#1: LIMIT is a booby trap.

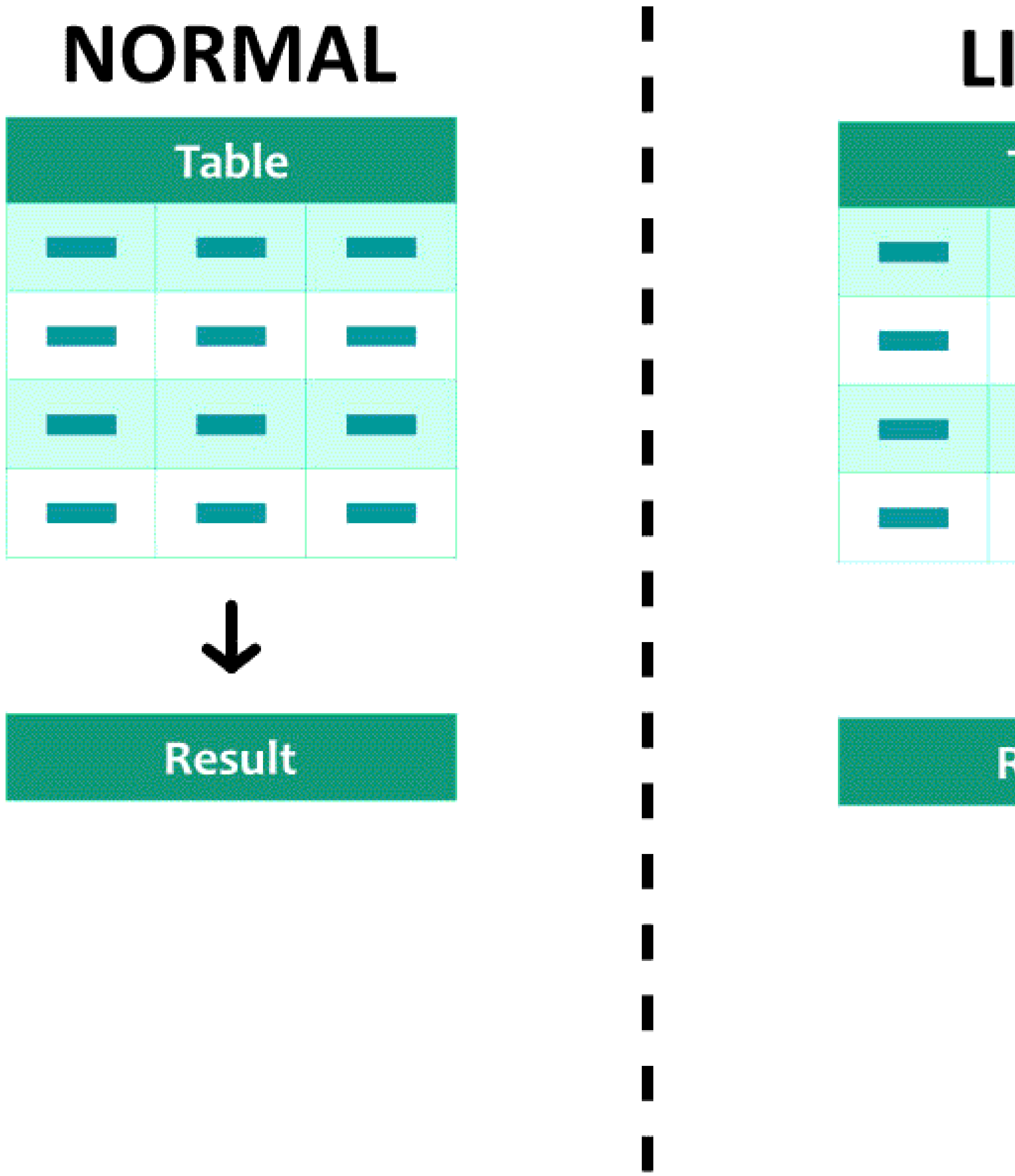
***Best practice**: `LIMIT` speeds up performance, but doesn't reduce costs. For data exploration, consider using BigQuery's (free) table preview option instead.*

It has to be said —

Most SQL practitioners were once prey to the illusion of safety misrepresented by `LIMIT 1000`. It's perfectly reasonable to assume that if we only show 1000 rows of outputs, there would be fewer loads on the database and hence lower cost.

Unfortunately, it isn't true.

The row restriction of `LIMIT` clause is applied *after* SQL databases scan the full range of data. Here's the kicker — most distributed database (including BigQuery) charges based on the data scans but *not* the outputs, which is why `LIMIT` doesn't help save a dime.



LIMIT clause speeds up performance by reducing shuffle time. Image by the author.

However, it's not all doom and gloom. 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.

To demonstrate, I'm using the `crypto_ethereum` table from BigQuery's public data repository which has 15 million rows of data.

```
# Not OptimizedSELECT
  miner
FROM
  `bigquery-public-data.crypto_ethereum.blocks`-----------------------
Elapsed Time   : 11s
Slot Time      : 162s
Bytes Processed: 617 MB
Bytes Shuffled : 1.7 GB
Bytes Spilled  : 0 B
-----------------------
```

Let's try the query again with `LIMIT`.

```
# Optimized (for speed only)SELECT
  miner
FROM
  `bigquery-public-data.crypto_ethereum.blocks`
LIMIT
  1000-----------------------
Elapsed Time   : 2s
Slot Time      : 0.01s
Bytes Processed: 617 MB
Bytes Shuffled : 92 KB
Bytes Spilled  : 0 B
-----------------------
```

Using `LIMIT` improved speed, but not cost.

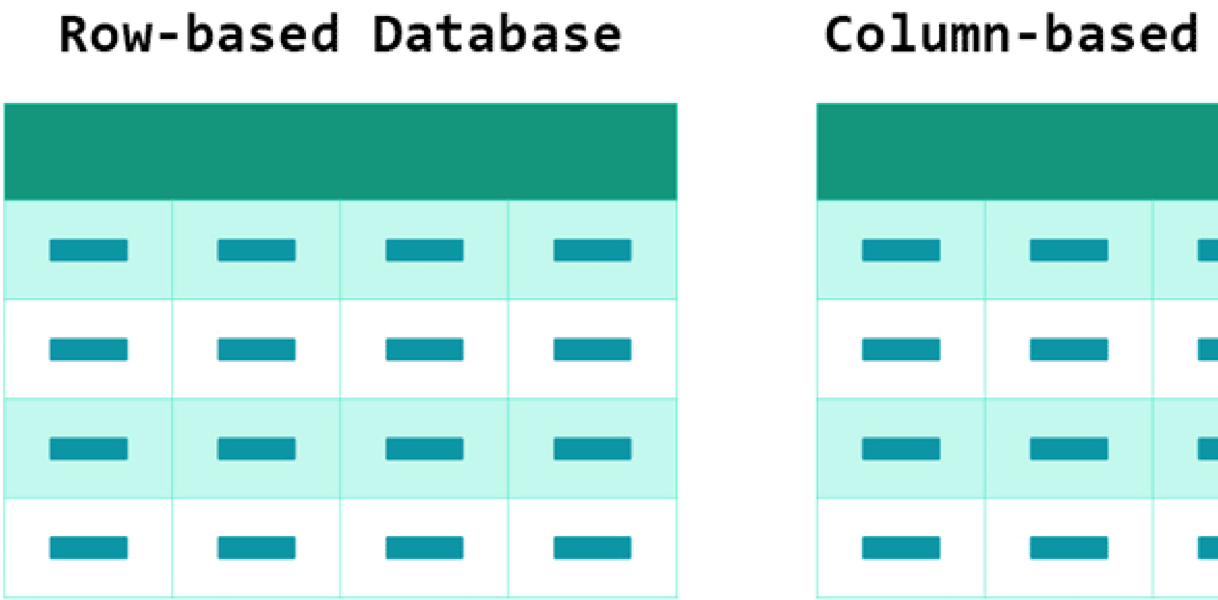
- Cost: Bytes processed remain the same at 617 MB.
- Speed: Bytes shuffled dropped from 1.7 GB to merely 92 KB, which explains the huge improvement in slot time (from 162s to 0.01s).

#2: SELECT as few columns as possible.

Best practice: *Avoid using* `SELECT *`*. Choose only the relevant columns that you need to avoid unnecessary, costly full table scans.* [*Source*].

BigQuery is not a traditional row-based database, but a [columnar] database. This distinction is meaningful because it reads data differently.

If a table has 100 columns but our query only needed data from 2 specific columns, a row-based database will go through each row — all 100 columns of each row—only to extract the 2 columns of interest. In contrast, a columnar database will process only the 2 relevant columns, which makes for a faster read operation and more efficient use of resources.



Row-based database and column-based database reads data differently. Image by the author.

Here is a typical query that is fast to write, but runs slow.

```
# Not OptimizedSELECT
  *
FROM
  `bigquery-public-data.crypto_ethereum.blocks`-----------------------
Elapsed Time   : 23s
Slot Time      : 31 min
Bytes Processed: 15 GB
Bytes Shuffled : 42 GB
Bytes Spilled  : 0 B
-----------------------
```

Since a columnar database can skip through columns, we can take advantage of this by querying *only* the columns we need.

```
# OptimizedSELECT
  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   : 35s
Slot Time      : 12 min
Bytes Processed: 5 GB
Bytes Shuffled : 11 GB
Bytes Spilled  : 0 B
-----------------------
```

In this example, query cost is reduced by 3x because the bytes we needed to process went down from 15 GB to 5 GB. On top of that, we also observe a performance gain with slot time decreasing from 31 minutes to 12 minutes.

The only downside of this approach is that we would need to type out the column names, which can be a hassle, especially when our tasks require most of the columns except a few. In cases like these, not all is lost, we can make use of `EXCEPT` statement to exclude the unnecessary columns.

```
# OptimizedSELECT
  *
  EXCEPT (
    `hash`,
    parent_hash,
    nonce,
    sha3_uncles,
    logs_bloom)
FROM
  `bigquery-public-data.crypto_ethereum.blocks`-----------------------
Elapsed Time   : 35s
Slot Time      : 12 min
Bytes Processed: 5 GB
Bytes Shuffled : 11 GB
Bytes Spilled  : 0 B
-----------------------
```

Avoid `SELECT *` unless absolutely necessary.

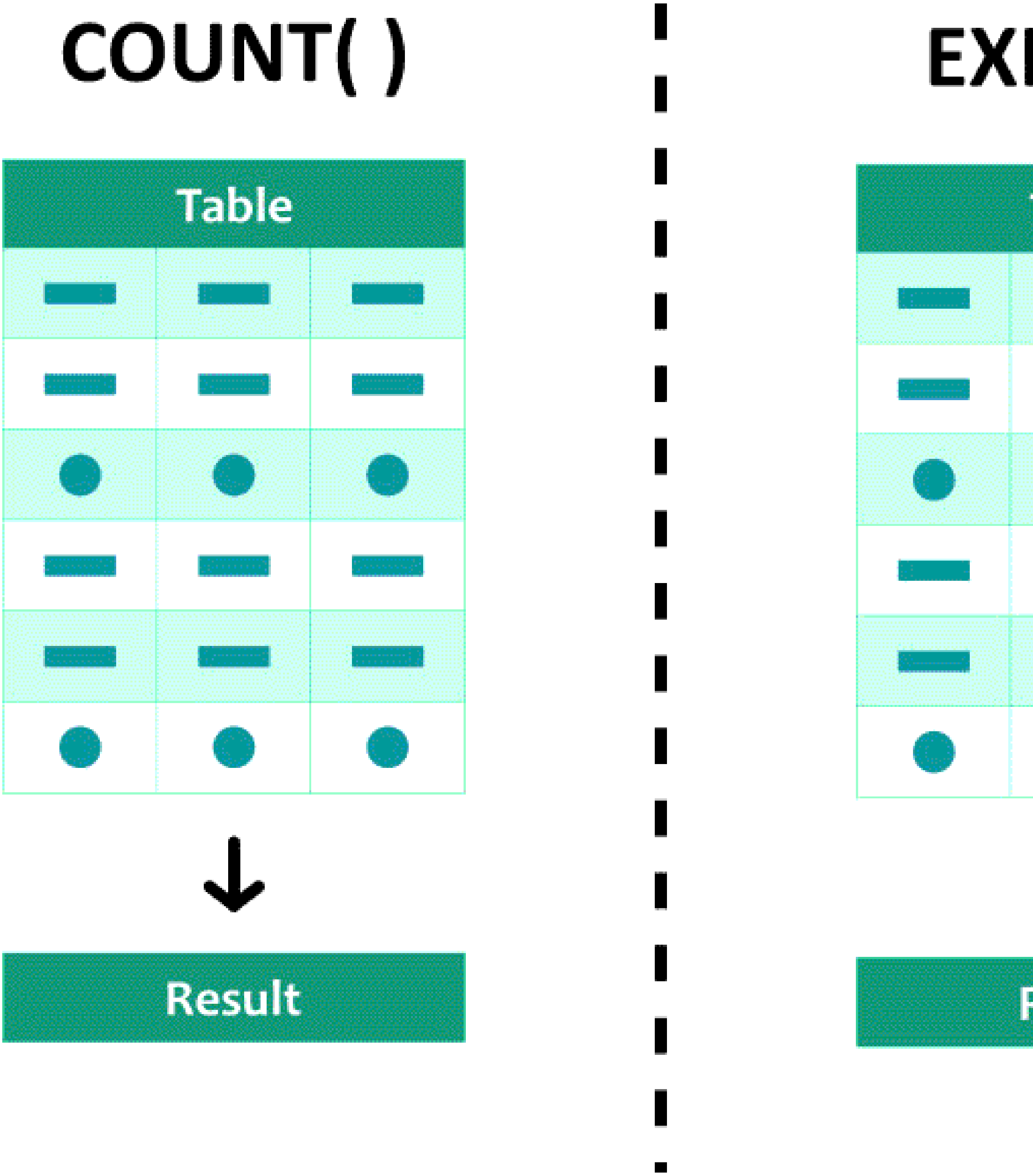
#3: Use EXISTS() instead of COUNT().

Best practice*: If we don't need the exact count, use `EXISTS()` because it exits the processing cycle as soon as the first matching row is found. Source.*

When exploring a brand new dataset, sometimes we find ourselves needing 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()`. If we don't need to know how frequently the value occurs, always use `EXISTS()` instead.

This is because `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.



EXISTS() clause exits processing as soon as a match is found. Image by the author.

Suppose that we want to know if the value `6857606` exists in the `number` column and we used `COUNT()` function...

```
# Not OptimizedSELECT
  COUNT(number) AS count
FROM
  `bigquery-public-data.crypto_ethereum.blocks`
WHERE
  timestamp BETWEEN '2018-12-01' AND '2019-12-31'
  AND number = 6857606-----------------------
Elapsed Time   : 6s
Slot Time      : 16s
Bytes Processed: 37 MB
Bytes Shuffled : 297 B
Bytes Spilled  : 0 B
-----------------------
```

The `COUNT()` returned 1 because only one row matches the value. Now, let's try with `EXISTS()` instead.

```
# OptimizedSELECT EXISTS (
  SELECT
    number
  FROM
    `bigquery-public-data.crypto_ethereum.blocks`
  WHERE
    timestamp BETWEEN "2018-12-01" AND "2019-12-31"
    AND number = 6857606
)-----------------------
Elapsed Time   : 0.7s
Slot Time      : 0.07s
Bytes Processed: 37 MB
Bytes Shuffled : 11 B
```

```
Bytes Spilled  : 0 B
----------------------
```

The query returns `True` because the value exists in the table. With `EXISTS()` function, we don't get information on its frequency, but in return, the query performance improved massively — from 16 seconds to just 0.07 seconds.

Aren't you glad that `EXISTS()` function exists?

#4: Use Approximate Aggregate Function.

***Best practice**: When you have a big dataset and you don't need the exact count, use approximate aggregate functions instead. Source*.

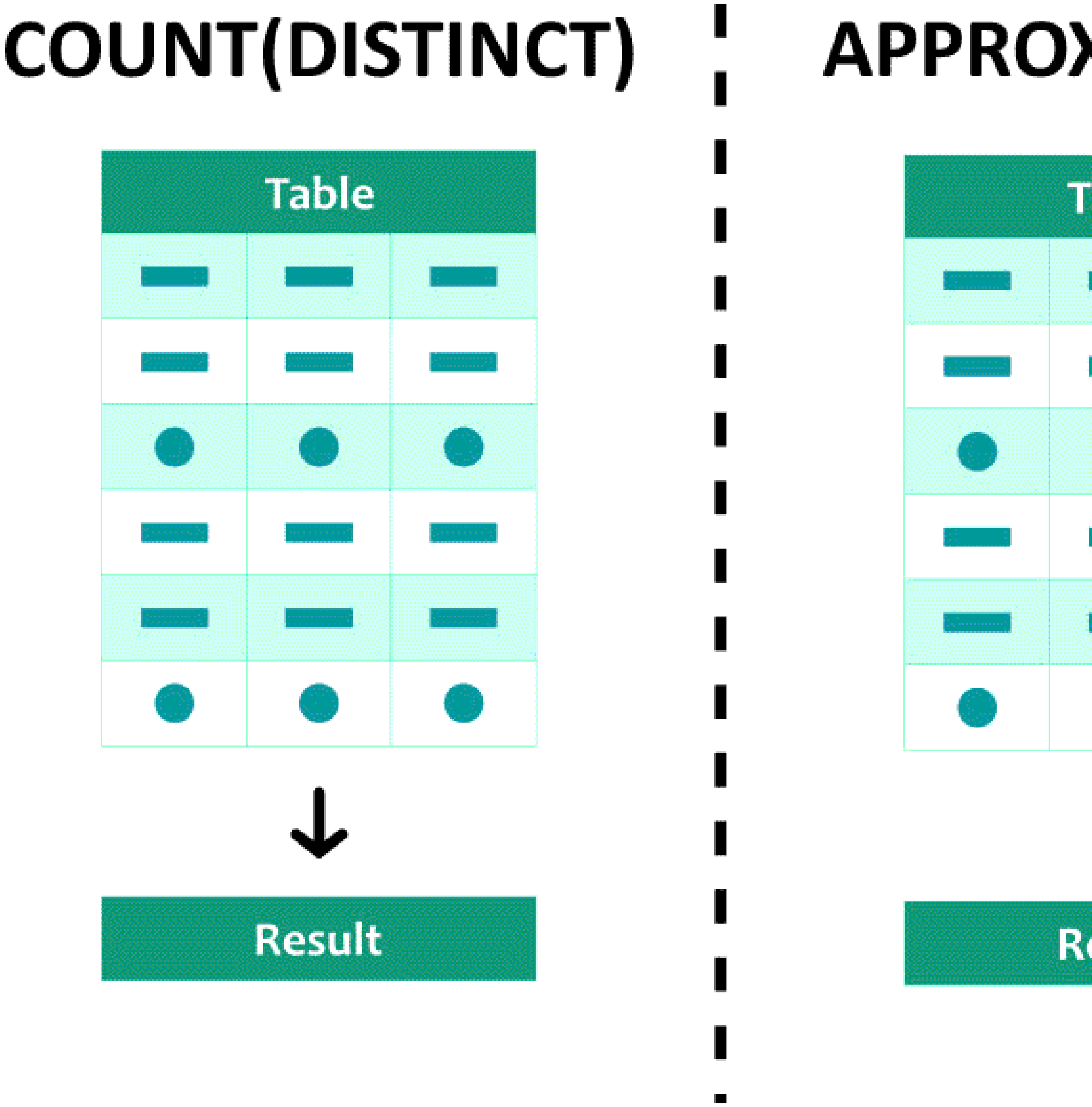
A `COUNT()` scans the entire table to determine the number of occurrences. Since this is done row-by-row, the operations will run at a time-space complexity of O(n). Performing such an operation on big data with hundreds of millions of rows will quickly become unfeasible as it requires massive amounts of computing resources.

To exacerbate the performance issue, `COUNT(DISTINCT)` will need an ungodly amount of computer memories to keep count of the unique ids of every user. When the list exceeds the memory capacity, the surplus will spill into disks, causing performance to take a nosedive.

In cases when data volumes are significant, it may be in our best interest to trade accuracy for performance by using approximate aggregation functions. For example:-

- APPROX_COUNT_DISTINCT()
- APPROX_QUANTILES()
- APPROX_TOP_COUNT()
- APPROX_TOP_SUM()
- HYPERLOGLOG++

Unlike the usual brute-force approach, approximate aggregate functions use statistics to produce an approximate result instead of an exact result. Expects the error rate to be 1~2%. Since we are not running a full table scan, approximate aggregate functions are highly scalable in terms of memory usage and time.



Approximate Aggregate Function use statistics to provide an approximate result fast. [Magnifier]() icon by Freepik from Flaticon, edited with permission by the author.

Suppose that we are interested in the number of unique Ethereum miners for the 2.2 million blocks, we can run the following query…

```
# Not OptimizedSELECT
  COUNT(DISTINCT miner)
FROM
```

```
  `bigquery-public-data.crypto_ethereum.blocks`
WHERE
  timestamp BETWEEN '2019-01-01' AND '2020-01-01'-----------------------
Elapsed Time   : 3s
Slot Time      : 14s
Bytes Processed: 110 MB
Bytes Shuffled : 939 KB
Bytes Spilled  : 0 B
-----------------------
```

The `COUNT(DISTINCT)` function returned 573 miners but took 14s to do it. We can compare that to `APPROX_COUNT_DISTINCT()`.

```
# OptimizedSELECT
  APPROX_COUNT_DISTINCT(miner)
FROM
  `bigquery-public-data.crypto_ethereum.blocks`
WHERE
  timestamp BETWEEN '2019-01-01' AND '2020-01-01'-----------------------
Elapsed Time   : 2s
Slot Time      : 7s
Bytes Processed: 110 MB
Bytes Shuffled : 58 KB
Bytes Spilled  : 0 B
-----------------------
```

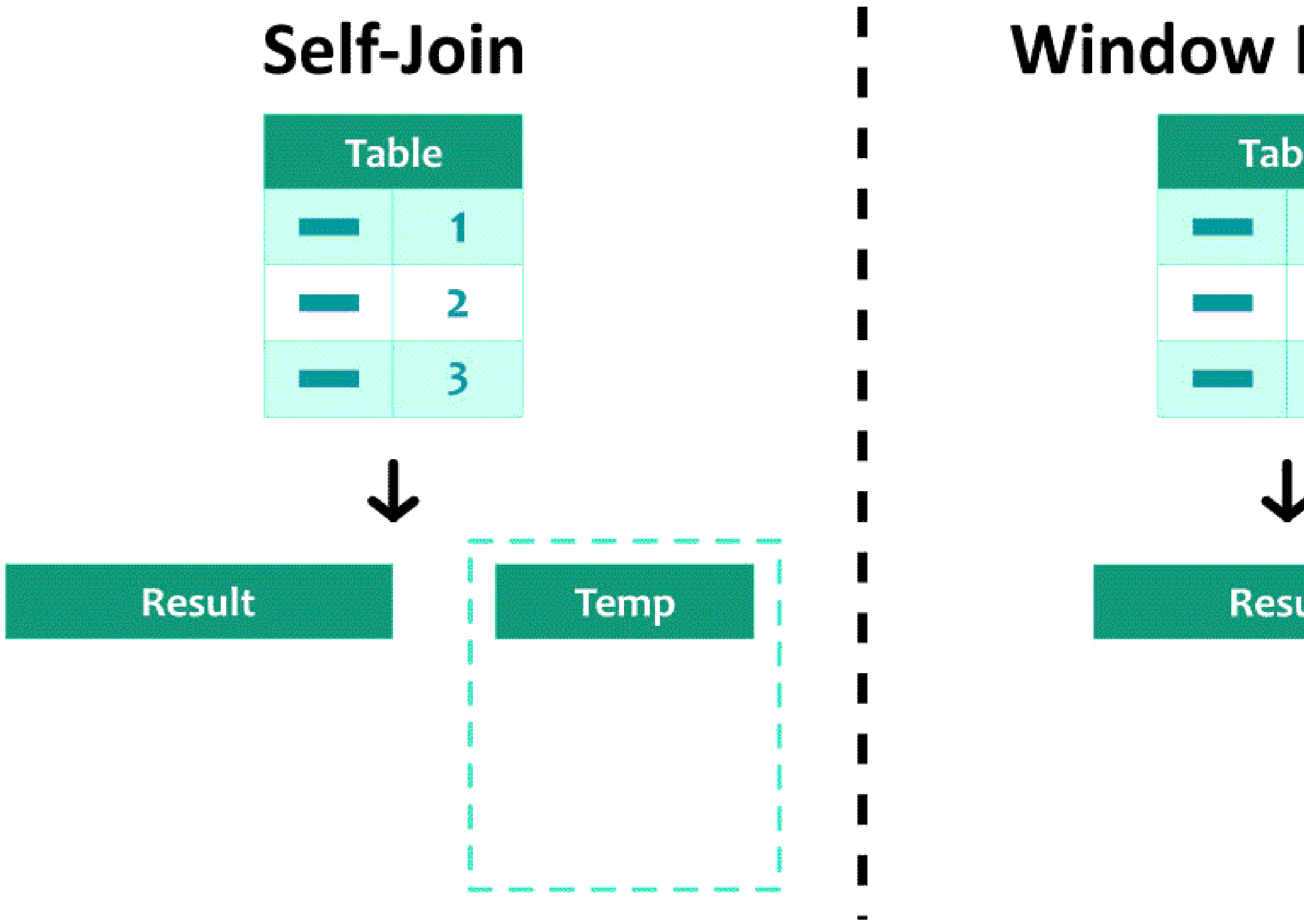
Much to my delight, the `APPROX_COUNT_DISTINCT()` returned the correct count of 573 miners (luck?) in half the slot time. The difference in performance is clear even with just 2.2 million rows of data but I’d imagine that difference will widen in our favor as the table gets bigger.

Whenever super-precise calculations are not needed, do consider utilizing approximate aggregate functions for a much higher level of responsiveness.

#5: Replace Self-Join with Windows Function.

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

A self-join is when a table is joined with itself. This is a common join operation when we need a table to reference its own data, usually in a parent-child relationship.



Self-join usually requires more reads than windows function, therefore slower. Image by the author.

A common use case — an Employee table with a manager_id column would contain row records of all employees, and assistant managers (who are also employees of the companies), who may also have a manager of their own. To get a list of all employees and their direct supervisors, we can perform a self-join with employee_id = manager_id.

This is typically a SQL anti-pattern because it can potentially square the number of output rows, or forces a lot of unnecessary reads, which slows our query performance exponentially as the table gets bigger.

For example, if we want to know the difference between the number of Ethereum blocks mined today and yesterday by each miner, we could write a self-join, although it is inefficient:-

```
# Not OptimizedWITH
 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 "2022-03-01"
  AND "2022-03-31"
 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   : 12s
Slot Time      : 36s
Bytes Processed: 12 MB
Bytes Shuffled : 24 MB
Bytes Spilled  : 0 B
-----------------------
```

Rather than performing a self-join, a window function in combination with a navigation function, `LEAD()`, would be a much better approach.

```
# OptimizedWITH
 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 "2022-03-01" AND "2022-03-31"
   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   : 3s
Slot Time      : 14s
Bytes Processed: 12 MB
Bytes Shuffled : 12 MB
Bytes Spilled  : 0 B
-----------------------
```

Both the queries gave us the same result, but there is a significant improvement in query speed (from 36 seconds slot time to 14 seconds slot time) with the latter approach.

Other than the `LEAD()` function, there is plenty of other navigation, numbering, and aggregate analytics functions that can be used in place of self-join operations. Personally, these are the functions that I use frequently in my day-to-day tasks:-

- Navigation Function: `LEAD()`, `LAG()`
- Numbering Function: `RANK()`, `ROW_NUMBER()`

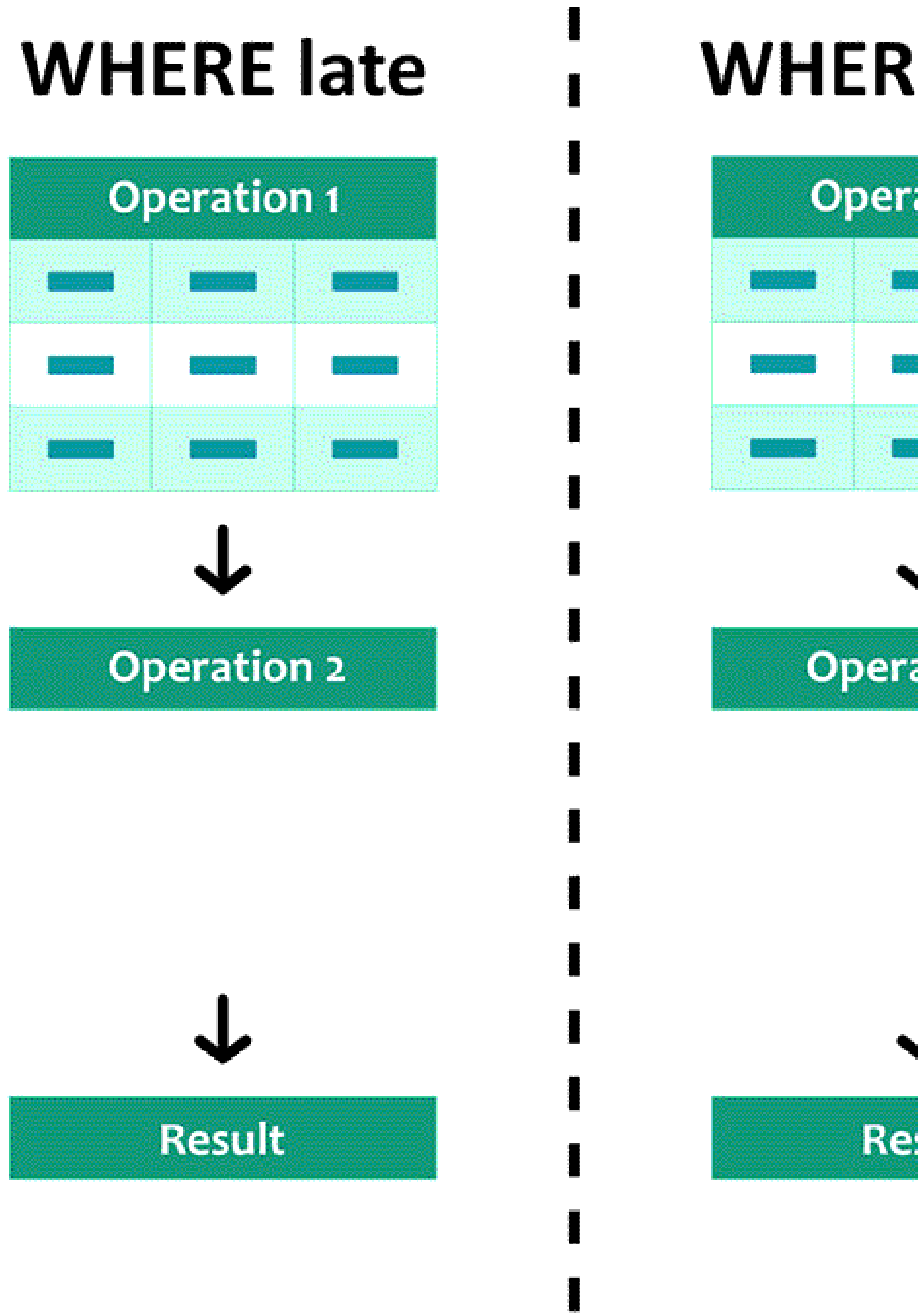
- Aggregate Analytics Function: `SUM()`, `AVG()`, `MAX()`, `MIN()`, `COUNT()`

The next time you see a self-join, remind yourself they are just windows of opportunity to flex your mastery of windows function.

#8: Trim your data early and often.

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.*

I'll sound like a broken record, but great advice is worth repeating — trim your data with `SELECT DISTINCT`, `INNER JOIN`, `WHERE`, `GROUP BY`, or any other filtering function whenever you get the chance. The earlier we do it, the lesser the load on every subsequent stage of our query, therefore compounding the performance gain every step of the way.


WHERE late
Operation 1
Operation 2
Result
WHER
Opera
Opera
Res

Trimming irrelevant data early saves compute resources downstream. Image by the author.

For instance, 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`.

```
# Not OptimizedWITH
  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   : 3s
Slot Time      : 8s
Bytes Processed: 50 MB
Bytes Shuffled : 91 MB
Bytes Spilled  : 0 B
-----------------------
```

In this scenario, the `GROUP BY` clause was performed in the outermost query so every row of commits is `JOIN` to the repository first. Since multiple commits can belong to the same repository, this results in an exponentially larger table that we need to `GROUP BY`.

For comparison, we can implement `GROUP BY` earlier in the `commits` table.

```
# OptimizedWITH
  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`
```

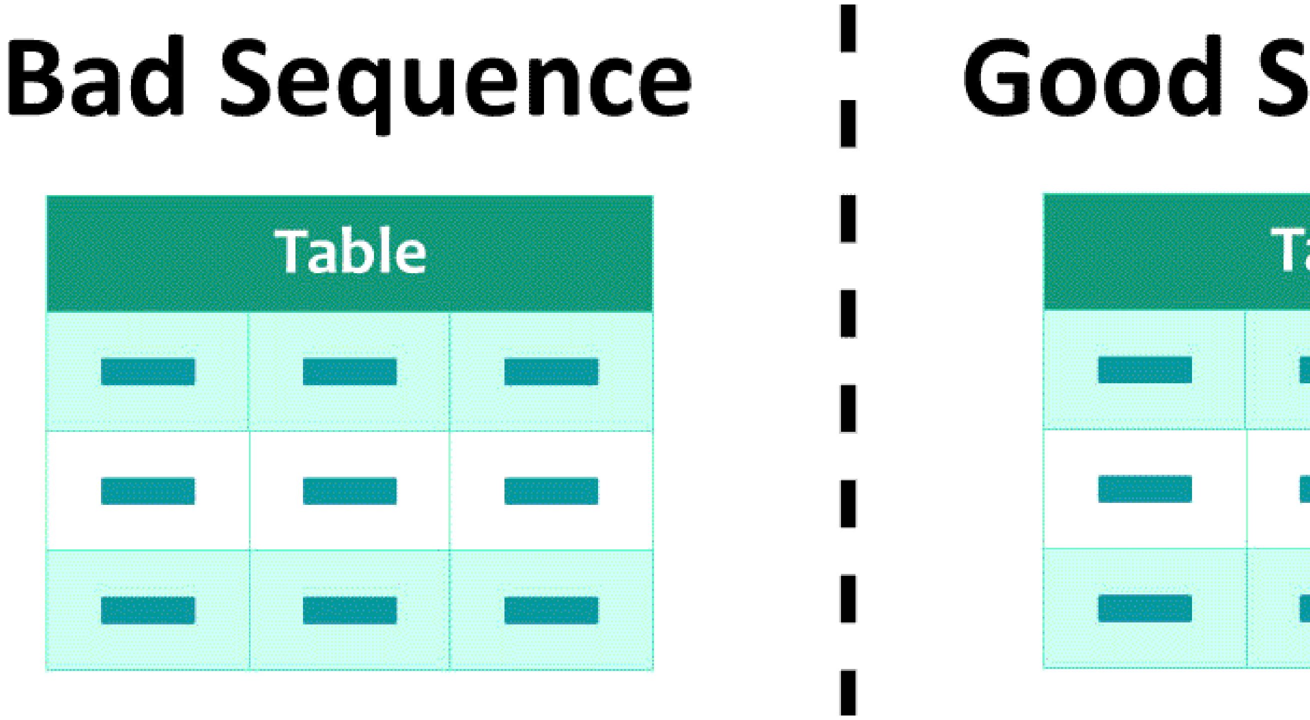
```
  GROUP BY
    1
 )SELECT
 r.repo_name,
 r.watch_count,
 c.commit_count
FROM
 cte_repo r
LEFT JOIN
 cte_commit c ON r.repo_name = c.repo_name-----------------------
Elapsed Time   : 2s
Slot Time      : 5s
Bytes Processed: 50 MB
Bytes Shuffled : 26 MB
Bytes Spilled  : 0 B
-----------------------
```

We see a huge improvement in Slot Time and Bytes Shuffled when we `GROUP BY` early. This is because all the commits are condensed from 672,000 records to 6 records so there are less data to move around.

#9: WHERE sequence matters (?)

Speculated best practice: *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.* [*Source*](Source).

This recommendation piques my interest because if it was true, it would be the easiest implementation with huge potential for optimization improvement. Google claims not only that using WHERE early in our query (on different tables) matter, but the sequence of WHERE within the same table also matters.



Is it better to apply the filtering clause first before the comparison clause? Image by the author.

I've decided to test it out myself.

```
# "Supposedly" Not OptimizedSELECT
  miner
FROM
  `bigquery-public-data.crypto_ethereum.blocks`
WHERE
  miner LIKE '%a%'
```

```
  AND miner LIKE '%b%'
  AND miner = '0xc3348b43d3881151224b490e4aa39e03d2b1cdea'-----------------------
Elapsed Time   : 7s
Slot Time      : 85s
Bytes Processed: 615 MB
Bytes Shuffled : 986 KB
Bytes Spilled  : 0 B
-----------------------
```

Amongst the three `WHERE` clauses that we use, the `LIKE` operators are string-comparison operations that are expensive to run, while the `=` operator selects a very specific miner which drastically reduces the number of relevant rows.

In an ideal state, the `=` operator will be executed before the other two so that the expensive `LIKE` operations will only be performed on the subset of the remaining rows.

If the sequence of `WHERE` do matters, then the query performance above should be indisputably worse than a similar query with the `=` operator as the first.

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

```
Elapsed Time   : 8s
Slot Time      : 92s
Bytes Processed: 615 MB
Bytes Shuffled : 986 KB
Bytes Spilled  : 0 B
----------------------
```

But as 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.

Tip 10: USE MAX() instead of RANK().

Objective 10. -
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.

How would you do that?

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
```

Time to run: 550-600 ms

How can we achieve this in less time?

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
```

Time to run: 300 ms