WITH cte_table AS (
                SELECT
                DATE(timestamp) AS date,
                miner,
                COUNT(DISTINCT number) AS block_count
        FROM `bigquery-public-data.crypto_ethereum.blocks`
        WHERE
                DATE(timestamp) BETWEEN "2015-01-01" AND "2023-12-31"
        GROUP BY 1,2 )
        SELECT
                miner,
                date AS today,
                block_count AS today_count,
                LEAD(date, 1) OVER (PARTITION BY miner ORDER BY date) AS
        tmr,
                LEAD(block_count, 1) OVER (PARTITION BY miner ORDER BY
        date) AS tmr_count,
                LEAD(block_count, 1) OVER (PARTITION BY miner ORDER BY
        date) -
        block_count AS diff
        FROM cte_table a
```

Elapsed Time : 3 sec
Slot Time : 14 sec
Bytes Shuffled : 12 MB

Best practice: Self-joins are always inefficient and should only be used when absolutely necessary. In most cases, we can replace it with a Windows function.

#7: Trim your data early and often

Question: If we want to know the popularity of each GitHub repository, we can look at -
(i) the number of views and
(ii) the number of commits.

To extract the data, we can JOIN the `repos` and `commits` table then aggregate the counts with GROUP BY.
basic_query =

```
    WITH
    cte_repo AS (
        SELECT
            repo_name,
            watch_count
        FROM `bigquery-public-data.github_repos.sample_repos`),
    cte_commit AS (
        SELECT
            repo_name,
            `commit`
        FROM `bigquery-public-data.github_repos.sample_commits`)

    SELECT
        r.repo_name,
        r.watch_count,
        COUNT(c.commit) AS commit_count
    FROM cte_repo r
    LEFT JOIN cte_commit c
    ON r.repo_name = c.repo_name
    GROUP BY 1,2
```

```
-------------------------
Elapsed Time : 3 sec
Slot Time : 9 sec
Bytes Shuffled : 91 MB
-------------------------
```

To compare, we can implement GROUP BY earlier in the `commits` table.

improved_query =
```
WITH
cte_repo AS (
    SELECT
        repo_name,
        watch_count
    FROM `bigquery-public-data.github_repos.sample_repos`),
cte_commit AS (
    SELECT
        repo_name,
        COUNT(`commit`) AS commit_count
    FROM `bigquery-public-data.github_repos.sample_commits`
    GROUP BY 1)

SELECT
    r.repo_name,
    r.watch_count,
    c.commit_count
FROM cte_repo r
LEFT JOIN cte_commit c
ON r.repo_name = c.repo_name
```

```
-------------------------
Elapsed Time : 2 sec
Slot Time : 9 sec
Bytes Shuffled : 26 MB
-------------------------
```

Best practice: Apply filtering functions early and often in your query to reduce data shuffling and wasting compute resources on irrelevant data that doesn't contribute to the final query result.

#8: Use MAX() instead of RANK()

Question: The team has a general assumption that the older the establishment the more popular it'll be. To verify the same assumption, fetch the station ids and their respective date of installation in order starting from the one installed most recently.

basic_query =
```
    SELECT
        t.station_id, t.installation_date
    FROM (
        SELECT station_id, installation_date,
        RANK() OVER(PARTITION BY station_id ORDER BY
    installation_date DESC) AS rnk
        FROM `bigquery-public-data.san_francisco.bikeshare_stations`) t
    WHERE rnk = 1
    ORDER BY t.installation_date DESC
```

Elapsed Time : 650 ms
Slot Time : 13 sec
Bytes Shuffled : 3 KB
----

improved_query =
```
    SELECT
        station_id,
        MAX(installation_date) AS doi
    FROM `bigquery-public-data.san_francisco.bikeshare_stations`
    GROUP BY 1
    ORDER BY doi DESC
```

Elapsed Time : 280 ms
Slot Time : 90 ms
Bytes Shuffled : 3 KB

While using RANK(), first you need to pull all results, then rank them based on your partitioning & ordering and at the end specify a condition to fetch only the rank 1 record.

Whereas for MAX() you can just group by and fetch the record with maximum date i.e. the most recent from each group.

Also, using MAX() instead of RANK() function definitely seems simpler and more understandable.

#9: Order your JOINs from larger tables to smaller tables

Question: San Francisco is a big city and usually has a good number of bike trips. Find the number of bikes and docks currently available at all stations in San Francisco so that proper restocking can be done.

basic_query =

```
        SELECT t1.station_id, t1.name,
         t2.bikes_available, t2.docks_available
        FROM
         `bigquery-public-data.san_francisco.bikeshare_stations` t1
        JOIN
         `bigquery-public-data.san_francisco.bikeshare_status` t2
        ON t1.station_id = t2.station_id
        WHERE
         t1.landmark = 'San Francisco'
```

Elapsed Time : 13 sec
Slot Time : 2 min

Here **'bikeshare_status'** is the larger table and **'bikeshare_stations'** is the smaller one.

improved_query =

```
               SELECT t2.station_id, t2.name,
                t1.bikes_available, t1.docks_available
               FROM
                 `bigquery-public-data.san_francisco.bikeshare_status` t1
               JOIN
                 `bigquery-public-data.san_francisco.bikeshare_stations` t2
               ON t1.station_id = t2.station_id
               WHERE
                t2.landmark = 'San Francisco'
```

```
-------------------------
Elapsed Time : 9 sec
Slot Time : 2 min
-------------------------
```

Table join order matters for reducing the number of rows that the rest of the query needs to process.

In general practice, keeping the larger table in that initial query may boost the query performance.

Does WHERE sequence matters?

Speculated best practice: Expressions in your WHERE clauses should be ordered with the most selective expression first.

Google claims not only that using WHERE early in our query (on different tables) matters, but the sequence of WHERE within the same table also matters.

Is it better to apply the filtering clause first before the comparison clause?

Query1 =

```
          SELECT
                    miner
```

```
    FROM `bigquery-public-data.crypto_ethereum.blocks`
    WHERE
        miner LIKE '%a%'
        AND miner LIKE '%b%'
        AND miner = '0xc3348b43d3881151224b490e4aa39e03d2b1cdea'
```

```
-------------------------
Elapsed Time : 5 sec
Slot Time : 2 min 30 sec
Bytes Shuffled : 986 KB
-------------------------
```

Query2 =

```
    SELECT
        miner
    FROM `bigquery-public-data.crypto_ethereum.blocks`
    WHERE
        miner = '0xc3348b43d3881151224b490e4aa39e03d2b1cdea'
        AND miner LIKE '%a%'
        AND miner LIKE '%b%'
```

```
-------------------------
Elapsed Time : 6 sec
Slot Time : 3 min
Bytes Shuffled : 986 KB
-----------------------
```

Conclusion: It seems, the slot time and bytes shuffled are comparable for both queries, indicating that BigQuery's SQL Optimizer is smart enough to run the most selective WHERE clause regardless of how we wrote the query.

Should we push ORDER BY to the end of the query?

The reason why it's recommended to delay using ORDER BY until the outermost query is because tables tend to be bigger at the start of the query, given that they have not gone through any pruning from WHERE or GROUP BY clauses yet.

The bigger the table, the more comparisons it needs to do, and therefore the slower the performance.

Query1 =

```
WITH
cte_blocks AS (
    SELECT *
    FROM `bigquery-public-data.crypto_ethereum.blocks`
    WHERE
        DATE(timestamp) BETWEEN '2021-03-01' AND '2021-03-31'
    ORDER BY 1,2,3,4,5,6),
cte_contracts AS (
    SELECT *
    FROM `bigquery-public-data.crypto_ethereum.contracts`
    WHERE
    DATE(block_timestamp) BETWEEN '2021-03-01' AND '2021-03-31'
    ORDER BY 1,2,4,5,6,7)
SELECT *
FROM cte_blocks b
LEFT JOIN cte_contracts c
ON c.block_number = b.number
ORDER BY size, block_hash
```

```
-----------------------
Elapsed Time : 12 sec
Slot Time : 3 min
Bytes Shuffled : 5.9 GB
-----------------------
```

For comparison, we removed the pointless ORDER BY clauses from both cte_tables and ran the query again.

Query2 =

```
WITH
cte_blocks AS (
    SELECT *
    FROM `bigquery-public-data.crypto_ethereum.blocks`
```

```
        WHERE
            DATE(timestamp) BETWEEN '2021-03-01' AND '2021-03-31'),
    cte_contracts AS (
        SELECT *
        FROM `bigquery-public-data.crypto_ethereum.contracts`
        WHERE
        DATE(block_timestamp) BETWEEN '2021-03-01' AND '2021-03-31')
    SELECT *
    FROM cte_blocks b
    LEFT JOIN cte_contracts c
    ON c.block_number = b.number
    ORDER BY size, block_hash
```

Elapsed Time : 14 sec
Slot Time : 3 min
Bytes Shuffled : 5.9 GB

Digging into the execution details reveals that both queries only ran the ORDER BY in the outermost query, regardless of how we wrote the queries.

Conclusion: As it turns out, the almighty BigQuery’s SQL Optimizer is yet again smart enough to figure out the redundant clauses and automatically exclude them from the computation.
However, some other legacy databases may not have the same capability.

Best practice: Remove unnecessary ORDER BY clauses and use them only in the outermost query.
