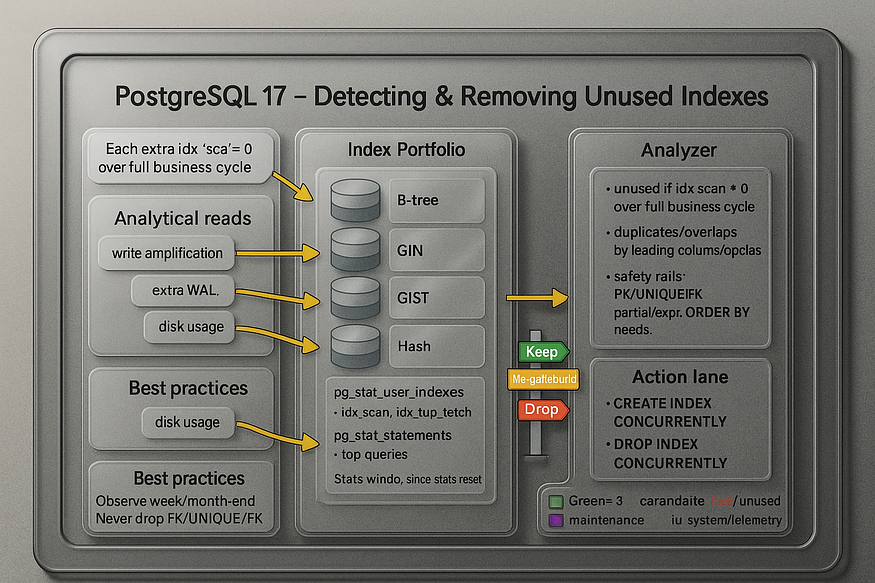
# **25 - PostgreSQL 17 Performance Tuning: Detecting and Removing Unused Indexes**



Indexes are one of the most important tools for query performance in PostgreSQL. But while a missing index slows down queries, an ****unused or pointless index**** can be just as harmful.

If your database is very small, a few extra indexes don’t matter much. But in a large production environment, ****unused indexes can waste hundreds of gigabytes and put unnecessary strain on the system****. Every extra index comes with hidden costs: more disk usage, slower writes, and more work for background processes like VACUUM.

That’s why tuning isn’t just about ****adding indexes****. It’s equally about ****finding indexes that shouldn’t exist****.

## **Why Unused Indexes Are a Problem**

1. ****Disk Space Waste****

* An index duplicates column values plus pointers to rows.
* On a 10M row table, each index can take hundreds of MB or even several GB.
* Multiple unused indexes = huge wasted storage.

2. ****Slower Write Performance****

* Every time you INSERT, UPDATE, or DELETE, PostgreSQL must also update all related indexes.
* More indexes = more overhead = slower write-heavy workloads.

3. ****Maintenance Overhead****

* VACUUM, REINDEX, and CLUSTER must also handle these indexes.
* Index bloat and dead entries pile up even if the index is never queried.

👉 In short: while indexes make reads faster, unused indexes ****slow everything else down****.

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

Let’s simulate a real-world dataset with 10 million rows:

CREATE TABLE products (  
 product\_id BIGSERIAL PRIMARY KEY,  
 product\_name TEXT,  
 category TEXT,  
 price NUMERIC(10,2),  
 stock\_qty INT  
);

postgres=# CREATE TABLE products (  
 product\_id BIGSERIAL PRIMARY KEY,  
 product\_name TEXT,  
 category TEXT,  
 price NUMERIC(10,2),  
 stock\_qty INT  
);  
CREATE TABLE  
postgres=#

-- Insert 10 million rows  
INSERT INTO products (product\_name, category, price, stock\_qty)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 (random()\*500)::NUMERIC(10,2),  
 (random()\*100)::INT  
FROM generate\_series(1, 10000000) g;  
ANALYZE products;

INSERT INTO products (product\_name, category, price, stock\_qty)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 (random()\*500)::NUMERIC(10,2),  
 (random()\*100)::INT  
FROM generate\_series(1, 10000000) g;  
ANALYZE products;  
INSERT 0 10000000  
ANALYZE  
postgres=#

## **Step 2: Add Some Indexes**

For demonstration, we’ll add three indexes:

-- Useful index for frequent category lookups  
CREATE INDEX idx\_products\_category ON products(category);

postgres=# CREATE INDEX idx\_products\_category ON products(category);  
CREATE INDEX  
postgres=#

-- Rarely used index on product\_name  
CREATE INDEX idx\_products\_name ON products(product\_name);  
  
  
-- Probably useless index on stock quantity  
CREATE INDEX idx\_products\_stock\_qty ON products(stock\_qty);

postgres=# -- Rarely used index on product\_name  
CREATE INDEX idx\_products\_name ON products(product\_name);  
CREATE INDEX  
postgres=#

postgres=# -- Probably useless index on stock quantity  
CREATE INDEX idx\_products\_stock\_qty ON products(stock\_qty);  
CREATE INDEX  
postgres=#

At this point, we don’t yet know which indexes are helpful and which are wasteful.

## **Step 3: Check Index Usage with**pg\_stat\_user\_indexes

PostgreSQL tracks how often indexes are used. We can query this data:

SELECT relname AS table\_name,  
 indexrelname AS index\_name,  
 idx\_scan,  
 idx\_tup\_read,  
 idx\_tup\_fetch,  
 pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size  
FROM pg\_stat\_user\_indexes  
JOIN pg\_index USING (indexrelid)  
WHERE relname = 'products'  
ORDER BY idx\_scan ASC;

## **Sample Output for using without index**

postgres=# SELECT relname AS table\_name,  
 indexrelname AS index\_name,  
 idx\_scan,  
 idx\_tup\_read,  
 idx\_tup\_fetch,  
 pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size  
FROM pg\_stat\_user\_indexes  
JOIN pg\_index USING (indexrelid)  
WHERE relname = 'products'  
ORDER BY idx\_scan ASC;  
 table\_name | index\_name | idx\_scan | idx\_tup\_read | idx\_tup\_fetch | index\_size  
------------+------------------------+----------+--------------+---------------+------------  
 products | products\_pkey | 0 | 0 | 0 | 216 MB  
 products | idx\_products\_category | 0 | 0 | 0 | 67 MB  
 products | idx\_products\_name | 0 | 0 | 0 | 299 MB  
 products | idx\_products\_stock\_qty | 0 | 0 | 0 | 65 MB  
(4 rows)  
  
postgres=#  
postgres=#

SELECT \* FROM products WHERE product\_id = 5000000;

postgres=# SELECT \* FROM products WHERE product\_id = 5000000;  
 product\_id | product\_name | category | price | stock\_qty  
------------+-----------------+------------+--------+-----------  
 5000000 | Product\_5000000 | Category\_0 | 152.44 | 78  
(1 row)  
  
postgres=#

SELECT \* FROM products WHERE category = 'Category\_20';

postgres=# SELECT \* FROM products WHERE category = 'Category\_20';  
 product\_id | product\_name | category | price | stock\_qty  
------------+-----------------+-------------+--------+-----------  
 20 | Product\_20 | Category\_20 | 469.36 | 30  
 70 | Product\_70 | Category\_20 | 481.26 | 31  
 120 | Product\_120 | Category\_20 | 318.93 | 28  
 170 | Product\_170 | Category\_20 | 12.32 | 93  
 220 | Product\_220 | Category\_20 | 113.61 | 43  
 270 | Product\_270 | Category\_20 | 470.95 | 68  
 320 | Product\_320 | Category\_20 | 368.27 | 85  
 370 | Product\_370 | Category\_20 | 464.31 | 34  
 420 | Product\_420 | Category\_20 | 255.68 | 16  
 470 | Product\_470 | Category\_20 | 88.36 | 58  
 520 | Product\_520 | Category\_20 | 233.06 | 11  
 570 | Product\_570 | Category\_20 | 47.56 | 4

SELECT relname AS table\_name,  
 indexrelname AS index\_name,  
 idx\_scan,  
 idx\_tup\_read,  
 idx\_tup\_fetch,  
 pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size  
FROM pg\_stat\_user\_indexes  
JOIN pg\_index USING (indexrelid)  
WHERE relname = 'products'  
ORDER BY idx\_scan ASC;

## **Step 4: Sample Output**

Example results might look like this:

postgres=# SELECT relname AS table\_name,  
 indexrelname AS index\_name,  
 idx\_scan,  
 idx\_tup\_read,  
 idx\_tup\_fetch,  
 pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size  
FROM pg\_stat\_user\_indexes  
JOIN pg\_index USING (indexrelid)  
WHERE relname = 'products'  
ORDER BY idx\_scan ASC;  
 table\_name | index\_name | idx\_scan | idx\_tup\_read | idx\_tup\_fetch | index\_size  
------------+------------------------+----------+--------------+---------------+------------  
 products | idx\_products\_stock\_qty | 0 | 0 | 0 | 250 MB  
 products | idx\_products\_name | 12 | 500000 | 500000 | 400 MB  
 products | idx\_products\_category | 20000 | 25000000 | 25000000 | 300 MB  
  
postgres=#

👉 Interpretation:

* ****idx\_products\_category**** → scanned 20,000 times, fetched 25M rows. Clearly useful.
* ****idx\_products\_name**** → scanned only 12 times. Might be justified if occasional lookups by product name are business-critical.
* ****idx\_products\_stock\_qty**** → ****never used**** (idx\_scan = 0), yet it consumes 250 MB. This is a waste.

## **Step 5: Deciding What to Do**

* If an index shows ****high usage**** → keep it.
* If it shows ****low or occasional usage**** → consult with developers/business before removing. It might be used in rare but critical reports.
* If it shows ****zero usage over a long period**** → it’s a candidate for dropping.

## **Step 6: Dropping an Unused Index**

If we decide that idx\_products\_stock\_qty is wasteful, we can safely remove it:

DROP INDEX idx\_products\_stock\_qty;

postgres=# DROP INDEX idx\_products\_stock\_qty;  
DROP INDEX  
postgres=#

Now the database has:

* Less storage overhead.
* Faster writes (INSERT/UPDATE/DELETE).
* Lower maintenance costs during VACUUM and reindexing.

## **Step 7: Continuous Monitoring**

Finding unused indexes is not a one-time task. Query patterns change over time. A rarely used index today may become critical tomorrow after an application change.

That’s why it’s best practice to:

* Regularly monitor pg\_stat\_user\_indexes.
* Integrate index usage metrics into your monitoring dashboards (e.g., Grafana, pgBadger, or custom queries).
* Review index usage before major schema changes.

## **Key Takeaways**

* ****Unused indexes**** harm performance by slowing writes and wasting space.
* PostgreSQL 17 provides ****system views (pg\_stat\_user\_indexes)**** to spot them.
* Always validate with developers before dropping indexes — some may serve niche but important queries.
* Regular monitoring ensures your database remains lean and efficient.

✅ Index tuning is not only about ****adding indexes**** where they are missing, but also about ****removing the ones you don’t need****.