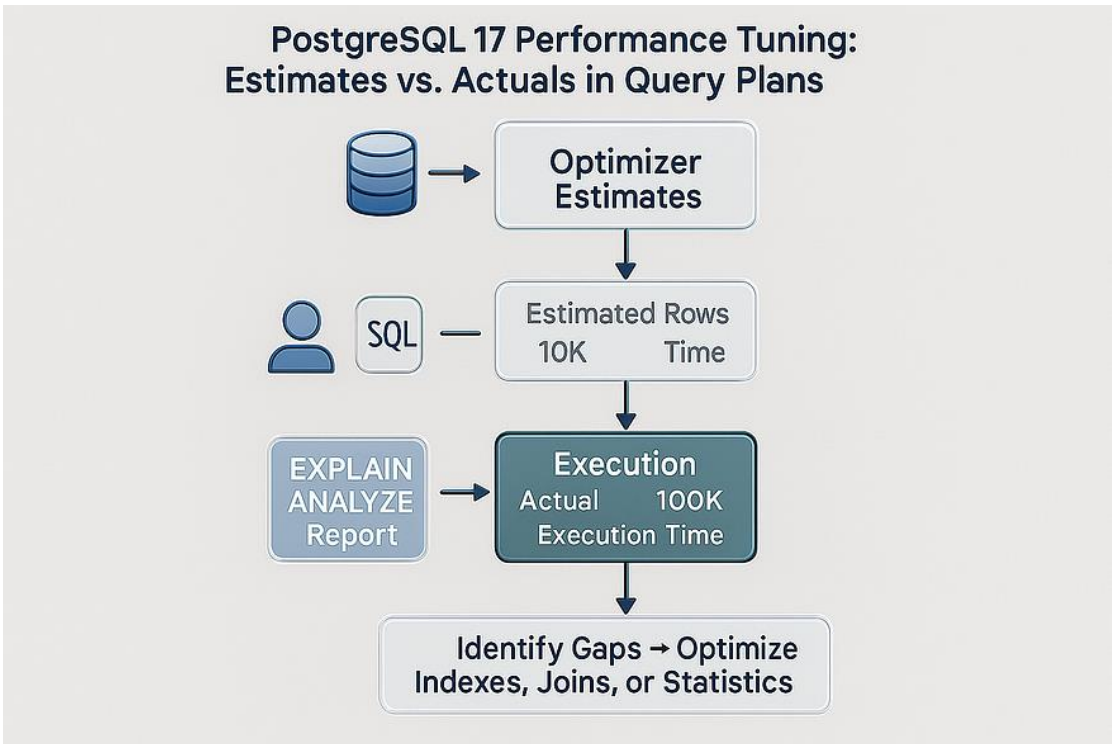
29 - PostgreSQL 17 Performance Tuning: Understanding Estimates vs. Actuals in Query Plans



Performance tuning in PostgreSQL often comes down to a single skill: **learning how to read query plans**. PostgreSQL's `EXPLAIN ANALYZE` command is the primary tool for this. It not only shows how the query was executed but also reveals **what PostgreSQL's optimizer expected versus what actually happened**.

When reviewing a query plan, there are always two big questions you should ask yourself:

1. **Are the times shown in the `EXPLAIN ANALYZE` output justified for this query?**
2. **Where does the runtime suddenly spike?**

Spotting these “jumps” in execution time often reveals the exact node or operation that is slowing down your query.

But there’s another key piece of advice: always compare the **optimizer’s estimates** with the **actual results**. Large differences here are a red flag that something is wrong.

Why Estimates vs. Actuals Matter

PostgreSQL’s query planner makes decisions based on **statistics**: how many rows it thinks will be returned, what data distributions look like, and how selective filters are. If its **estimates** don’t match reality, the chosen plan can be inefficient.

- **Underestimation**: PostgreSQL thinks only a few rows will match but millions actually do. It might choose a nested loop join, which works for small sets but collapses under large ones.
- **Overestimation**: PostgreSQL thinks millions of rows will be returned but only a few actually match. It might allocate unnecessary memory or pick an overly complex join method.

The root causes of poor estimates can include:

- Out-of-date statistics (fixed with `ANALYZE`).
- Lack of statistics on **expressions** or **computed values**.
- Highly skewed data distributions that fool the planner.

The autovacuum daemon in PostgreSQL usually handles statistics collection, but sometimes you need to take extra steps — especially when dealing with expressions.

Step 1: Creating a Large Test Table

To make this concrete, let's create a test table named `big_orders` and load it with **10 million rows**. This will simulate a realistic workload where optimizer estimates matter.

```
CREATE TABLE big_orders
(
  id           BIGSERIAL PRIMARY KEY,
  customer_id  INT NOT NULL,
  status       TEXT NOT NULL CHECK (status IN ('open','closed')),
  amount       NUMERIC(10,2) NOT NULL,
  created_at   DATE NOT NULL,
  note         TEXT
);



postgres=# CREATE TABLE big_orders
(
  id           BIGSERIAL PRIMARY KEY,
  customer_id  INT NOT NULL,
  status       TEXT NOT NULL CHECK (status IN ('open','closed')),
  amount       NUMERIC(10,2) NOT NULL,
  created_at   DATE NOT NULL,
  note         TEXT
);
CREATE TABLE
postgres=#



-- Insert 10 million rows with skewed data
INSERT INTO big_orders (customer_id, status, amount, created_at, note)
SELECT
  GREATEST(1, LEAST(100000, CEIL(100000 * power(random(), 2))))::int,
  CASE WHEN random() < 0.9 THEN 'open' ELSE 'closed' END,
  ROUND((exp(random()*log(1000+1)) - 1)::numeric, 2),
  CASE
    WHEN random() < 0.8 THEN (CURRENT_DATE - (random()*90)::int)
    ELSE (CURRENT_DATE - (90 + (random()*275)::int))
  END,
  repeat('x', 20 + (random()*20)::int)
FROM generate_series(1, 10000000);



postgres=# -- Insert 10 million rows with skewed data
INSERT INTO big_orders (customer_id, status, amount, created_at, note)
SELECT
  GREATEST(1, LEAST(100000, CEIL(100000 * power(random(), 2))))::int,
  CASE WHEN random() < 0.9 THEN 'open' ELSE 'closed' END,
  ROUND((exp(random()*log(1000+1)) - 1)::numeric, 2),
  CASE
    WHEN random() < 0.8 THEN (CURRENT_DATE - (random()*90)::int)
    ELSE (CURRENT_DATE - (90 + (random()*275)::int))
  END,
  repeat('x', 20 + (random()*20)::int)
FROM generate_series(1, 10000000);
INSERT 0 10000000
postgres=#



CREATE INDEX ON big_orders (customer_id);
CREATE INDEX ON big_orders (status);
CREATE INDEX ON big_orders (created_at);
-- Ensure statistics are collected
ANALYZE orders;



postgres=# CREATE INDEX ON big_orders (customer_id);
CREATE INDEX
postgres=# CREATE INDEX ON big_orders (created_at);
CREATE INDEX
postgres=# CREATE INDEX ON big_orders (status);
CREATE INDEX

postgres=# ANALYZE orders;
ANALYZE
postgres=#
```

This setup gives us:

- **Skew on `status`** → 90% `'open'`, 10% `'closed'`.
- **Skew on `customer_id`** → small IDs are very frequent.
- **Skew on `created_at`** → most rows in last 90 days.

Perfect ground for planner misestimates.

Step 2: Running a Simple Filter Query

Let's start with a basic filter:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT *
FROM big_orders
WHERE customer_id = 42
  AND status = 'open';



postgres=# ANALYZE orders;
ANALYZE
postgres=# EXPLAIN (ANALYZE, BUFFERS)
```

```
SELECT *
FROM big_orders
WHERE customer_id = 42
  AND status = 'open';
                                                                                   QUERY PLAN
-------------------------------------------------------------------------------------
------------------------------------------------------
 Bitmap Heap Scan on big_orders  (cost=35.61..10480.72 rows=2690 width=57) (actual
time=3.263..1133.014 rows=2207 loops=1)
   Recheck Cond: (customer_id = 42)
   Filter: (status = 'open'::text)
   Rows Removed by Filter: 236
   Heap Blocks: exact=2413
   Buffers: shared hit=219 read=2199
   I/O Timings: shared read=1084.008
   ->  Bitmap Index Scan on big_orders_customer_id_idx  (cost=0.00..34.94 rows=3000
width=0) (actual time=2.067..2.068 rows=2443 loops=1)
         Index Cond: (customer_id = 42)
         Buffers: shared read=5
         I/O Timings: shared read=1.719
 Planning:
   Buffers: shared hit=77 read=4
   I/O Timings: shared read=2.545
 Planning Time: 2.845 ms
 Execution Time: 1134.558 ms
(16 rows)

postgres=#
```

- If the **estimate is close to actual**, the plan is reliable.
- If the **estimate is way off** (e.g., planner thinks 100 rows, actual is 10,000), that's a red flag.

Step 3: Where Expressions Break Estimates

Things get more interesting with expressions. PostgreSQL collects statistics on **columns**, not on **expressions**. So if you write a query like this:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT *
FROM big_orders
WHERE status = 'open'
  AND created_at >= CURRENT_DATE - 30
  AND amount < 50;



postgres=# EXPLAIN (ANALYZE, BUFFERS)
SELECT *
FROM big_orders
WHERE status = 'open'
  AND created_at >= CURRENT_DATE - 30
  AND amount < 50;
```

```
                                                                      QUERY PLAN
-------------------------------------------------------------------------------------
------------------------------------------------------------------
 Bitmap Heap Scan on big_orders  (cost=29788.62..318698.27 rows=2402702 width=57)
(actual time=138.001..70388.494 rows=2439332 loops=1)
   Recheck Cond: (created_at >= (CURRENT_DATE - 30))
   Rows Removed by Index Recheck: 4283417
   Filter: ((amount < '50'::numeric) AND (status = 'open'::text))
   Rows Removed by Filter: 271332
   Heap Blocks: exact=45906 lossy=66163
   Buffers: shared hit=57 read=114299
   I/O Timings: shared read=65492.783
   ->  Bitmap Index Scan on big_orders_created_at_idx  (cost=0.00..29187.95
rows=2679401 width=0) (actual time=126.656..126.656 rows=2710664 loops=1)
         Index Cond: (created_at >= (CURRENT_DATE - 30))
         Buffers: shared hit=2 read=2285
         I/O Timings: shared read=19.294
 Planning:
   Buffers: shared hit=142 read=3
   I/O Timings: shared read=2.747
 Planning Time: 3.244 ms
 Execution Time: 71717.862 ms
(17 rows)

postgres=#
postgres=#
```

- **Why risky?**
  `status = 'open'` matches millions of rows. If PostgreSQL underestimates, it may use **nested loops** or repeated index lookups that blow up in runtime.

- **Expected misestimate:**
  Estimated rows ≪ Actual rows.

This mismatch can cause the query planner to choose the wrong join type, apply poor parallelization, or waste memory. And when this happens **deep inside a complex query plan**, performance can suffer dramatically.

Step 4: Indexes for Performance and Statistics

Indexes don't just speed up queries. They also improve PostgreSQL's **understanding of the data**. When you create an index:

- PostgreSQL tracks **detailed statistics** for the indexed column or expression.

- Even if the index isn't used in execution, those statistics inform the optimizer.
- This makes query plans more accurate and stable.

In our example, the expression index on `(quantity + 1)` not only enables possible index scans but also fixes the statistics problem.

Final Thoughts

PostgreSQL's query planner is powerful, but it relies heavily on accurate statistics.

When tuning performance:

- Always compare **estimated vs. actual row counts** in `EXPLAIN ANALYZE`.
- Look for sudden **runtime jumps** in execution nodes.
- Remember that PostgreSQL doesn't collect stats on expressions by default.
- Use **expression indexes** to improve estimates, even if the index itself isn't used.
- Refresh statistics with `ANALYZE` when necessary.

Ultimately, becoming an expert in PostgreSQL performance tuning means **practicing with query plans** until you recognize patterns and understand how PostgreSQL makes decisions. With PostgreSQL 17, tools like expression indexes make it easier than ever to align estimates with reality, leading to faster, more predictable queries.