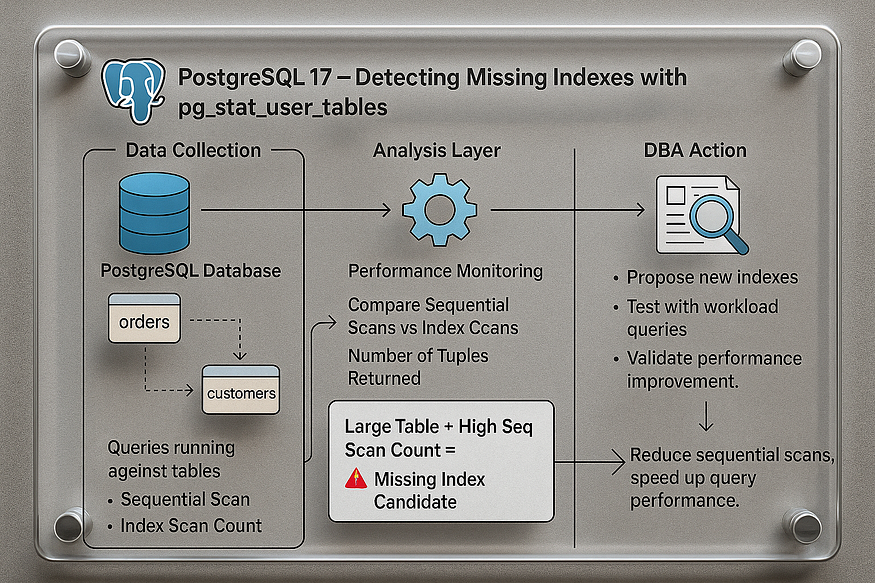
# **26 - PostgreSQL 17 Performance Tuning: Detecting Missing Indexes with**pg\_stat\_user\_tables



Collecting statistics is only the first step — the real challenge is making sense of the data. Simply reading counts of scans, tuples, or dead rows won’t improve performance unless you know how to interpret them.

One practical way to use pg\_stat\_user\_tables is to ****detect which tables might need an index****. The approach is simple:

* Look for ****large tables****.
* Check if they are frequently accessed with ****sequential scans****.
* If a big table consistently shows high sequential scan counts, it is a strong candidate for indexing.

## **Step 1: Create a Products Table with 10 Million Rows**

For demonstration, let’s create a realistic table and load it with data.

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 % 100), -- 100 categories  
 (random()\*500)::NUMERIC(10,2),  
 (random()\*100)::INT  
FROM generate\_series(1, 10000000) g;  
ANALYZE products;

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

This gives us a large table that will generate meaningful scan statistics.

## **Step 2: Run Some Queries to Generate Load**

-- Category filter (no index on category yet)  
SELECT \* FROM products WHERE category = 'Category\_42';

postgres=# SELECT \* FROM products WHERE category = 'Category\_42';  
 product\_id | product\_name | category | price | stock\_qty  
------------+-----------------+-------------+--------+-----------  
 42 | Product\_42 | Category\_42 | 240.05 | 70  
 142 | Product\_142 | Category\_42 | 12.28 | 1  
 242 | Product\_242 | Category\_42 | 228.58 | 37  
 342 | Product\_342 | Category\_42 | 341.42 | 58  
 442 | Product\_442 | Category\_42 | 370.30 | 15  
 542 | Product\_542 | Category\_42 | 100.69 | 73  
 642 | Product\_642 | Category\_42 | 47.06 | 30  
 742 | Product\_742 | Category\_42 | 434.37 | 94  
 842 | Product\_842 | Category\_42 | 228.29 | 9  
 942 | Product\_942 | Category\_42 | 197.54 | 28  
 1042 | Product\_1042 | Category\_42 | 163.16 | 98  
 1142 | Product\_1142 | Category\_42 | 244.24 | 27  
 1242 | Product\_1242 | Category\_42 | 50.84 | 43  
 1342 | Product\_1342 | Category\_42 | 0.04 | 79

-- Price filter  
SELECT \* FROM products WHERE price > 250;  
  
-- Update by category  
UPDATE products SET stock\_qty = stock\_qty + 1  
WHERE category = 'Category\_10';

postgres=# UPDATE products SET stock\_qty = stock\_qty + 1  
WHERE category = 'Category\_10';  
UPDATE 100000  
postgres=#

Without indexes on category or price, these queries will trigger ****sequential scans****.

## **Step 3: Identify Tables with Heavy Sequential Scans**

You can run the following query to detect which tables might need an index:

SELECT relname AS table\_name,  
 seq\_scan,  
 seq\_tup\_read,  
 idx\_scan,  
 idx\_tup\_fetch,  
 n\_live\_tup  
FROM pg\_stat\_user\_tables  
ORDER BY seq\_tup\_read DESC  
LIMIT 10;

👉 This query highlights:

* ****seq\_scan**** → number of sequential scans.
* ****seq\_tup\_read**** → total rows read by sequential scans.
* ****idx\_scan**** → number of index scans.
* ****n\_live\_tup**** → approximate live row count in the table.

Large tables at the ****top of this list**** with ****high sequential scan counts**** are often missing indexes.

## **Step 4: Interpreting the Results**

Example output:

postgres=# UPDATE products SET stock\_qty = stock\_qty + 1  
WHERE category = 'Category\_10';  
UPDATE 100000  
postgres=# SELECT relname AS table\_name,  
 seq\_scan,  
 seq\_tup\_read,  
 idx\_scan,  
 idx\_tup\_fetch,  
 n\_live\_tup  
FROM pg\_stat\_user\_tables  
ORDER BY seq\_tup\_read DESC  
LIMIT 10;  
 table\_name | seq\_scan | seq\_tup\_read | idx\_scan | idx\_tup\_fetch | n\_live\_tup  
------------+----------+--------------+----------+---------------+------------  
 products | 6 | 28639097 | 0 | 0 | 10000000  
(1 row)  
  
postgres=#

* The products table shows ****millions of tuples read sequentially****.
* No index scans were used, meaning PostgreSQL has no choice but to scan the whole table.
* On a large dataset, this will ****seriously degrade performance**** for frequent queries.

## **Important Note**

Not all sequential scans are bad. They are natural in cases like:

* Full-table analytical queries (OLAP).
* Backups or reporting jobs.

But if you see ****frequent large sequential scans in transactional workloads (OLTP)****, performance will inevitably degrade.

👉 Experience shows that ****missing indexes are the #1 cause of poor performance in PostgreSQL****. By monitoring pg\_stat\_user\_tables and focusing on tables with high sequential scans, you can systematically detect and fix indexing gaps.

✅ With this method, PostgreSQL performance tuning becomes ****fact-driven****: you’re not guessing, but identifying exactly which tables and queries need indexing support.

## **Example: Missing Indexes with**pg\_stat\_user\_tables

## **Step 1: Add Indexes**

Now, let’s fix it with proper indexes:

CREATE INDEX idx\_products\_category ON products(category);  
CREATE INDEX idx\_products\_price ON products(price);  
ANALYZE products;

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

## **Step 5: Run Queries Again**

-- Now this query should use an index  
SELECT \* FROM products WHERE category = 'Category\_42';

-- This one too  
SELECT \* FROM products WHERE price > 300;

## **Step 6: Check Statistics Again**

SELECT relname AS table\_name,  
 seq\_scan,  
 seq\_tup\_read,  
 idx\_scan,  
 idx\_tup\_fetch  
FROM pg\_stat\_user\_tables  
WHERE relname = 'products';

📊 ****Sample Output (after indexing):****

postgres=# SELECT relname AS table\_name,  
 seq\_scan,  
 seq\_tup\_read,  
 idx\_scan,  
 idx\_tup\_fetch  
FROM pg\_stat\_user\_tables  
WHERE relname = 'products';  
 table\_name | seq\_scan | seq\_tup\_read | idx\_scan | idx\_tup\_fetch  
------------+----------+--------------+----------+---------------  
 products | 11 | 58584731 | 1 | 3562964  
(1 row)  
  
postgres=#

* Sequential scans remain for earlier queries, but ****new queries now use indexes****.
* ****idx\_scan = 120**** → PostgreSQL chose indexes instead of scanning the whole table.
* ****idx\_tup\_fetch**** shows how many rows were efficiently fetched via the index.

## **Key Takeaway**

* ****Before indexing**** → 10M rows were read repeatedly via sequential scans.
* ****After indexing**** → PostgreSQL switched to index scans, reading only relevant rows, saving massive I/O.
* ****Practical lesson**** → Monitor pg\_stat\_user\_tables, and when you see frequent sequential scans on large tables, it’s a strong signal you need indexes.

✅ This example shows how PostgreSQL 17 makes it easy to ****spot missing indexes**** using table statistics and how a single change can dramatically improve query performance.