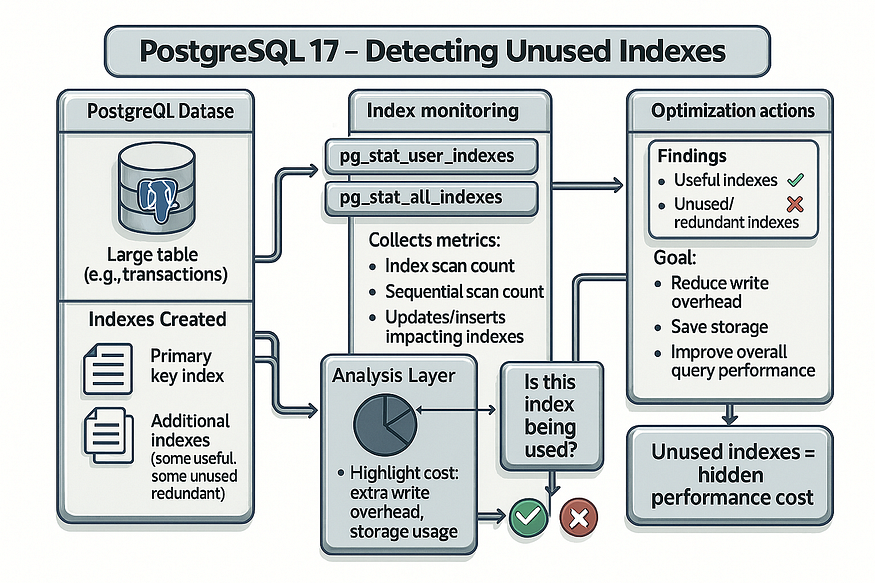
# **28 - PostgreSQL 17 Performance Tuning: Creating Tables, Populating 10M Records, and Detecting Unused Indexes**



When tuning PostgreSQL, most developers and DBAs focus on finding ****missing indexes**** to accelerate queries. That’s an important step, but there’s another side to the story: sometimes you need to identify ****indexes that should not exist at all****.

Unused or redundant indexes are often overlooked, yet they can silently degrade system performance. In this article, we’ll explore why this happens, and walk through a practical example of creating a large dataset, building indexes, and then analyzing which ones are actually useful.

## **Why Unused Indexes Can Hurt Performance**

Indexes exist to make queries faster, but they come with trade-offs. While they improve read performance, they add extra work to every write operation. Here’s why:

* ****Disk Space Waste**** — Every index consumes physical storage. If your database holds tens or hundreds of gigabytes of unused indexes, you’re wasting space that could be used for valuable data.
* ****Slower Write Operations**** — Each INSERT, UPDATE, or DELETE must also update every index on the table. The more indexes you have, the more overhead is added to each transaction.
* ****Maintenance Overhead**** — Background tasks like ****VACUUM****, ****autovacuum****, and ****analyze**** have to maintain every index. This makes them slower and consumes additional CPU and I/O.

In short: ****an unused index is not free****. In large production systems, a few unused indexes can be costly, while dozens or hundreds can seriously harm performance.

## **Step 1: Creating a Test Table**

Let’s simulate a real-world scenario with a product catalog table.

CREATE TABLE products (  
 product\_id BIGINT GENERATED BY DEFAULT AS IDENTITY,  
 product\_name TEXT NOT NULL,  
 category TEXT NOT NULL,  
 price NUMERIC(10,2) NOT NULL,  
 stock\_qty INT NOT NULL,  
 status TEXT NOT NULL,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);

postgres=# CREATE TABLE products (  
postgres(# product\_id BIGINT GENERATED BY DEFAULT AS IDENTITY,  
postgres(# product\_name TEXT NOT NULL,  
postgres(# category TEXT NOT NULL,  
postgres(# price NUMERIC(10,2) NOT NULL,  
postgres(# stock\_qty INT NOT NULL,  
postgres(# status TEXT NOT NULL,  
postgres(# created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
postgres(# updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
postgres(# );  
CREATE TABLE  
postgres=#

This table includes common fields you’d find in an e-commerce or inventory management system: product names, categories, prices, stock levels, and status flags.

## **Step 2: Inserting 10 Million Rows**

To properly test index usage, we need a large dataset. Using PostgreSQL’s generate\_series() and random functions, we can quickly populate 10 million rows:

INSERT INTO products (product\_name, category, price, stock\_qty, status, created\_at, updated\_at)  
SELECT  
 'Product\_' || gs::text,  
 'Category\_' || (1 + (gs % 20))::text,  
 round((random()\*999 + 1)::numeric, 2),  
 (random()\*1000)::int,  
 CASE  
 WHEN random() < 0.8 THEN 'open'  
 WHEN random() < 0.15 THEN 'archived'  
 ELSE 'backorder'  
 END,  
 now() - ((random()\*365)::int || ' days')::interval,  
 now()  
FROM generate\_series(1, 10000000) AS gs;

postgres=# INSERT INTO products (product\_name, category, price, stock\_qty, status, created\_at, updated\_at)  
SELECT  
 'Product\_' || gs::text,  
 'Category\_' || (1 + (gs % 20))::text,  
 round((random()\*999 + 1)::numeric, 2),  
 (random()\*1000)::int,  
 CASE  
 WHEN random() < 0.8 THEN 'open'  
 WHEN random() < 0.15 THEN 'archived'  
 ELSE 'backorder'  
 END,  
 now() - ((random()\*365)::int || ' days')::interval,  
 now()  
FROM generate\_series(1, 10000000) AS gs;  
  
INSERT 0 10000000  
postgres=#

This script generates:

* ****20 categories**** (Category\_1 … Category\_20)
* Prices between ****$1.00 and $999.00****
* Random stock levels from ****0 to 1000****
* Status values, 80% open, 15% archived, 5% backorder
* Creation dates spread across the past year

By the end, we have a ****realistic 10M row table**** — large enough to illustrate the performance impact of indexes.

## **Step 3: Creating Indexes**

Next, we’ll create several indexes. Some of these will be useful for queries, while others may rarely get used.

-- Primary key  
ALTER TABLE products ADD CONSTRAINT products\_pkey PRIMARY KEY (product\_id);

-- Index on status  
CREATE INDEX idx\_products\_status ON products (status);  
  
-- Partial index for "open" status only  
CREATE INDEX idx\_products\_status\_open ON products (status)  
WHERE status = 'open';  
  
-- Multi-column index on category and price  
CREATE INDEX idx\_products\_category\_price ON products (category, price);  
  
-- Index on stock quantity (may or may not be useful)  
CREATE INDEX idx\_products\_stock ON products (stock\_qty);  
  
-- Expression index for case-insensitive searches  
CREATE INDEX idx\_products\_lower\_name ON products (lower(product\_name));  
  
-- Index on creation timestamp for time-range queries  
CREATE INDEX idx\_products\_created\_at ON products (created\_at);

postgres=# ALTER TABLE products ADD CONSTRAINT products\_pkey PRIMARY KEY (product\_id);  
ALTER TABLE  
postgres=#  
postgres=# CREATE INDEX idx\_products\_status ON products (status);  
CREATE INDEX  
postgres=# CREATE INDEX idx\_products\_status\_open ON products (status)  
postgres-# WHERE status = 'open';  
CREATE INDEX  
postgres=#  
postgres=# CREATE INDEX idx\_products\_category\_price ON products (category, price);  
CREATE INDEX  
postgres=#  
postgres=# CREATE INDEX idx\_products\_stock ON products (stock\_qty);  
CREATE INDEX  
postgres=# CREATE INDEX idx\_products\_lower\_name ON products (lower(product\_name));  
CREATE INDEX  
postgres=# CREATE INDEX idx\_products\_created\_at ON products (created\_at);  
CREATE INDEX  
postgres=#

Now, our table has ****seven indexes**** in addition to the primary key. Some of these will be heavily used, others may end up as dead weight.

## **Step 4: Detecting use Indexes**

PostgreSQL tracks index usage statistics in the pg\_stat\_all\_indexes system view. To identify which indexes are actively used, you can filter for those with non-zero scan counts (idx\_scan > 0). This helps distinguish valuable indexes from those that may be candidates for review or removal.

Use the following query to list only the indexes that have been used at least once:

SELECT  
 s.schemaname,  
 c.relname AS table\_name,  
 s.indexrelname AS index\_name,  
 s.idx\_scan,  
 pg\_size\_pretty(pg\_relation\_size(s.indexrelid)) AS index\_size  
FROM  
 pg\_stat\_all\_indexes s  
JOIN  
 pg\_class c ON c.oid = s.indexrelid  
WHERE  
 s.schemaname NOT IN ('pg\_catalog', 'information\_schema')  
 AND s.idx\_scan > 0 -- ✅ Only include indexes that have been used  
ORDER BY  
 s.idx\_scan ASC,  
 pg\_relation\_size(s.indexrelid) DESC;

postgres=# SELECT  
postgres-# s.schemaname,  
postgres-# c.relname AS table\_name,  
postgres-# s.indexrelname AS index\_name,  
postgres-# s.idx\_scan,  
postgres-# pg\_size\_pretty(pg\_relation\_size(s.indexrelid)) AS index\_size  
postgres-# FROM  
postgres-# pg\_stat\_all\_indexes s  
postgres-# JOIN  
postgres-# pg\_class c ON c.oid = s.indexrelid  
postgres-# WHERE  
postgres-# s.schemaname NOT IN ('pg\_catalog', 'information\_schema')  
postgres-# AND s.idx\_scan > 0 -- ✅ Only include indexes that have been used  
postgres-# ORDER BY  
postgres-# s.idx\_scan ASC,  
postgres-# pg\_relation\_size(s.indexrelid) DESC;  
 schemaname | table\_name | index\_name | idx\_scan | index\_size  
------------+-----------------------------+-----------------------------+----------+------------  
 public | idx\_products\_status | idx\_products\_status | 1 | 66 MB  
 pg\_toast | pg\_toast\_2618\_index | pg\_toast\_2618\_index | 1 | 16 kB  
 public | idx\_products\_category\_price | idx\_products\_category\_price | 5 | 387 MB  
(3 rows)  
  
postgres=#

## **Step 4: Detecting Unused Indexes**

PostgreSQL provides index usage statistics in pg\_stat\_all\_indexes. With a simple query, you can see how often an index has been used and how much space it consumes:

SELECT  
 s.schemaname,  
 c.relname AS table\_name,  
 s.indexrelname AS index\_name,  
 s.idx\_scan,  
 pg\_size\_pretty(pg\_relation\_size(s.indexrelid)) AS index\_size  
FROM  
 pg\_stat\_all\_indexes s  
JOIN  
 pg\_class c ON c.oid = s.indexrelid  
WHERE  
 s.schemaname NOT IN ('pg\_catalog', 'information\_schema')  
ORDER BY  
 s.idx\_scan ASC,  
 pg\_relation\_size(s.indexrelid) DESC;

postgres=# SELECT  
postgres-# s.schemaname,  
postgres-# c.relname AS table\_name,  
postgres-# s.indexrelname AS index\_name,  
postgres-# s.idx\_scan,  
postgres-# pg\_size\_pretty(pg\_relation\_size(s.indexrelid)) AS index\_size  
postgres-# FROM  
postgres-# pg\_stat\_all\_indexes s  
postgres-# JOIN  
postgres-# pg\_class c ON c.oid = s.indexrelid  
postgres-# WHERE  
postgres-# s.schemaname NOT IN ('pg\_catalog', 'information\_schema')  
postgres-# ORDER BY  
postgres-# s.idx\_scan ASC,  
postgres-# pg\_relation\_size(s.indexrelid) DESC;  
 schemaname | table\_name | index\_name | idx\_scan | index\_size  
------------+----------------------+----------------------+----------+------------  
 pg\_toast | pg\_toast\_2618\_index | pg\_toast\_2618\_index | 0 | 16 kB  
 pg\_toast | pg\_toast\_2619\_index | pg\_toast\_2619\_index | 0 | 16 kB  
 pg\_toast | pg\_toast\_1255\_index | pg\_toast\_1255\_index | 0 | 16 kB

## **What the Columns Mean:**

* ****idx\_scan**** → number of times the index was scanned since the last stats reset
* ****index\_size**** → how much disk space the index uses

If idx\_scan = 0 and the index is several gigabytes in size, it’s a candidate for review and possibly removal.

## **Step 5: Why Manual Review Still Matters**

Dropping indexes isn’t always straightforward. There are edge cases:

* Some indexes are rarely used but ****critical for month-end reporting****.
* Unique indexes enforce constraints, even if not scanned.
* Certain application features might rely on an index only under specific conditions.

That’s why it’s wise to:

* Perform ****manual checking**** before dropping.
* Use monitoring tools like ****pgBadger****, ****pg\_stat\_statements****, or enterprise observability dashboards.
* Reset statistics (SELECT pg\_stat\_reset();) in a staging environment to measure fresh usage.

## **Step 6: Best Practices for Index Monitoring**

1. ****Regularly check index statistics**** — Run the pg\_stat\_all\_indexes query during maintenance cycles.
2. ****Automate monitoring**** — Integrate with tools that track index usage over time.
3. ****Avoid over-indexing**** — Create indexes only for queries that genuinely benefit from them.
4. ****Benchmark before dropping**** — Test query performance with and without the index.

## **Final Thoughts**

Unused indexes are a ****hidden performance tax****. They don’t throw errors or show up in query plans, but they quietly consume storage, slow down writes, and increase maintenance overhead.

By creating a table, populating it with millions of rows, and monitoring index usage in PostgreSQL 17, you can:

* Identify indexes that truly add value.
* Spot those that waste space and hurt performance.
* Keep your database lean, efficient, and scalable.

Monitoring activity and statistics lookups isn’t just a nice-to-have — it’s a ****must-have for serious performance tuning****.