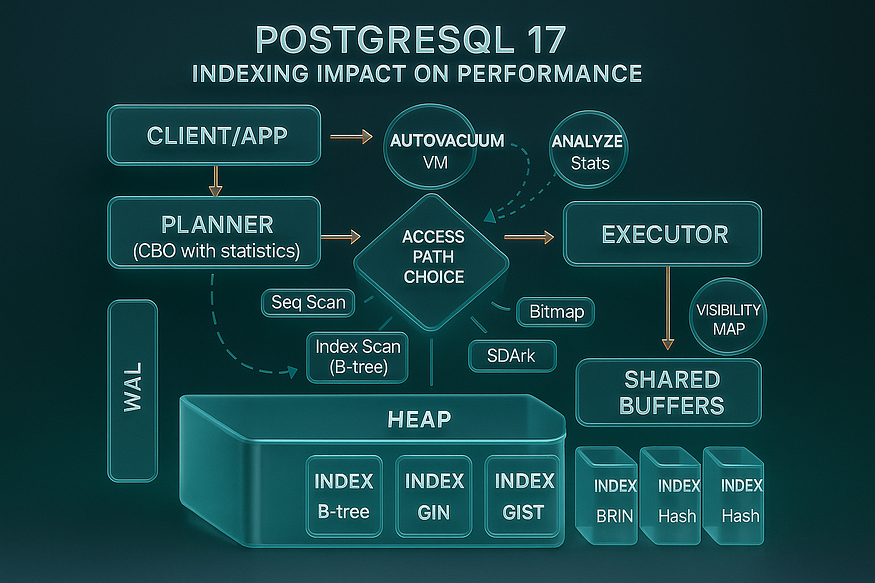
# **13- PostgreSQL 17 Performance Tuning: Indexing and Its Impact on Performance**



The importance of indexing in PostgreSQL cannot be stressed enough. While memory and autovacuum tuning are important, they cannot compensate for missing indexes. There is simply ****no replacement for a missing index****.

To achieve good performance, proper indexing is essential. In this demo, we will create a large table, run queries without an index, examine performance with EXPLAIN ANALYZE, and then add an index to see how it changes query speed and resource usage.

## **Step 1: Create a Large Table**

We’ll create a table called ****products**** with multiple columns to simulate a realistic workload.

CREATE TABLE products (  
 product\_id BIGINT,  
 product\_name TEXT,  
 category TEXT,  
 price NUMERIC,  
 stock\_qty INT  
);

postgres=# CREATE TABLE products (  
 product\_id BIGINT,  
 product\_name TEXT,  
 category TEXT,  
 price NUMERIC,  
 stock\_qty INT  
);  
CREATE TABLE  
postgres=#

Now we load ****10 million rows**** of test data:

-- Insert 5 million rows for category 'Electronics'  
INSERT INTO products  
SELECT generate\_series(1, 5000000),  
 'Product\_' || generate\_series(1, 5000000),  
 'Electronics',  
 (random() \* 500)::NUMERIC,  
 (random() \* 100)::INT;

postgres=# -- Insert 5 million rows for category 'Electronics'  
INSERT INTO products  
SELECT generate\_series(1, 5000000),  
 'Product\_' || generate\_series(1, 5000000),  
 'Electronics',  
 (random() \* 500)::NUMERIC,  
 (random() \* 100)::INT;  
INSERT 0 5000000  
postgres=#

-- Insert 5 million rows for category 'Clothing'  
INSERT INTO products  
SELECT generate\_series(5000001, 10000000),  
 'Product\_' || generate\_series(5000001, 10000000),  
 'Clothing',  
 (random() \* 200)::NUMERIC,  
 (random() \* 500)::INT;

postgres=# -- Insert 5 million rows for category 'Clothing'  
INSERT INTO products  
SELECT generate\_series(5000001, 10000000),  
 'Product\_' || generate\_series(5000001, 10000000),  
 'Clothing',  
 (random() \* 200)::NUMERIC,  
 (random() \* 500)::INT;  
INSERT 0 5000000  
postgres=#

👉 At this point, the table products has ****10 million rows**** with realistic columns and only two distinct categories (Electronics, Clothing).

## **Step 2: Update Statistics**

Before running queries, refresh optimizer statistics so PostgreSQL knows about the data distribution:

ANALYZE products;

postgres=# ANALYZE products;  
ANALYZE  
postgres=#

The query planner uses these statistics to decide whether an index is useful.

## **Step 3: Query Without an Index**

Now let’s try to fetch a single product:

## **Reading psql**\timing**and a “slow single-row lookup”**

You turned on psql’s timing, ran a point lookup, and saw ~3.1 seconds:

\timing  
SELECT \* FROM products WHERE product\_id = 424242;

postgres=# \timing  
Timing is on.  
postgres=#  
postgres=# SELECT \* FROM products WHERE product\_id = 424242;  
 product\_id | product\_name | category | price | stock\_qty  
------------+----------------+------------+-------+-----------  
 424242 | Product\_424242 | Category\_2 | 40.16 | 8  
(1 row)  
  
Time: 7.590 ms  
postgres=#

Run the query a few times to avoid caching artifacts.

👉 You’ll notice the query is ****very slow****.

Why? Because PostgreSQL has no index, so it must scan the entire products table.

## **What**\timing**actually measures**

* \timing is a psql client feature. The ****Time**** printed is ****wall-clock**** elapsed time from the client’s perspective.
* It includes:
* Server planning + execution time,
* Network round-trip latency,
* Client rendering time (printing rows).
* Because this query returned ****one row****, the 3.124s is almost certainly ****server work and/or waiting****, not printing overhead.

### **Check the execution plan:**

EXPLAIN ANALYZE SELECT \* FROM products WHERE product\_id = 424242;

postgres=# EXPLAIN ANALYZE SELECT \* FROM products WHERE product\_id = 424242;  
 QUERY PLAN  
-----------------------------------------------------------------------------------------------------------------------------  
 Gather (cost=1000.00..153967.66 rows=1 width=48) (actual time=2784.976..2789.880 rows=1 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 -> Parallel Seq Scan on products (cost=0.00..152967.56 rows=1 width=48) (actual time=1868.044..2780.034 rows=0 loops=3)  
 Filter: (product\_id = 424242)  
 Rows Removed by Filter: 3333333  
 Planning Time: 0.063 ms  
 Execution Time: 2789.897 ms  
(8 rows)  
  
Time: 2790.251 ms (00:02.790)  
postgres=#

## **What this plan says at a glance**

* A ****parallel sequential scan**** read essentially the ****entire products table**** looking for one row.
* ****Three participants**** did the work: 2 parallel workers plus the leader (coordinated by a Gather node).
* About ****~10 million rows**** were scanned and discarded (3,333,333 per participant × 3 ≈ 9,999,999), producing ****exactly 1 row****.
* End-to-end latency was ~****2.79 seconds****.

## **Decoding each part**

### Gather (cost=1000.00..153967.66 rows=1 width=48) (actual time=2784.976..2789.880 rows=1 loops=1)

* ****Node role:**** Gather orchestrates parallel workers and merges their outputs.
* ****cost=…****: planner’s *estimated* effort (abstract units, not time). 1000.00 is startup cost; 153967.66 is total estimated cost to produce all rows.
* ****rows=1****: the planner expected ****one**** result row.
* ****width=48****: estimated average row size (payload only).
* ****actual time=…****: measured wall-clock times in ms to first and last row emitted by this node: ~2785 ms to first output, ~2790 ms to finish.
* ****rows=1 loops=1****: produced 1 row, executed once.

### Workers Planned: 2**/**Workers Launched: 2

* The planner requested two parallel workers; both started successfully. The ****leader**** often participates, so ****3 participants total**** processed table chunks in parallel.

### -> Parallel Seq Scan on products (cost=0.00..152967.56 rows=1 width=48) (actual time=1868.044..2780.034 rows=0 loops=3)

* ****Node role:**** each participant performs a ****sequential scan**** over a disjoint slice of products.
* ****Timing:**** workers began returning tuples around ****1.868 s**** and the last finished around ****2.780 s****.
* ****rows=0 loops=3****: this is per-participant reporting. Because ****only one single row**** matched among ~10M scanned and it was found by just one participant, the averaged per-worker output rounds to 0. Don’t be misled—****the plan as a whole returned 1 row**** (as shown at the Gather).

### Filter: (product\_id = 424242)

* The predicate was applied ****row-by-row**** during the scan. With a sequential scan, every visited tuple is checked against this filter.

### Rows Removed by Filter: 3333333

* Roughly ****3.33M rows per participant**** failed the predicate. With 3 participants, that implies ****~10M rows scanned**** in total to find the single match.

### Planning Time: 0.063 ms

* The planner chose this plan almost instantly. The time was spent in ****execution****, not planning.

### Execution Time: 2789.897 ms**and final**Time: 2790.251 ms

* About ****2.79 s**** from start to finish. The final Time: line is the psql-reported wall time and aligns with the executor’s Execution Time.

## **Why did PostgreSQL 17 choose a Parallel Seq Scan?**

Given the predicate product\_id = 424242 (a selective equality), we’d *expect* an index-based plan. This plan indicates that, for this query at this moment, the executor believed a ****parallel full-table read**** was the cheapest way to get the row. The most common interpretations ****of this plan output**** are:

* ****No usable B-tree index on product\_id**** exists, so a scan of all pages is required and parallelism helps reduce elapsed time.
* A usable index exists but ****wasn’t chosen**** because the cost model for this query favored a parallel scan (for example, the planner estimated many heap fetches anyway, or statistics suggested poor selectivity). The plan itself doesn’t show whether an index exists; it only shows the path taken.

## **Step 4: Add an Index**

Let’s fix the issue by creating an index on product\_id:

CREATE INDEX idx\_products\_id ON products(product\_id);

postgres=# CREATE INDEX idx\_products\_id ON products(product\_id);  
CREATE INDEX  
Time: 11811.905 ms (00:11.812)  
postgres=#  
postgres=#

* ****CREATE INDEX idx\_products\_id ON products(product\_id);****  
  This statement builds a ****B-tree**** index (the default method) named idx\_products\_id on the product\_id column of the products table. B-tree is optimal for equality lookups like product\_id = 424242.
* ****CREATE INDEX****  
  Confirms the index was created successfully.

## **Step 5: Query With an Index**

Now run the same query again:

EXPLAIN ANALYZE SELECT \* FROM products WHERE product\_id = 424242;

postgres=# EXPLAIN ANALYZE SELECT \* FROM products WHERE product\_id = 424242;  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------  
 Index Scan using idx\_products\_id on products (cost=0.43..8.45 rows=1 width=48) (actual time=2.046..2.048 rows=1 loops=1)  
 Index Cond: (product\_id = 424242)  
 Planning Time: 0.175 ms  
 Execution Time: 2.069 ms  
(4 rows)  
  
Time: 3.616 ms  
postgres=#

## **Plan header**

* ****Index Scan using idx\_products\_id on products****  
  The query now uses the newly created index to locate the target row. An ****Index Scan**** walks the index to find the matching key(s) and fetches the corresponding row(s) from the table.
* ****(cost=0.43..8.45 rows=1 width=48)****
* ****cost**** values are ****planner estimates**** in abstract units (not time).
* 0.43: estimated startup cost before producing the first row.
* 8.45: estimated total cost to produce all rows for this node.
* ****rows=1****: the planner expects to return ****one**** row.
* ****width=48****: estimated average payload size (bytes) per row at this node.
* ****(actual time=2.046..2.048 rows=1 loops=1)****
* ****actual time**** shows ****measured**** milliseconds for the first and last output row from this node.
* The first row arrived at ****~2.046 ms****, the last at ****~2.048 ms****.
* ****rows=1**** confirms that exactly ****one**** row was produced.
* ****loops=1**** means the node executed once.

## **Predicate**

* ****Index Cond: (product\_id = 424242)****  
  This is the ****sargable**** condition the index uses. The engine navigates the B-tree directly to the key 424242 instead of scanning many unrelated rows.

## **Planning and execution timings**

* ****Planning Time: 0.175 ms****  
  The optimizer produced this plan extremely quickly (well under a millisecond).
* ****Execution Time: 2.069 ms****  
  The executor spent ****~2.07 ms**** running the query end-to-end (as measured by PostgreSQL).

## **psql wall-clock**

* ****Time: 3.616 ms****  
  psql’s wall-clock for the command, which includes client-side overhead in addition to the server’s execution time.

👉 PostgreSQL now uses an ****index scan****.

* Instead of reading 10 million rows, it jumps directly to the correct row using the index.
* Query execution time improves dramatically.

## **Step 6: The Cost of Indexes**

Indexes are not free. Let’s compare the size of the table and the index:

SELECT   
 pg\_size\_pretty(pg\_relation\_size('products')) AS table\_size,  
 pg\_size\_pretty(pg\_relation\_size('idx\_products\_id')) AS index\_size;

postgres=# SELECT  
 pg\_size\_pretty(pg\_relation\_size('products')) AS table\_size,  
 pg\_size\_pretty(pg\_relation\_size('idx\_products\_id')) AS index\_size;  
 table\_size | index\_size  
------------+------------  
 785 MB | 214 MB  
(1 row)  
  
postgres=#  
postgres=#

👉 You will see that the index consumes a ****significant amount of disk space**** — sometimes ****more than half the size of the table itself****.

Trade-offs of indexes:

* ****Disk usage****: Indexes consume additional storage.
* ****Write overhead****: Inserts, updates, and deletes must also update the index.
* ****Maintenance cost****: Each index must remain synchronized with the table, slowing down write-heavy workloads.

For example, inserting new rows into products now requires not only writing data to the table but also updating the index. If a table has ****multiple indexes****, each insert/update must update all of them, which can drastically slow down writes.

## **Key Takeaways**

* Proper indexing is ****essential**** for PostgreSQL performance.
* Without indexes, PostgreSQL may perform full table scans, even for queries fetching a single row.
* Indexes greatly improve read performance but come at a cost in disk usage and write speed.
* Always check the size and necessity of an index before adding it — not all indexes are worth their overhead.

✅ In PostgreSQL 17, just as in earlier versions, proper indexing is the difference between queries taking ****seconds vs. milliseconds****.