# dbt --empty flag

You can use the --empty flag to dry run the model.

What does dry run means ?

The --empty flag will put `WHERE FALSE LIMIT 0` at the end of all the `SELECT` statement in your query.

In SQL, `WHERE FALSE LIMIT 0` is an unusual but valid query structure. It's often used in certain contexts for specific reasons. Let's break it down:

1. `WHERE FALSE`: This part of the query effectively filters out all rows because the condition is always false. So, no rows will be selected by this query.
2. `LIMIT 0`: This limits the result set to zero rows. Even if there were rows that could match other criteria, `LIMIT 0` ensures that the query will return no rows.

## Why Use `WHERE FALSE LIMIT 0`?

This kind of query is sometimes used in automated scripts, query builders, or complex SQL generation scenarios where an initial or placeholder query needs to be generated without actually fetching data. Here are some reasons it might appear:

- **Testing query syntax**: Sometimes developers or tools want to test the SQL syntax to ensure that the query compiles without executing it on the actual data.
- **Placeholders in query builders**: In dynamic SQL generation, an initial query might be generated with `WHERE FALSE LIMIT 0` and then expanded or modified later when conditions are actually added.
- **Schema inspection**: Some tools use queries like this to inspect the structure of the result set, such as column names and types, without pulling any data. It provides the metadata without the overhead of data transfer.

In Google BigQuery, a query with `WHERE FALSE LIMIT 0` will not process any actual data rows, meaning it won't incur the same data processing costs as a regular query. It provides the metadata without the overhead of data transfer. BigQuery is optimized to recognize that no rows are being selected, and it typically skips scanning or processing underlying data for such queries.

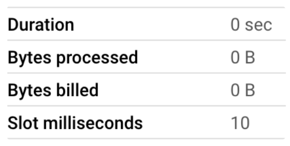
However, there are a few key aspects to keep in mind:

1. **Metadata and Schema Processing**:
   - BigQuery will still analyze the query structure to understand the table schema, column names, and data types. This metadata processing doesn't involve actual data scanning or row access, so it doesn't incur charges related to reading data.
2. **Minimal Cost or Free Query**:
   - Because no data rows are scanned or processed, queries like `WHERE FALSE LIMIT 0` typically result in no charges for data scanning. In BigQuery, you are charged based on the amount of data read or processed, so filtering out all rows with `WHERE FALSE` generally keeps the cost at or near zero.
3. **Use in Schema Inspection**:
   - Queries like this can still be useful for inspecting table structure without incurring data processing costs. For example, if you want to see the columns in a table without loading any rows, `SELECT * FROM my_table WHERE FALSE LIMIT 0` allows you to preview the schema structure without triggering a data scan.

Let's take an example query in bigquery:

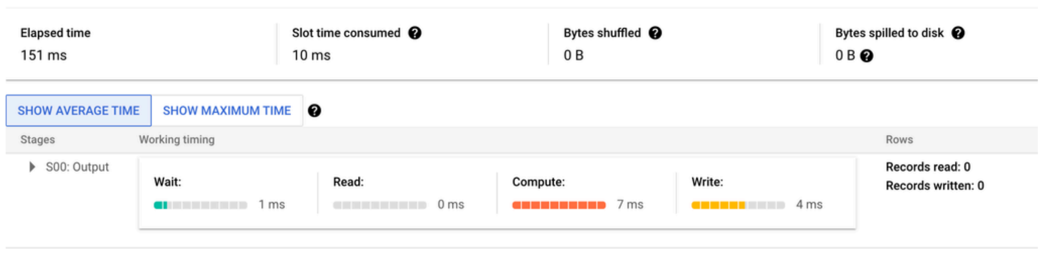
```
with model_a as (
  SELECT * FROM `[redacted]` WHERE FALSE LIMIT 0
),

model_b as (
  SELECT order_transaction_date_timestamp FROM model_a
)

select * from model_b
```

Lets inspect what happens when you execute this query:



| | |
|---|---|
| Duration | 0 sec |
| Bytes processed | 0 B |
| Bytes billed | 0 B |
| Slot milliseconds | 10 |

Job information

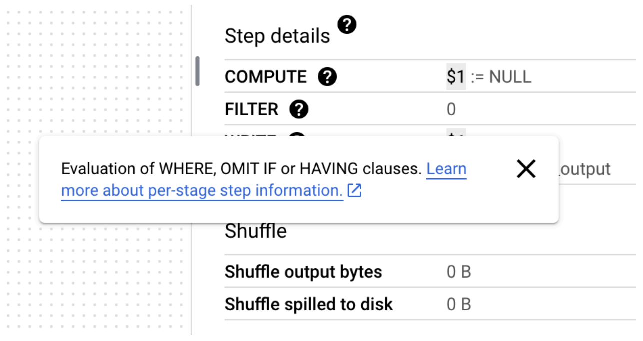


Elapsed time 151 ms | Slot time consumed 10 ms | Bytes shuffled 0 B | Bytes spilled to disk 0 B

SHOW AVERAGE TIME | SHOW MAXIMUM TIME

| Stages | Working timing | | | | Rows |
|---|---|---|---|---|---|
| S00: Output | Wait: 1 ms | Read: 0 ms | Compute: 7 ms | Write: 4 ms | Records read: 0<br>Records written: 0 |

Execution Details



Step details

| | |
|---|---|
| COMPUTE | $1 := NULL |
| FILTER | 0 |

Evaluation of WHERE, OMIT IF or HAVING clauses. Learn more about per-stage step information.

Shuffle

| | |
|---|---|
| Shuffle output bytes | 0 B |
| Shuffle spilled to disk | 0 B |

Execution graph

As you can see, there is no evaluation of WHERE clause. Also look at the compute part. Although this was a simple CTE. But lets explain what ‘COMPUTE’ means in the execution graph:

In BigQuery's execution graph, the **COMPUTE** step refers to operations where expressions and SQL functions are evaluated on the data. This is distinct from **SCAN** (where data is read from storage) and **WRITE** (where data is saved to a destination table). The **COMPUTE** stage focuses on calculations and transformations applied to the data after it has been scanned and loaded into memory.

## What Happens in the COMPUTE Stage?

During the **COMPUTE** step, BigQuery performs a range of processing tasks, which may include:

- **Expression Evaluation**: Calculating or transforming data based on expressions.
- **SQL Functions**: Applying built-in SQL functions, like mathematical operations, string functions, date functions, etc.
- **JOINs, Aggregations, and Sorting**: Executing operations that require processing data in memory, such as joining tables, aggregating rows, and ordering results.

This stage doesn't involve scanning or reading additional data from storage but instead uses the data already brought into memory. It’s typically CPU-intensive rather than I/O-intensive.

### Example: Understanding COMPUTE in a Query

Suppose you have a `sales` table with columns `product_id`, `quantity`, `unit_price`, and `sale_date`. Let's look at a few query examples and how **COMPUTE** steps would work in each.

#### Example 1: Basic Calculation

```sql
SELECT
  product_id,
  quantity * unit_price AS total_sale
FROM
  sales;
```

In this query:

- **SCAN**: BigQuery reads `product_id`, `quantity`, and `unit_price` columns from the `sales` table.
- **COMPUTE**: BigQuery evaluates the expression `quantity * unit_price` to calculate `total_sale` for each row. The **COMPUTE** step includes the multiplication operation for every row in the result set.

#### Example 2: Aggregation and Filtering

```sql
SELECT
  product_id,
  SUM(quantity * unit_price) AS total_sales
FROM
  sales
WHERE
  sale_date >= '2023-01-01'
GROUP BY
  product_id;
```

Here's how BigQuery executes this:

- **SCAN**: Reads `product_id`, `quantity`, `unit_price`, and `sale_date` columns.
- **COMPUTE**:
  - **Filtering**: Evaluates the `WHERE` condition `sale_date >= '2023-01-01'`.
  - **Expression Evaluation**: Calculates `quantity * unit_price` for each row that meets the `WHERE` condition.
  - **Aggregation**: Sums up the `quantity * unit_price` values grouped by `product_id`.

The **COMPUTE** phase here is more complex because it involves both filtering, expression evaluation, and grouping/aggregation.

#### Example 3: Using SQL Functions

```sql
SELECT
  product_id,
  COUNT(*) AS sale_count,
  MAX(DATE_DIFF(CURRENT_DATE(), sale_date, DAY)) AS days_since_last_sale
FROM
  sales
GROUP BY
  product_id;
```

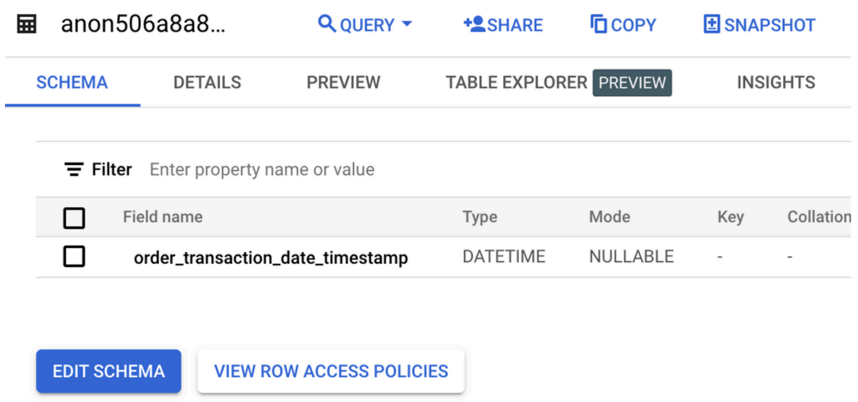
In this example:

- **SCAN**: Reads the `product_id` and `sale_date` columns.
- **COMPUTE**:
  - **COUNT**: Counts the number of rows for each `product_id`.

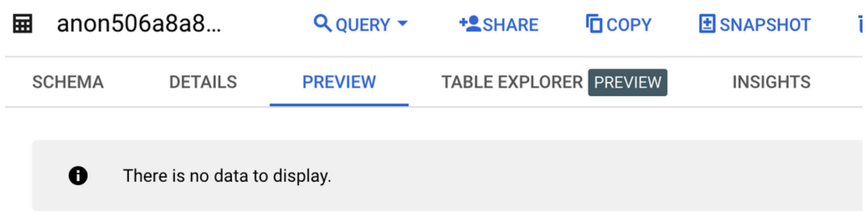
- **MAX and DATE_DIFF**: Calculates the days between `CURRENT_DATE()` and `sale_date` for each row, then finds the maximum value within each group (i.e., days since the last sale for each product).

Each of these function calls (e.g., `COUNT`, `MAX`, `DATE_DIFF`) happens in the **COMPUTE** step, making this stage more CPU-intensive than if no functions were applied.

Alright! Let's get back to --empty flag. When bigquery executes a query that has `WHERE FALSE LIMIT 0`, does it generate the temporary result table ? Yes it does.



| Field name | Type | Mode | Key | Collation |
|---|---|---|---|---|
| order_transaction_date_timestamp | DATETIME | NULLABLE | - | - |



There is no data to display.

As you can see above, the temporary result table got created with no data. That means the metadata, schema were evaluated. So, if you write the query in the wrong way i.e put a wrong cast, then it will still throw error.

```
)
```