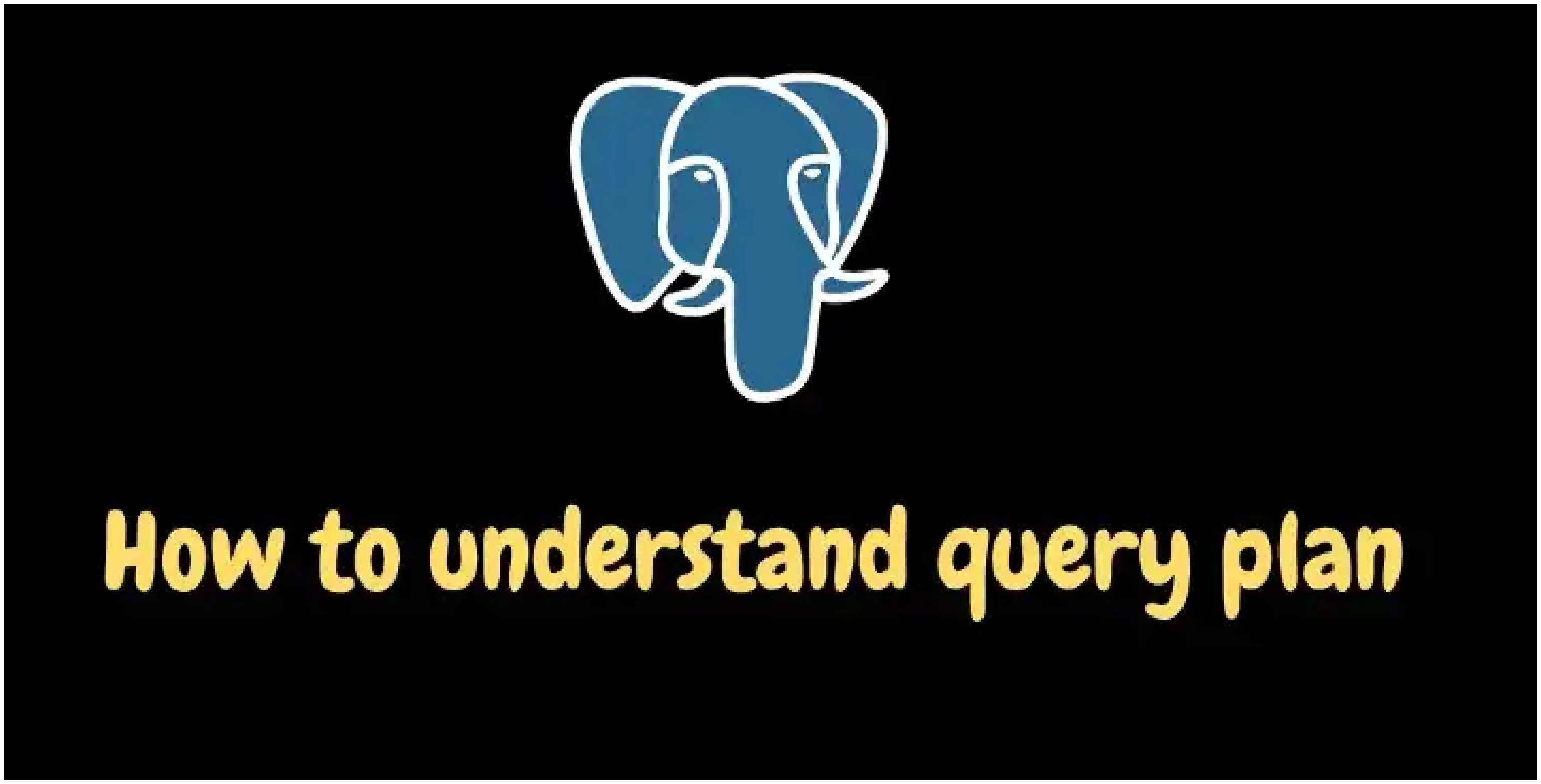
Published in Geek Culture

Minh Nguyen Follow
May 23, 2021 · 5 min read · Listen

Save

How to read PostgreSQL query plan



If you are a full stack or back end software engineer, there is a high chance that you have to work with database. Dealing with database performance is not only a job for DBA but also any engineers who work on the product. As a software engineer, most often my job is to write performant PostgreSQL queries to meet our API SLA. In order to achieve that, understanding PostgreSQL query plan is crucial.

In this blog post, I will show you my simple guide on how to read PostgreSQL query plan. If you prefer a tutorial video instead of reading, I have a video on this topic.

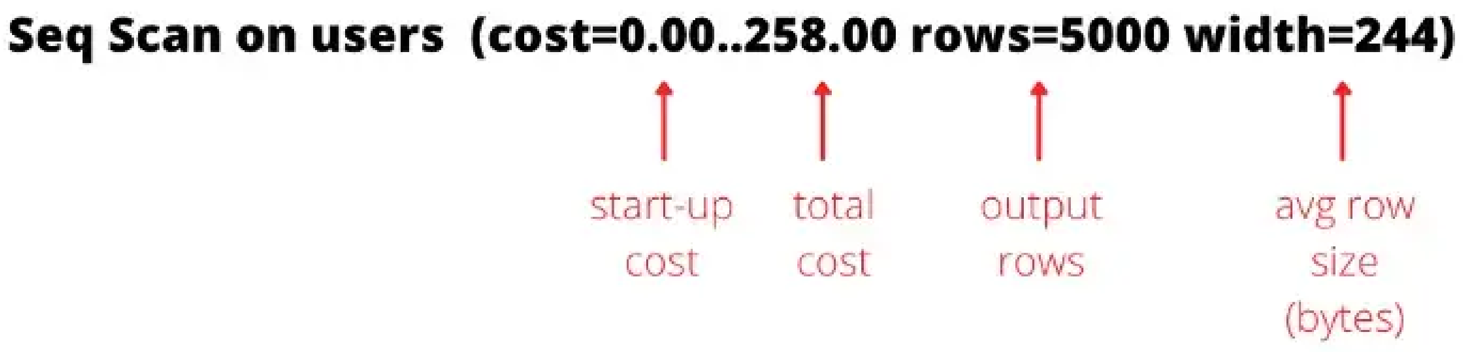
The EXPLAIN command

Postgres allows you to obtain the query plan by using the **EXPLAIN** command. For example:

```
EXPLAIN SELECT * FROM users;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on users  (cost=0.00..258.00 rows=5000 width=244)
```

It is important that **EXPLAIN** does not actually execute the query but rather gives an estimation which is in most of the cases, fairly close to the real statistics after query execution. Let's demystify the query plan above.



In the image above:

- **start-up cost** is the estimated Postgres computational units to start up a node to start process the query.
- **total cost** is the estimated Postgres computational units to finish process the query and return results.
- **output rows** is the estimated number of rows returned.
- **width or average row size (in bytes)** is the estimated size of each output row.

The **start-up cost** and **total cost** are separated by .. simple. From the definition above, we can say the SELECT query costs 258 computational units and returns 5000 rows with each weighs 244 bytes. The sequential scan means it iterates through all rows in the table. **Note:** these stats are all estimated.

The EXPLAIN ANALYZE command

If you want an accurate stats in query plan, **EXPLAIN ANALYZE** command is for you because it actually executes the query. I advise do not run this command in production database. Let's add **ANALYZE** in the previous query.

```
EXPLAIN ANALYZE SELECT * FROM users;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on users  (cost=0.00..258.00 rows=5000 width=244)(actual
time=0.000..7.149 rows=5000 loops=1)

Planning time: 0.004 ms
Execution time: 7.180 ms
```

As you can see, the plan shows planning and execution time. It also added **actual time** for the sequential scan with similar format as **cost: start-up time..total time**.

- **start-up time:** time taken to start up the node and start sequential scan.
- **total time:** time taken to finish the query and return output rows
- **rows**: number of rows returned.
- **loops**: how many seq scan was performed.

This query does 1 sequential scan that returns 5000 rows and takes around 7.2 ms. Simple right!!

The BUFFERS command

If you would like to know how much memory used for your query, BUFFERS will show you the stats. **Note:** in this example, I made up the buffer stats.

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM users;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on users  (cost=0.00..258.00 rows=5000 width=244)(actual
time=0.000..7.149 rows=5000 loops=1)
 Buffers: shared hit=300
 Total runtime: 7.180 ms
```

This line `Buffers: shared hit=300` indicates there are 300 cache blocks have been hit to support the sequential scan. Postgres has internal cache to speed up data retrieval. If there are no data in the cache, this would have been **shared read** which basically means reading data blocks from disk.

I think each block is from 3kb to 8kb (I do not have the exact number. This is just a rough estimation). In total, this query consume around 900 kb.

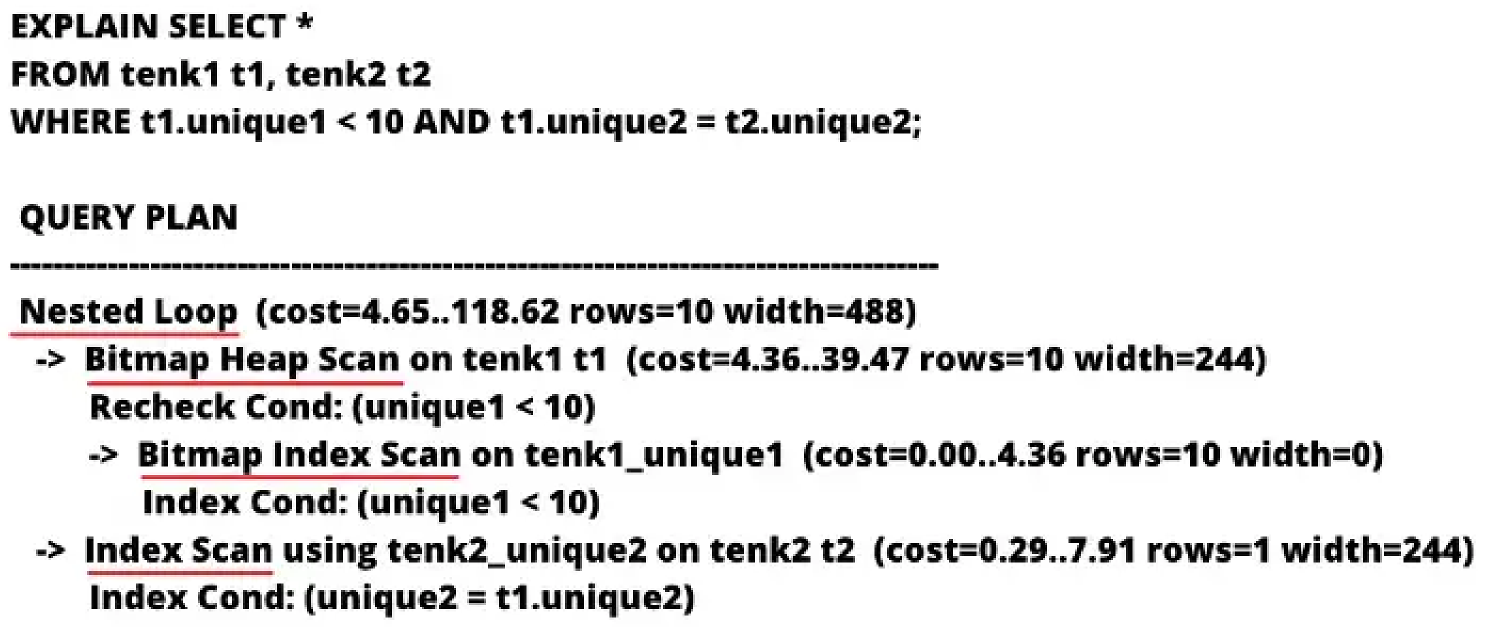
Query plan structure

So far we studied a very simple query. I have mentioned the term **node** in query plan above. But what does it mean? Let's look at a slightly more complicated query from Postgres doc to understand query plan structure.

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                      QUERY PLAN
------------------------------------------------------------------------------
---------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377
rows=10 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
(actual time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10
width=0) (actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1
width=244) (actual time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

Looks a bit scary right. Let's focus on the text with red line underneath.



```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

 QUERY PLAN
---------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   -> Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
        Recheck Cond: (unique1 < 10)
        -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
             Index Cond: (unique1 < 10)
   -> Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
        Index Cond: (unique2 = t1.unique2)
```

The text with red line underneath indicate the work is being carried out. Each work is handled by a **node** in Postgres. You can think of a **node** is almost like a node in a tree data structure. Each **node** can have child **nodes.** The easiest way to spot a **node** in query plan is where the arrows point to.

Postgres will process the lowest level **nodes** first. In this case, it is the **Bitmap Index Scan node** to check condition unique1 < 10 on index tenk1_unique1. This node returns 10 rows. Then, the results will be returned to its parent **node** (**Bitmap Heap Scan**) for further processing. Again, 10 rows are returned from this **node.**

At the same level as the **Bitmap Heap Scan node** is the **Index Scan node**. The **Index Scan** returns 1 row that satisfy condition unique2 = t1.unique2 using index tenk2_unique2.

After **Bitmap Heap Scan** and **Index Scan** complete, **Nested Loop** combines results from these 2 nodes and output results to the client.

I did not mention anything on cost, time and memory here because I would like you guys to read it yourself based on my instruction in previous sections.

I hope this blog post useful for the readers. If you like more contents on tech tutorial, follow me on Medium and Youtube.

12 | 1

Postgres   Postgresql   Query   Sql   Query Optimization