# **24 - PostgreSQL 17 Performance Tuning: Monitoring Table-Level Statistics with**pg\_stat\_user\_tables



When tuning PostgreSQL, one of the most important steps is to ****observe table-level statistics****. You cannot optimize what you cannot measure. PostgreSQL provides the system view ****pg\_stat\_user\_tables****, which tracks how your tables are accessed, how many tuples (rows) are alive or dead, and when maintenance operations like VACUUM last occurred.

This system view is invaluable for performance tuning — when used properly, it can reveal how your queries and workloads truly interact with your data.

## **Step 1: Setup Demo with a Products Table**

Let’s first create a large table (products) with multiple columns and insert 10 million rows to simulate a real workload.

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), -- 50 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 % 50), -- 50 categories  
 (random()\*500)::NUMERIC(10,2),  
 (random()\*100)::INT  
FROM generate\_series(1, 10000000) g;  
ANALYZE products;  
INSERT 0 10000000  
ANALYZE  
postgres=#

Now we have a realistic table large enough to observe ****sequential scans, index scans, and vacuum behavior****.

## **Step 2: Generate Some Load**

To simulate activity, run queries that update and read from this table:

-- Update stock for a specific category  
UPDATE products SET stock\_qty = stock\_qty + 1  
WHERE category = 'Category\_10';

postgres=# -- Update stock for a specific category  
UPDATE products SET stock\_qty = stock\_qty + 1  
WHERE category = 'Category\_10';  
UPDATE 200000  
postgres=#

-- Select by primary key (index scan)  
SELECT \* FROM products WHERE product\_id = 5000000;  
  
  
-- Select by category (likely sequential scan if no index exists)  
SELECT \* FROM products WHERE category = 'Category\_20';

postgres=# SELECT \* FROM products WHERE product\_id = 5000000;  
 product\_id | product\_name | category | price | stock\_qty  
------------+-----------------+------------+--------+-----------  
 5000000 | Product\_5000000 | Category\_0 | 194.44 | 85  
(1 row)  
  
postgres=#

postgres=# -- Select by category (likely sequential scan if no index exists)  
SELECT \* FROM products WHERE category = 'Category\_2';  
 product\_id | product\_name | category | price | stock\_qty  
------------+-------------------+------------+--------+-----------  
 252 | Product\_252 | Category\_2 | 236.69 | 73  
 302 | Product\_302 | Category\_2 | 9.44 | 50  
 352 | Product\_352 | Category\_2 | 310.24 | 24  
 402 | Product\_402 | Category\_2 | 175.06 | 52  
 452 | Product\_452 | Category\_2 | 74.66 | 91  
 502 | Product\_502 | Category\_2 | 73.18 | 98  
 2402 | Product\_2402 | Category\_2 | 57.82 | 87  
 2452 | Product\_2452 | Category\_2 | 136.81 | 25  
 2502 | Product\_2502 | Category\_2 | 490.25 | 12  
 2552 | Product\_2552 | Category\_2 | 456.85 | 6  
 2602 | Product\_2602 | Category\_2 | 215.48 | 55  
 2652 | Product\_2652 | Category\_2 | 278.45 | 53  
 2702 | Product\_2702 | Category\_2 | 358.29 | 64  
 2752 | Product\_2752 | Category\_2 | 178.38 | 24  
 2802 | Product\_2802 | Category\_2 | 77.39 | 23

Running these repeatedly (or using a tool like pgbench for load testing) will produce useful statistics in pg\_stat\_user\_tables.

## **Step 3: Inspect**pg\_stat\_user\_tables

Now, query the system view to see how PostgreSQL is handling the table:

SELECT relname,  
 n\_live\_tup,  
 n\_dead\_tup,  
 seq\_scan,  
 seq\_tup\_read,  
 idx\_scan,  
 idx\_tup\_fetch,  
 last\_vacuum,  
 last\_autovacuum  
FROM pg\_stat\_user\_tables  
WHERE relname = 'products';

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

* ****n\_live\_tup**** → current estimate of live rows in the table.
* ****n\_dead\_tup**** → number of dead tuples waiting for cleanup (important for monitoring VACUUM effectiveness).
* ****seq\_scan / seq\_tup\_read**** → how many sequential scans occurred and how many rows they processed.
* ****idx\_scan / idx\_tup\_fetch**** → how many index scans occurred and how many rows they fetched.
* ****last\_vacuum / last\_autovacuum**** → when the table was last vacuumed.

For example, you may observe:

postgres=# SELECT relname,  
 n\_live\_tup,  
 n\_dead\_tup,  
 seq\_scan,  
 seq\_tup\_read,  
 idx\_scan,  
 idx\_tup\_fetch,  
 last\_vacuum,  
 last\_autovacuum  
FROM pg\_stat\_user\_tables  
WHERE relname = 'products';  
 relname | n\_live\_tup | n\_dead\_tup | seq\_scan | seq\_tup\_read | idx\_scan | idx\_tup\_fetch | last\_vacuum | last\_autovacuum  
----------+------------+------------+----------+--------------+----------+---------------+-------------+-------------------------------  
 products | 9798007 | 606000 | 10 | 60398000 | 5 | 7047086 | | 2025-09-07 14:58:24.914076+00  
(1 row)  
  
postgres=#

👉 This means:

* Around ****50,000 dead tuples**** exist, which VACUUM will eventually reclaim.
* Sequential scans processed millions of rows — showing that some queries are likely ****not using indexes****.
* Index scans fetched 600,000 rows across 6,000 scans — showing frequent lookups by indexed columns.

## **Why This Matters**

* If ****dead tuples keep growing**** and VACUUM isn’t cleaning them efficiently, you risk ****table bloat**** and performance degradation.
* If ****sequential scans dominate****, queries may be missing useful indexes or the planner finds them cheaper due to data distribution.
* Both ****number of scans**** and ****rows fetched**** are critical metrics — they reveal not just how queries run, but their total impact on your database.

✅ By continuously monitoring pg\_stat\_user\_tables, you can identify:

* When to VACUUM or ANALYZE more aggressively.
* Which queries need indexing help.
* Whether workloads are efficiently using PostgreSQL’s storage and planner decisions.