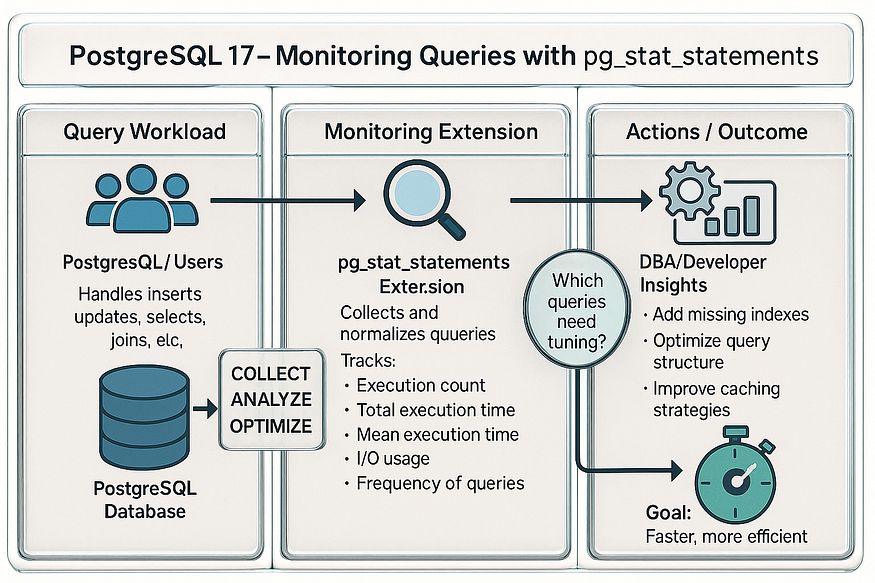
# **27 - PostgreSQL 17 Performance Tuning: Monitoring Queries with**pg\_stat\_statements



You cannot improve database performance without first collecting the right data. PostgreSQL provides a wide range of ****system views**** that let administrators and developers understand what’s really happening inside their database.

One of the most powerful tools for this purpose is the ****pg\_stat\_statements**** extension. It helps you identify:

* Which queries are slow.
* How often queries are executed.
* Whether queries have stable or fluctuating runtimes.

Instead of guessing or parsing log files, pg\_stat\_statements provides a clear, aggregated view of query behavior.

## **Step 1: 🧩 Installing PostgreSQL Extension Modules**

To enable advanced features like pg\_stat\_statements, you need to install the PostgreSQL contrib package. This package includes a collection of officially supported extensions that enhance PostgreSQL's functionality.

On Red Hat-based systems (RHEL, CentOS, Amazon Linux), run:

sudo yum install postgresql17-contrib

[ec2-user@ip-172-31-29-78 ~]$ sudo yum install postgresql17-contrib  
Updating Subscription Management repositories.  
Unable to read consumer identity  
  
This system is not registered with an entitlement server. You can use "rhc" or "subscription-manager" to register.  
  
Last metadata expiration check: 2:44:15 ago on Fri Sep 19 18:47:07 2025.  
Dependencies resolved.  
===================================================================================================================================================================================  
 Package Architecture Version Repository Size  
===================================================================================================================================================================================  
Installing:  
 postgresql17-contrib x86\_64 17.6-1PGDG.rhel10 pgdg17 731 k  
Installing dependencies:  
 libxslt x86\_64 1.1.39-8.el10\_0 rhel-10-appstream-rhui-rpms 190 k  
  
Transaction Summary  
===================================================================================================================================================================================  
Install 2 Packages  
  
Total download size: 920 k  
Installed size: 3.2 M  
Is this ok [y/N]: y  
Downloading Packages:  
(1/2): libxslt-1.1.39-8.el10\_0.x86\_64.rpm 4.7 MB/s | 190 kB 00:00  
(2/2): postgresql17-contrib-17.6-1PGDG.rhel10.x86\_64.rpm 5.4 MB/s | 731 kB 00:00  
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
Total 5.6 MB/s | 920 kB 00:00  
Running transaction check  
Transaction check succeeded.  
Running transaction test  
Transaction test succeeded.  
Running transaction  
 Preparing : 1/1  
 Installing : libxslt-1.1.39-8.el10\_0.x86\_64 1/2  
 Installing : postgresql17-contrib-17.6-1PGDG.rhel10.x86\_64 2/2  
 Running scriptlet: postgresql17-contrib-17.6-1PGDG.rhel10.x86\_64 2/2  
Installed products updated.  
  
Installed:  
 libxslt-1.1.39-8.el10\_0.x86\_64 postgresql17-contrib-17.6-1PGDG.rhel10.x86\_64  
  
Complete!  
[ec2-user@ip-172-31-29-78 ~]$

This command:

* Installs the ****contrib module**** for PostgreSQL 17
* Adds support for extensions like pg\_stat\_statements, tablefunc, uuid-ossp, and more
* Is required before enabling certain features in postgresql.conf (e.g., shared\_preload\_libraries = 'pg\_stat\_statements')

🛠️ After installation, don’t forget to restart the PostgreSQL service and run CREATE EXTENSION pg\_stat\_statements; inside your database to activate it.

## **Step 2: Enable**pg\_stat\_statements

By default, this extension is not active. You need to enable it in postgresql.conf.

1. Open your PostgreSQL config file (e.g., postgresql.conf).
2. Find the line:

#shared\_preload\_libraries = ''

postgres=# show shared\_preload\_libraries;  
 shared\_preload\_libraries  
--------------------------  
  
(1 row)  
  
postgres=#

3. Uncomment it and set the value to include pg\_stat\_statements:

shared\_preload\_libraries = 'pg\_stat\_statements'

[postgres@ip-172-31-29-78 data]$ cat postgresql.conf | grep -i shared\_preload\_libraries  
shared\_preload\_libraries = 'pg\_stat\_statements' # (change requires restart)  
[postgres@ip-172-31-29-78 data]$

postgres=# show shared\_preload\_libraries;  
 shared\_preload\_libraries  
--------------------------  
 pg\_stat\_statements  
(1 row)  
  
postgres=#

4. Restart PostgreSQL for the change to take effect.

[ec2-user@ip-172-31-29-78 ~]$ sudo systemctl restart postgresql-17  
[ec2-user@ip-172-31-29-78 ~]$

Finally, install the extension in the database you want to monitor:

CREATE EXTENSION pg\_stat\_statements;

This will create the system view pg\_stat\_statements.

postgres=# CREATE EXTENSION pg\_stat\_statements;  
CREATE EXTENSION  
postgres=#

## **Step 3: Generate Some Demo Load**

For demonstration, let’s create a products table with 10 million rows and run a few queries against it.

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

postgres=# CREATE EXTENSION pg\_stat\_statements;  
CREATE EXTENSION  
postgres=#  
postgres=# drop table products;  
DROP TABLE  
postgres=# CREATE TABLE products (  
 product\_id BIGSERIAL PRIMARY KEY,  
 product\_name TEXT,  
 category TEXT,  
 price NUMERIC,  
 stock\_qty INT  
);  
CREATE TABLE  
postgres=#

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

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

Now run queries such as:

SELECT \* FROM products WHERE product\_id = 100;  
SELECT \* FROM products WHERE category = 'Category\_3';  
UPDATE products SET stock\_qty = stock\_qty + 1 WHERE category = 'Category\_5';

postgres=# SELECT \* FROM products WHERE product\_id = 100;  
 product\_id | product\_name | category | price | stock\_qty  
------------+--------------+------------+------------------+-----------  
 100 | Product\_100 | Category\_0 | 228.865334176435 | 36  
(1 row)  
  
postgres=#

postgres=# SELECT \* FROM products WHERE category = 'Category\_3';  
 product\_id | product\_name | category | price | stock\_qty  
------------+-----------------+------------+----------------------+-----------  
 3 | Product\_3 | Category\_3 | 216.496292091521 | 9  
 13 | Product\_13 | Category\_3 | 158.801528884299 | 50  
 23 | Product\_23 | Category\_3 | 402.177798888444 | 60  
 33 | Product\_33 | Category\_3 | 188.638645203066 | 26  
 43 | Product\_43 | Category\_3 | 217.520085735964 | 95  
 53 | Product\_53 | Category\_3 | 204.260688438616 | 43  
 63 | Product\_63 | Category\_3 | 4.38792827933232 | 36  
 73 | Product\_73 | Category\_3 | 179.192418212635 | 13  
 83 | Product\_83 | Category\_3 | 177.750125505008 | 33  
 93 | Product\_93 | Category\_3 | 406.374932408794 | 32  
 103 | Product\_103 | Category\_3 | 47.2825437678309 | 5  
 113 | Product\_113 | Category\_3 | 347.047753193479 | 34  
 123 | Product\_123 | Category\_3 | 243.453650026469 | 46  
 133 | Product\_133 | Category\_3 | 400.878856999775 | 99  
 143 | Product\_143 | Category\_3 | 110.710054872779 | 84  
 153 | Product\_153 | Category\_3 | 422.519471685829 | 43  
 163 | Product\_163 | Category\_3 | 418.99101277994 | 10  
 173 | Product\_173 | Category\_3 | 330.138006726809 | 66  
 183 | Product\_183 | Category\_3 | 266.622715912463 | 60

postgres=# UPDATE products SET stock\_qty = stock\_qty + 1 WHERE category = 'Category\_5';  
UPDATE 1000000  
postgres=#

## **Step 4: Query**pg\_stat\_statements

Now check what PostgreSQL recorded:

SELECT query, calls, total\_exec\_time, mean\_exec\_time, rows, stddev\_exec\_time  
FROM pg\_stat\_statements  
ORDER BY total\_exec\_time DESC  
LIMIT 10;

👉 Each row in this view represents one ****normalized query****. Queries with different parameter values are aggregated together. For example:

* SELECT \* FROM products WHERE product\_id = 100;
* SELECT \* FROM products WHERE product\_id = 200;

Both are shown as:

postgres=# SELECT query, calls, total\_exec\_time, mean\_exec\_time, rows, stddev\_exec\_time  
FROM pg\_stat\_statements  
ORDER BY total\_exec\_time DESC  
LIMIT 10;  
 query | calls | total\_exec\_time | mean\_exec\_time | rows | stddev\_exec\_time  
--------------------------------------------------------------------+-------+--------------------+--------------------+----------+-----------------------  
 INSERT INTO products (product\_name, category, price, stock\_qty) +| 1 | 33720.995524000005 | 33720.995524000005 | 10000000 | 0  
 SELECT +| | | | |  
 $1 || g, +| | | | |  
 $2 || (g % $3), +| | | | |  
 (random()\*$4)::NUMERIC, +| | | | |  
 (random()\*$5)::INT +| | | | |  
 FROM generate\_series($6, $7) g | | | | |  
 UPDATE products SET stock\_qty = stock\_qty + $1 WHERE category = $2 | 1 | 25936.125529 | 25936.125529 | 1000000 | 0  
 ANALYZE products | 1 | 7514.966781 | 7514.966781 | 0 | 0  
 SELECT \* FROM products WHERE category = $1 | 1 | 6051.37657 | 6051.37657 | 1000000 | 0  
 CREATE EXTENSION pg\_stat\_statements | 1 | 51.733621 | 51.733621 | 0 | 0  
 CREATE TABLE products ( +| 1 | 7.303899 | 7.303899 | 0 | 0  
 product\_id BIGSERIAL PRIMARY KEY, +| | | | |  
 product\_name TEXT, +| | | | |  
 category TEXT, +| | | | |  
 price NUMERIC, +| | | | |  
 stock\_qty INT +| | | | |  
 ) | | | | |  
 drop table products | 1 | 6.519512 | 6.519512 | 0 | 0  
 SELECT \* FROM products WHERE product\_id = $1 | 1 | 1.144964 | 1.144964 | 1 | 0  
 SELECT query, calls, total\_exec\_time, mean\_exec\_time, rows +| 1 | 0.088163 | 0.088163 | 9 | 0  
 FROM pg\_stat\_statements +| | | | |  
 ORDER BY total\_exec\_time DESC +| | | | |  
 LIMIT $1 | | | | |  
 show shared\_preload\_libraries | 2 | 0.02019 | 0.010095 | 0 | 0.0014140000000000003  
(10 rows)  
  
postgres=#

This makes it easier to measure query patterns, not just individual runs.

## **Step 5: Useful Columns in**pg\_stat\_statements

* ****calls**** → how often the query ran.
* ****total\_exec\_time**** → total time spent across all executions.
* ****mean\_exec\_time**** → average execution time.
* ****rows**** → average number of rows returned.
* ****stddev\_exec\_time**** → (standard deviation) shows if query times are stable or fluctuate heavily.

## **Why is this important?**

* If runtimes are unstable, it could mean the query sometimes hits the cache and sometimes reads from disk.
* ****Shared blocks vs read blocks**** columns help identify whether data came from PostgreSQL’s buffer cache (shared\_blks\_hit) or from disk (shared\_blks\_read).

SELECT  
 query,  
 calls,  
 shared\_blks\_hit,  
 shared\_blks\_read,  
 ROUND(shared\_blks\_hit \* 100.0 / GREATEST(shared\_blks\_hit + shared\_blks\_read, 1), 2) AS hit\_ratio  
FROM  
 pg\_stat\_statements  
ORDER BY  
 hit\_ratio ASC  
LIMIT 20;

query,  
 calls,  
 shared\_blks\_hit,  
 shared\_blks\_read,  
 ROUND(shared\_blks\_hit \* 100.0 / GREATEST(shared\_blks\_hit + shared\_blks\_read, 1), 2) AS hit\_ratio  
FROM  
 pg\_stat\_statements  
ORDER BY  
 hit\_ratio ASC  
LIMIT 20;  
 query | calls | shared\_blks\_hit | shared\_blks\_read | hit\_ratio  
------------------------------------------------------------------------------+-------+-----------------+------------------+-----------  
 show shared\_preload\_libraries | 2 | 0 | 0 | 0.00  
 SELECT query, calls, total\_exec\_time, mean\_exec\_time, rows, stddev\_exec\_time+| 1 | 0 | 0 | 0.00  
 FROM pg\_stat\_statements +| | | |  
 ORDER BY total\_exec\_time DESC +| | | |  
 LIMIT $1 | | | |  
 SELECT \* FROM products WHERE category = $1 | 1 | 12867 | 90148 | 12.49  
 ANALYZE products | 1 | 3909 | 26339 | 12.92  
 SELECT \* FROM products WHERE product\_id = $1 | 1 | 4 | 3 | 57.14  
 CREATE TABLE products ( +| 1 | 630 | 25 | 96.18  
 product\_id BIGSERIAL PRIMARY KEY, +| | | |  
 product\_name TEXT, +| | | |  
 category TEXT, +| | | |  
 price NUMERIC, +| | | |  
 stock\_qty INT +| | | |  
 ) | | | |  
 CREATE EXTENSION pg\_stat\_statements | 1 | 3380 | 125 | 96.43  
 drop table products | 1 | 488 | 16 | 96.83  
 UPDATE products SET stock\_qty = stock\_qty + $1 WHERE category = $2 | 1 | 7096086 | 130436 | 98.20  
 SELECT query, calls, total\_exec\_time, mean\_exec\_time, rows +| 1 | 3 | 0 | 100.00  
 FROM pg\_stat\_statements +| | | |  
 ORDER BY total\_exec\_time DESC +| | | |  
 LIMIT $1 | | | |  
 INSERT INTO products (product\_name, category, price, stock\_qty) +| 1 | 30516981 | 1 | 100.00  
 SELECT +| | | |  
 $1 || g, +| | | |  
 $2 || (g % $3), +| | | |  
 (random()\*$4)::NUMERIC, +| | | |  
 (random()\*$5)::INT +| | | |  
 FROM generate\_series($6, $7) g | | | |  
(11 rows)  
  
postgres=#

## **Step 6: Tracking Temporary File Usage**

Queries that use too much memory can spill to disk, creating ****temporary files****. In OLTP systems (transaction-heavy workloads), this is usually a red flag because it slows everything down.

pg\_stat\_statements tracks this too. High temporary file I/O may indicate:

* Work memory (work\_mem) is too low.
* Large sorts, hash joins, or index builds are happening.

SELECT  
 query,  
 calls,  
 temp\_blks\_written,  
 temp\_blks\_read,  
 ROUND(temp\_blks\_written \* 8 / 1024.0, 2) AS temp\_written\_MB,  
 ROUND(temp\_blks\_read \* 8 / 1024.0, 2) AS temp\_read\_MB  
FROM  
 pg\_stat\_statements  
WHERE  
 temp\_blks\_written > 0  
ORDER BY  
 temp\_blks\_written DESC  
LIMIT 20;

postgres=# SELECT  
 query,  
 calls,  
 temp\_blks\_written,  
 temp\_blks\_read,  
 ROUND(temp\_blks\_written \* 8 / 1024.0, 2) AS temp\_written\_MB,  
 ROUND(temp\_blks\_read \* 8 / 1024.0, 2) AS temp\_read\_MB  
FROM  
 pg\_stat\_statements  
WHERE  
 temp\_blks\_written > 0  
ORDER BY  
 temp\_blks\_written DESC  
LIMIT 20;  
 query | calls | temp\_blks\_written | temp\_blks\_read | temp\_written\_mb | temp\_read\_mb  
-----------------------------------------------------------------+-------+-------------------+----------------+-----------------+--------------  
 INSERT INTO products (product\_name, category, price, stock\_qty)+| 1 | 17090 | 17090 | 133.52 | 133.52  
 SELECT +| | | | |  
 $1 || g, +| | | | |  
 $2 || (g % $3), +| | | | |  
 (random()\*$4)::NUMERIC, +| | | | |  
 (random()\*$5)::INT +| | | | |  
 FROM generate\_series($6, $7) g | | | | |  
(1 row)  
  
postgres=#

## **Step 7: Enable I/O Timing**

By default, I/O timing is not recorded because it adds some overhead. To track it, enable:

track\_io\_timing = on

postgres=# show track\_io\_timing;  
 track\_io\_timing  
-----------------  
 off  
(1 row)  
  
postgres=#

[postgres@ip-172-31-29-78 data]$ cat postgresql.conf | grep -i track\_io\_timing  
track\_io\_timing = on  
[postgres@ip-172-31-29-78 data]$

[ec2-user@ip-172-31-29-78 ~]$ sudo systemctl restart postgresql-17  
[ec2-user@ip-172-31-29-78 ~]$

postgres=# show track\_io\_timing;  
 track\_io\_timing  
-----------------  
 on  
(1 row)  
  
postgres=#

This will let you see how much query time was spent on actual disk I/O.

## **🔍 Using**EXPLAIN (ANALYZE, BUFFERS)**to Diagnose Query Performance**

PostgreSQL’s EXPLAIN (ANALYZE, BUFFERS) command provides deep visibility into how a query interacts with memory and disk. Here's a real-world example:

EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM products WHERE category = 'Category\_5';

****Sample Output:****

postgres=# EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM products WHERE category = 'electronics';  
 QUERY PLAN  
-----------------------------------------------------------------------------------------------------------------------------  
 Gather (cost=1000.00..187116.59 rows=1 width=49) (actual time=5987.833..5998.910 rows=0 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 Buffers: shared hit=12025 read=111592 dirtied=19987 written=13943  
 I/O Timings: shared read=13342.256 write=1658.516  
 -> Parallel Seq Scan on products (cost=0.00..186116.49 rows=1 width=49) (actual time=5934.791..5934.792 rows=0 loops=3)  
 Filter: (category = 'electronics'::text)  
 Rows Removed by Filter: 3333333  
 Buffers: shared hit=12025 read=111592 dirtied=19987 written=13943  
 I/O Timings: shared read=13342.256 write=1658.516  
 Planning:  
 Buffers: shared hit=6 read=3  
 I/O Timings: shared read=1.824  
 Planning Time: 1.928 ms  
 Execution Time: 6000.232 ms  
(15 rows)  
  
postgres=#

## **🧠 What This Tells Us**

* ****Sequential Scan****: PostgreSQL scanned the entire products table—this is often a red flag in OLTP workloads.
* ****Rows Removed by Filter****: 9 million rows were scanned but didn’t match the filter, indicating poor selectivity.
* ****Buffers****:
* shared hit=14584: These blocks were found in memory (PostgreSQL’s buffer cache).
* shared read=109033: These blocks were read from disk—this is expensive and slows down performance.
* ****I/O Timings****:
* shared read=4033.907 ms: Over 4 seconds were spent just reading from disk.
* ****Execution Time****: The query took over 8 seconds to complete.

## **⚠️ Optimization Opportunity**

This query would benefit from:

* A ****B-tree index**** on category:

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

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

EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM products WHERE category = 'Category\_5';

postgres=# EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM products WHERE category = 'Category\_5';  
 QUERY PLAN  
------------------------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on products (cost=11229.68..230035.48 rows=1003000 width=49) (actual time=55.351..5377.773 rows=1000000 loops=1)  
 Recheck Cond: (category = 'Category\_5'::text)  
 Heap Blocks: exact=10321  
 Buffers: shared hit=468 read=10711 written=59  
 I/O Timings: shared read=4543.616 write=0.454  
 -> Bitmap Index Scan on idx\_products\_category (cost=0.00..10978.93 rows=1003000 width=0) (actual time=50.116..50.117 rows=1000000 loops=1)  
 Index Cond: (category = 'Category\_5'::text)  
 Buffers: shared read=858 written=25  
 I/O Timings: shared read=6.589 write=0.144  
 Planning:  
 Buffers: shared hit=43 read=2  
 I/O Timings: shared read=0.945  
 Planning Time: 1.156 ms  
 Execution Time: 5955.813 ms  
(14 rows)  
  
postgres=#

* Increasing work\_mem if sorts or joins are involved.
* Rewriting the query to reduce the result set or improve filtering.

💡 Always pair EXPLAIN (ANALYZE, BUFFERS) with track\_io\_timing = on in postgresql.conf to capture disk latency.

## **Key Takeaways**

* pg\_stat\_statements is the go-to extension for query monitoring in PostgreSQL 17.
* It aggregates queries with parameters, helping spot patterns instead of noise.
* Key insights:
* ****Most expensive queries**** (by time).
* ****Most frequent queries**** (by calls).
* ****Queries with unstable runtimes**** (cache vs disk).
* ****Queries causing temp file I/O**** (tune work\_mem).
* Without monitoring, tuning is guesswork — with pg\_stat\_statements, it’s ****data-driven optimization****.

✅ Before adjusting indexes, memory, or autovacuum, always check what queries are actually causing the load.