- **EXPLAIN:**

- Shows the **planned execution steps** of a query **without running it**.
- Only estimates — based on statistics from `ANALYZE`.
- Output includes: which indexes/joins/scans will be used, and estimated rows/cost.

- **EXPLAIN ANALYZE :**

- Runs the query **and** shows the actual execution steps + timing + row counts.
- Includes both **estimates** and **real performance numbers**.
- Useful to check if the planner's estimates match reality.

Key Components of the Query Plan

1. **Node Type:** Each line in the query plan represents a "node," indicating the operation being performed (e.g., Seq Scan, Index Scan, Hash Join, etc.).
2. **Relation Name:** The table or relation involved in the operation.
3. **Alias:** If an alias is used in the query, it is displayed here.
4. **Actual Rows:** The number of rows processed by the node during execution (in EXPLAIN ANALYZE).
5. **Actual Loops:** The number of times the node was executed (for nested loop joins).
6. **Cost:** The estimated cost of the operation in terms of disk blocks and CPU cycles, represented as two numbers: startup cost and total cost.
7. **Start Time:** The time taken for the node to start execution (in EXPLAIN ANALYZE).
8. **End Time:** The time taken for the node to complete execution (in EXPLAIN ANALYZE).

Buffers: Indicates the memory consumed by the query, detailing shared buffers hit, read, and temporary buffers used.

1. Sequential Scan

Reads the whole table row by row.
Applies filters while reading, but can't skip rows.
Best for small tables or when most rows are needed.

2. Index Scan

Uses an index to find relevant rows.
Fetches table rows to check visibility and get missing columns.
Fast when the query is selective.

3. Index-Only Scan

Reads directly from the index, no table access needed.
Very fast, but only works if all required columns are in the index.

4. Bitmap Heap Scan

Uses a bitmap from one or more indexes to fetch rows efficiently.
Handles multiple conditions well.
May fetch full pages if memory is limited, then re-check conditions.

```
EXPLAIN (ANALYZE, BUFFERS)SELECT * FROM orders WHERE customer_id = 123;
```

Sample Output :

```
Index Scan using idx_orders_customer_id on orders
(cost=0.42..8.44 rows=3 width=64)
(actual time=0.030..0.060 rows=3 loops=1)
Buffers: shared hit=4
```

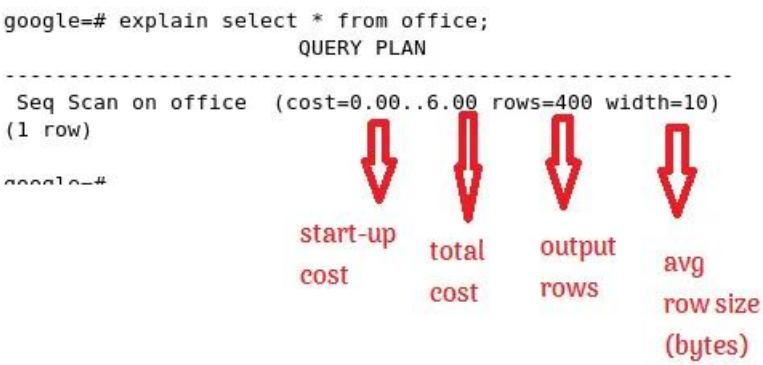
❖ **Cost:-**

1.**start-up cost** is the estimated Postgres computational units to start up a node tostart process the query.

2.**Total cost** is the estimated Postgres computational units to finish process thequery and return results.

3.**output rows**is the estimated number of rows returned.

5. **width or average row size (in bytes)**is the estimated size of each output row.



Explain plan in postgresql

- ❖ cost=0.42..8.44

- These are the planner's estimated costs of executing the operation.
- The first number (0.42) is the startup cost – the cost before producing the first row.
- The second number (8.44) is the total cost – the cost to produce all rows.
- These are unitless values used for comparing query plans, not actual time in ms.

- ❖ **rows=3**

Estimated number of rows that PostgreSQL thinks will be returned by this step.

Here it estimates ~3 rows.

- ❖ **width=64**

The average size (in bytes) of each row in this step of the plan.

Here PostgreSQL thinks each row is ~64 bytes wide.

- ❖ **actual time=0.030..0.060 rows=3:**

The query actually returned 3 rows in ~0.06 ms

- ❖ **loops=1:**

This operation ran once. - In nested loops, this can be higher.

- ❖ **Buffers: shared hit=4 :**

All pages were found in memory (shared buffer cache).

1. hit : read from memory
2. read : read from disk

3. written : pages written to disk

Ideally, you want high hit values and low reads to minimize disk I/O.

Sample Query Plan

To illustrate the components of a query plan, let's consider the following SQL query:

```
SELECT o.order_id, c.customer_name, SUM(oi.quantity)FROM orders oJOIN order_items oi ON
o.order_id = oi.order_idJOIN customers c ON o.customer_id = c.customer_idWHERE o.order_date >
'2023-01-01'GROUP BY o.order_id, c.customer_nameORDER BY SUM(oi.quantity) DESC;
```

Running EXPLAIN ANALYZE for this query might yield an output like the following:

```
Copy                                    QUERY PLAN                                    ----------------------------------
-----------------------------------------------------------------------------------------
 Sort  (cost=234.50..235.50 rows=1000 width=64) (actual time=10.235..10.735 rows=1000 loops=1)
   Sort Key: sum(oi.quantity) DESC
   Sort Method: quicksort  Memory: 1200kB
   Buffers: shared hit=100 read=50, temp written=0
   ->  Hash GroupAgg  (cost=234.50..235.00 rows=1000 width=64) (actual time=10.180..10.250
rows=1000 loops=1)
         Group Key: o.order_id, c.customer_name
         Buffers: shared hit=100 read=50
         ->  Hash Join  (cost=233.00..234.00 rows=1000 width=64) (actual time=9.800..10.000 rows=1000
loops=1)
               Hash Cond: (oi.order_id = o.order_id)
               Buffers: shared hit=90 read=50
               ->  Seq Scan on order_items oi  (cost=0.00..50.00 rows=5000 width=32) (actual
time=0.005..0.020 rows=5000 loops=1)
                     Buffers: shared hit=50
               ->  Hash  (cost=100.00..100.00 rows=1000 width=32) (actual time=2.500..2.500 rows=1000
loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 1500kB
                     Buffers: shared hit=40 read=0
                     ->  Seq Scan on orders o  (cost=0.00..100.00 rows=1000 width=32) (actual
time=0.003..0.004 rows=1000 loops=1)
                           Buffers: shared hit=40
               ->  Hash Motion  (cost=100.00..100.00 rows=1000 width=32) (actual time=2.500..2.500
rows=1000 loops=1)
                     Hash Key: c.customer_id
                     Buffers: shared hit=50
                     ->  Seq Scan on customers c  (cost=0.00..100.00 rows=1000 width=32) (actual
time=0.002..0.003 rows=1000 loops=1)
                           Buffers: shared hit=50
 Planning time: 0.120 ms
 Execution time: 11.235 ms
```

Analyzing the Sample Plan

Let's break down the output step by step:

- **Sort Operation**: -

Node Type: Sort indicates that the results are being sorted by the SUM(oi.quantity) in descending order. - **Memory Usage**: Memory: 1200kB shows how much memory is being consumed during the sort operation. - **Buffers**: It shows the buffers accessed, including shared hits and reads from disk.

- **Hash GroupAgg**: -

This node indicates a hash-based aggregation operation is being performed on the grouped results. - The Group Key reveals the keys by which the results are grouped (i.e., o.order_id and c.customer_name).

- **Hash Join**: -

Hash Condition: This node joins order_items and orders on order_id. Hash joins are typically efficient for larger datasets, especially when the hash table can fit in memory. -

Buffers: The Buffers indicate the shared buffers accessed during this join operation.

- **Sequential Scans**:

Seq Scan on order_items and Seq Scan on orders indicate that the entire tables are being scanned. This may suggest missing indices on these tables, which could be improved by adding appropriate indices.

- **Hash Motion**:

This node indicates that data is being redistributed based on the hash of customer_id, which is necessary for ensuring that all rows sharing the same key are processed together in a parallel execution context.

- **Memory Consumption**:

The overall query execution time is shown at the end, with specific details about planning time and execution time.

4. Understanding Row Estimates and Consumption

- **Estimating Rows**:

PostgreSQL uses statistics gathered by the ANALYZE command to estimate how many rows each node will process. If your estimates are significantly off, you might need to run ANALYZE on your tables to update statistics or adjust your query for better estimates.

- **Memory Consumption**:

Use the Buffers section in the query output to understand memory usage. The breakdown will typically show: -

Shared Buffers Hit: Number of buffers read from shared memory.

- **Buffers Read**: Number of buffers read from disk.

- **Temporary Buffers**: Memory allocated for intermediate results. This information helps you gauge if your query is consuming excessive memory or causing I/O bottlenecks.

- **Redirect Motion**:

In cases where a query involves a large number of rows, you may encounter "redirect motion," where the plan indicates that data is being moved between different nodes. This can be seen in operations like Gather or Gather Merge, which collect results from multiple parallel workers. Monitoring this can reveal inefficiencies in how data is processed and transmitted.

5. Common Bottlenecks

- **Missing or Inefficient Indices**: If your query relies on Seq Scan or Bitmap Heap Scan operations, a missing or inefficient index is likely the culprit.

- **Inefficient Join Methods**: Nested loop joins with large datasets can be very slow. Consider alternative join methods like Hash Join or Merge Join.

- **Large Sort and Group Operations**: Sorting or grouping large datasets can be expensive. Indices or materialized views can sometimes help.

- **Slow Disk I/O**: Excessive disk I/O can impact performance. Consider improving disk hardware or caching frequently accessed data.

6. Optimization Techniques.

Here are several optimization techniques to enhance query performance:

- **Add Indices**: Create appropriate indices for frequently accessed columns in WHERE, JOIN, and ORDER BY clauses.

```
CREATE INDEX idx_column_name ON your_table (column_name);
```

- **Improve Query Structure**: Break down complex queries into smaller, more manageable ones. This can help PostgreSQL optimize execution.

- **Tune Database Settings**: Optimize PostgreSQL parameters related to memory and caching in your postgresql.conffile to improve overall performance.

- **Use Materialized Views**: Materialized views can speed up retrieval of complex query results. Assess their effectiveness based on your specific use case — if the underlying data changes infrequently and you query the results often, they may provide a benefit. However, if your data is highly dynamic, the overhead may outweigh the advantages.

- **Analyze and Vacuum**: Regularly use the `VACUUM` and `ANALYZE` commands to maintain optimal performance, especially after significant data changes.

By understanding how to read and analyze query plans in PostgreSQL, you can identify performance bottlenecks and take actionable steps to optimize your queries. Regularly monitoring performance and applying these optimization techniques will help ensure your PostgreSQL database operates efficiently.