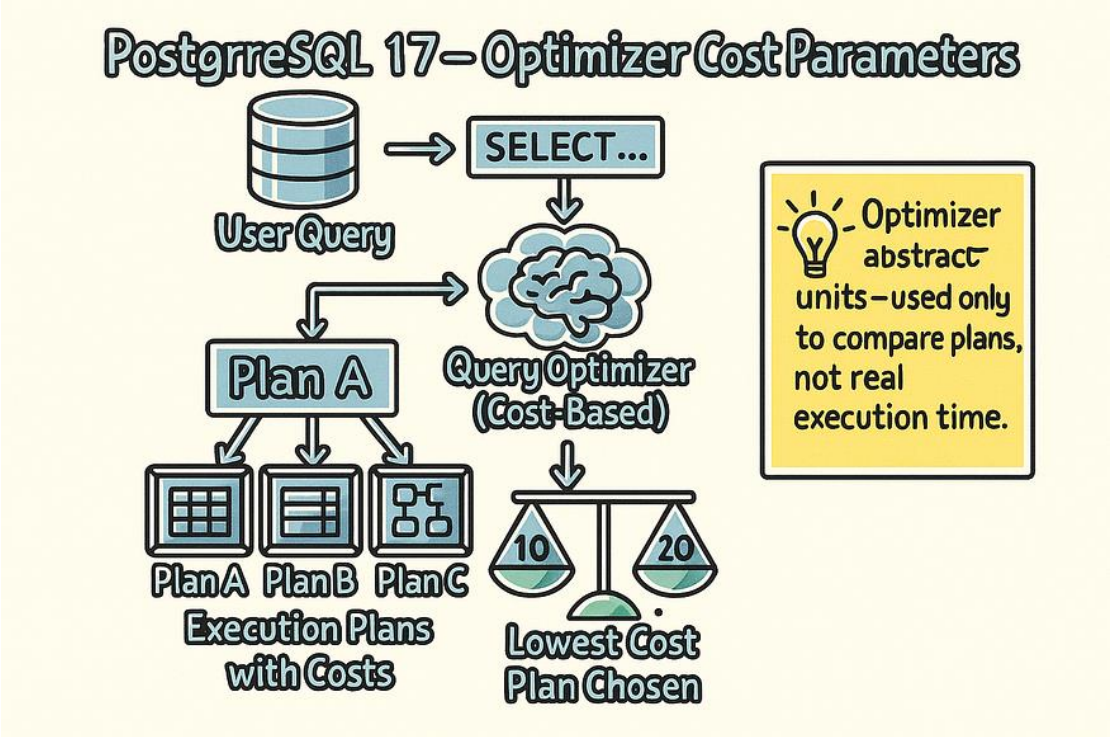
32 - PostgreSQL 17 Performance Tuning: Understanding Optimizer Cost Parameters



PostgreSQL is widely known as one of the most advanced open-source relational databases, and one of the main reasons for its power is its **cost-based query optimizer**.

When you execute a query, the optimizer doesn't just run it directly. Instead, it generates multiple possible execution plans and estimates their **costs**. The plan with the lowest estimated cost is chosen. These costs aren't measured in milliseconds or CPU cycles — they're **abstract units** that PostgreSQL uses for comparison.

In PostgreSQL 17, understanding these cost parameters is essential for performance tuning, especially when dealing with very large tables and complex queries.

In this post, we'll:

1. Create a table with **10 million rows** to simulate a real workload.
2. Build indexes to give PostgreSQL multiple execution options.
3. Break down PostgreSQL's **cost model** in detail.
4. Show how adjusting cost parameters can change query planning decisions.

Step 1: Create a Large Test Table

Let's create a table called `items` and populate it with 10 million rows.

```
CREATE TABLE items AS
SELECT g AS item_id,
       (random() * 1000)::int AS quantity,
       (random() * 500)::numeric(10,2) AS price
FROM generate_series(1, 10000000) g;
```

```
postgres=# CREATE TABLE items AS
postgres-# SELECT g AS item_id,
postgres-#        (random() * 1000)::int AS quantity,
postgres-#        (random() * 500)::numeric(10,2) AS price
postgres-# FROM generate_series(1, 10000000) g;
SELECT 10000000
postgres=#
```

```
-- Refresh statistics so the optimizer has accurate data
ANALYZE items;
```

```
postgres=# ANALYZE items;
ANALYZE
postgres=#
```

This creates:

- `item_id` → a sequential unique ID.
- `quantity` → random integer between 0 and 1000.
- `price` → random decimal between 0 and 500.

With 10 million rows, the optimizer has to think carefully about whether to use a **sequential scan** or an **index scan** for queries.

Step 2: Add Indexes

Indexes give PostgreSQL options. Let's add some:

```
-- Primary key on item_id
ALTER TABLE items ADD CONSTRAINT items pkey PRIMARY KEY (item id);



postgres=# ALTER TABLE items ADD CONSTRAINT items pkey PRIMARY KEY (item id);
ALTER TABLE
postgres=#



-- Index on quantity
CREATE INDEX idx_items_quantity ON items(quantity);
-- Index on price
CREATE INDEX idx_items_price ON items(price);



postgres=# CREATE INDEX idx_items_quantity ON items(quantity);
CREATE INDEX
postgres=#
postgres=# CREATE INDEX idx_items_price ON items(price);
CREATE INDEX
postgres=#
```

Now the optimizer can choose between:

- **Sequential Scan** → reading the whole table.
- **Index Scan** → reading specific rows using the index.

Which one it chooses depends on the **cost model**.

Step 3: PostgreSQL Cost Parameters

PostgreSQL calculates query plan costs using five main parameters:

1. **Sequential Page Cost (`seq_page_cost`)**

- Default: `1.0`
- Represents the cost of reading one page sequentially from disk.
- Example: Reading 100 pages costs about `100`.

2. **Random Page Cost (`random_page_cost`)**

- Default: `4.0`
- Represents the cost of reading one page randomly from disk.
- Historically higher because random reads on spinning disks were slower.
- On modern SSDs or RAM-heavy systems, this is often reduced to `2.0` or even `1.1`.

3. **CPU Tuple Cost (`cpu_tuple_cost`)**

- Default: `0.01`
- Cost to process a single row of data.

4. **CPU Index Tuple Cost (`cpu_index_tuple_cost`)**

- Default: `0.005`
- Cost to process a single index entry.
- Slightly cheaper than row processing because index entries are smaller.

5. **CPU Operator Cost (`cpu_operator_cost`)**

- Default: `0.0025`
- Cost to evaluate a single operator or function (e.g., comparisons, arithmetic).

Step 4: Breaking Down an Example

Let's run a query:

```
EXPLAIN
SELECT * FROM items WHERE quantity < 10;
```

Suppose the output is:

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM items WHERE quantity < 10;
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------
-----------------------------------------------------------
 Bitmap Heap Scan on items  (cost=765.93..57994.05 rows=88692 width=14) (actual
time=36.607..259.650 rows=94983 loops=1)
   Recheck Cond: (quantity < 10)
   Heap Blocks: exact=44911
   ->  Bitmap Index Scan on idx items quantity lt10  (cost=0.00..743.75 rows=88692
width=0) (actual time=19.548..19.549 rows=94983 loops=1)
 Planning Time: 0.128 ms
 Execution Time: 268.532 ms
(6 rows)

postgres=#
```

How does PostgreSQL compute this cost?

- **Page reads**: 11 pages × `seq_page_cost (1.0)` = `11`.
- **Row processing**: 500 rows × `cpu_tuple_cost (0.01)` = `5`.
- **Total cost**: `11 + 5 = 16`.

That's the cost shown in the plan.

This example demonstrates that cost isn't about real time — it's about combining different factors into a comparable score.

Step 5: Sequential vs. Index Scans

The optimizer's decision often comes down to sequential vs. index scans.

Case 1: High `random_page_cost` (default: 4.0)

If random access is "expensive," PostgreSQL will avoid index scans unless they return very few rows. For large tables, it will prefer a **sequential scan** even if an index exists.

Case 2: Lower `random_page_cost` (e.g., 2.0)

On SSDs or in-memory systems, random access is much cheaper. Lowering this cost makes PostgreSQL **more likely to use index scans**, even for moderately selective queries.

```
SET random page cost = 2.0;

EXPLAIN
SELECT * FROM items WHERE quantity < 10;



postgres=# SET random_page_cost = 2.0;
SET
postgres=#

postgres=# EXPLAIN ANALYZE
SELECT * FROM items WHERE price < 10;
                                                                        QUERY PLAN
-------------------------------------------------------------------------------------------
-------------------------------------------------------------
 Gather  (cost=7639.03..214256.96 rows=501238 width=62) (actual
time=85.107..4568.975 rows=500130.00 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=37 read=117970
   ->  Parallel Bitmap Heap Scan on items  (cost=6639.03..163133.16 rows=208849
width=62) (actual time=67.791..4446.307 rows=166710.00 loops=3)
         Recheck Cond: (price < '10'::numeric)
         Rows Removed by Index Recheck: 1764749
         Heap Blocks: exact=10163 lossy=13424
         Buffers: shared hit=37 read=117970
```

```
         Worker 0:  Heap Blocks: exact=19944 lossy=26166
         Worker 1:  Heap Blocks: exact=20272 lossy=26629
         ->  Bitmap Index Scan on idx_items_price  (cost=0.00..6513.72 rows=501238
width=0) (actual time=72.100..72.101 rows=500130.00 loops=1)
               Index Cond: (price < '10'::numeric)
               Index Searches: 1
               Buffers: shared read=1369
 Planning Time: 0.079 ms
 Execution Time: 4997.999 ms
(17 rows)

postgres=#
```

You may now see an **Index Scan** instead of a **Seq Scan**, because the planner estimates that index access is cheaper under the new assumptions.

Step 6: Real-World Tuning

In practice, DBAs often adjust these parameters:

- On **HDD-based systems**: Leave defaults or keep random page cost high (4.0).
- On **SSD/NVMe-based systems**: Lower `random_page_cost` to 2.0 or lower.
- On **RAM-heavy systems** (where most of the database fits in memory): Even more aggressive tuning is possible, making random reads nearly as cheap as sequential.

This tuning ensures the optimizer's cost model better reflects real-world performance, leading to smarter plan choices.

Step 7: The Building Blocks of Plans

Every query plan in PostgreSQL is built from a combination of these operations:

1. Sequential page read
2. Random page read

3. Row processing
4. Index entry processing
5. Operator execution

Complex query plans are nothing more than structured combinations of these five primitives. By understanding how each is costed, you can understand why PostgreSQL makes the choices it does.

Final Thoughts

PostgreSQL's query optimizer is powerful, but it relies on a **relative cost model**.

- Costs are abstract numbers, not milliseconds.
- The defaults are reasonable, but they reflect an era of spinning disks.
- On modern systems with SSDs and large RAM, tuning `random_page_cost` is one of the **most common optimizations**.

By experimenting with a large table like `items`, creating indexes, and running `EXPLAIN`, you can see exactly how PostgreSQL 17 evaluates your queries—and adjust parameters so the optimizer matches the reality of your hardware.