# **Monitoring Active Queries in PostgreSQL: Real-Time Performance Diagnostics Using**pg\_stat\_activity

[](https://medium.com/@jramcloud1?source=post_page---byline--cd707a42aee7---------------------------------------)

[Jeyaram Ayyalusamy](https://medium.com/@jramcloud1?source=post_page---byline--cd707a42aee7---------------------------------------)

Following

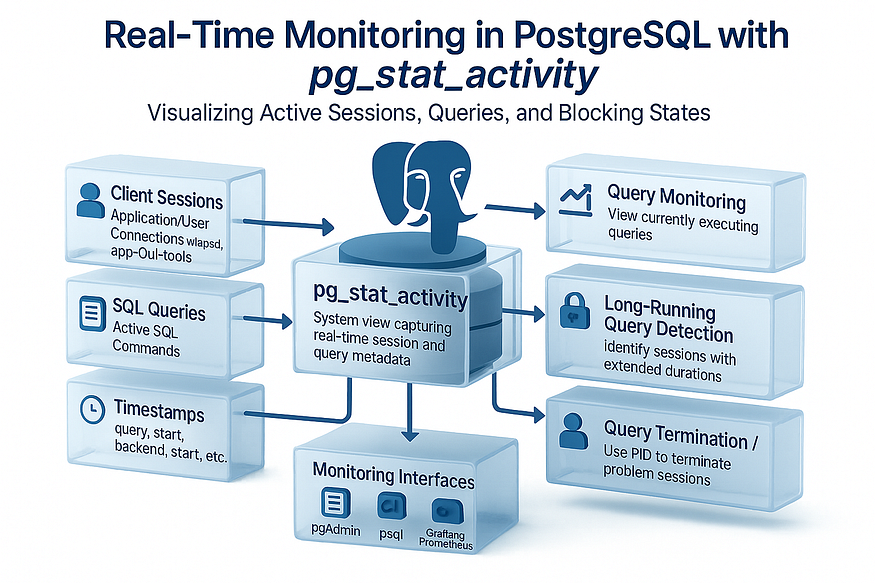
17 min read

·

Jul 12, 2025

52

Press enter or click to view image in full size



PostgreSQL administrators often face the challenge of diagnosing slow performance, managing long-running queries, and identifying blocked sessions — especially in production environments where every second counts. Fortunately, PostgreSQL provides a powerful built-in system view called pg\_stat\_activity, which enables ****real-time visibility into active database sessions****.

In this guide, we’ll walk through a ****practical PostgreSQL monitoring framework**** using SQL queries and scripts that leverage pg\_stat\_activity—ideal for production troubleshooting, performance diagnostics, and query auditing.

## **📘 What Is**pg\_stat\_activity**?**

pg\_stat\_activity is a system catalog view in PostgreSQL that ****exposes real-time information about all active sessions**** connected to the database. It includes details such as:

* Session PID (process ID)
* Username and client address
* Query text and current state
* Backend start and query start time
* Wait events and lock information

This view is invaluable for understanding what your PostgreSQL instance is doing at any given moment.

🧠 Tip: You must be a superuser or have proper monitoring privileges to access all session data.

## **🧪 Core Diagnostic Query: View Active Queries**

To monitor currently running queries:

SELECT pid,  
 usename,  
 datname,  
 client\_addr,  
 application\_name,  
 state,  
 wait\_event\_type,  
 wait\_event,  
 backend\_start,  
 query\_start,  
 now() - query\_start AS runtime,  
 query  
FROM pg\_stat\_activity  
WHERE state = 'active'  
ORDER BY runtime DESC;

pid | usename | datname | client\_addr | application\_name | state | wait\_event\_type | wait\_event | backend\_start | query\_start | runtime | query  
  
-------+----------+----------+-------------+------------------+--------+-----------------+--------------+-------------------------------+-------------------------------+-----------------+-------------------------  
---------------  
 38820 | | postgres | | | active | Timeout | VacuumDelay | 2025-07-11 23:17:00.254164+00 | 2025-07-11 23:17:00.270024+00 | 00:03:03.581424 | autovacuum: VACUUM publi  
c.big\_table  
 38841 | postgres | postgres | | psql | active | IO | DataFileRead | 2025-07-11 23:19:39.354226+00 | 2025-07-11 23:19:39.333873+00 | 00:00:24.517575 | SELECT COUNT(\*) FROM big  
\_table +  
 | | | | | | | | | | | WHERE data LIKE '%zzz%';  
 38840 | postgres | postgres | | psql | active | IO | DataFileRead | 2025-07-11 23:19:39.353725+00 | 2025-07-11 23:19:39.333873+00 | 00:00:24.517575 | SELECT COUNT(\*) FROM big  
\_table +  
 | | | | | | | | | | | WHERE data LIKE '%zzz%';  
 38792 | postgres | postgres | | psql | active | IO | DataFileRead | 2025-07-11 23:14:42.884986+00 | 2025-07-11 23:19:39.333873+00 | 00:00:24.517575 | SELECT COUNT(\*) FROM big  
\_table +  
 | | | | | | | | | | | WHERE data LIKE '%zzz%';  
 38776 | postgres | postgres | | psql | active | | | 2025-07-11 23:13:40.016091+00 | 2025-07-11 23:20:03.851448+00 | 00:00:00 | SELECT pid,  
 +  
 | | | | | | | | | | | usename,  
 +  
 | | | | | | | | | | | datname,  
 +  
 | | | | | | | | | | | client\_addr,  
 +  
 | | | | | | | | | | | application\_name,  
 +  
 | | | | | | | | | | | state,  
 +  
 | | | | | | | | | | | wait\_event\_type,  
 +  
 | | | | | | | | | | | wait\_event,  
 +  
 | | | | | | | | | | | backend\_start,  
 +  
 | | | | | | | | | | | query\_start,  
 +  
 | | | | | | | | | | | now() - query\_sta  
rt AS runtime,+  
 | | | | | | | | | | | query  
 +  
 | | | | | | | | | | | FROM pg\_stat\_activity  
 +  
 | | | | | | | | | | | WHERE state = 'active'  
 +  
 | | | | | | | | | | | ORDER BY runtime DESC;  
(5 rows)

## **✅ What This Tells You:**

* Who is running queries right now
* Which queries are long-running
* What those queries are doing (text + wait state)

📌 Best Practice: Run this query periodically or integrate it into a dashboard for continuous visibility.

## **⏱️ Identify Long-Running Queries**

Long-running queries can lock resources or block other sessions. Use the following query to isolate them:

SELECT pid,  
 now() - query\_start AS duration,  
 usename,  
 query  
FROM pg\_stat\_activity  
WHERE state = 'active'  
 AND now() - query\_start > interval '5 minutes'  
ORDER BY duration DESC;

pid | duration | usename | query  
------+-----------------+----------+---------------------------------------  
 1824 | 00:04:00.363051 | postgres | INSERT INTO big\_table (data) +  
 | | | SELECT repeat('x', 1000) +  
 | | | FROM generate\_series(1, 10000000000);  
(1 row)  
  
postgres=#

## **✅ Best Practice:**

Set the time threshold (interval '5 minutes') based on your workload expectations.

🛑 Consider terminating queries that consistently exceed normal execution time and are impacting system performance.

## **🔒 Detect Blocking Sessions and Wait Events**

PostgreSQL sessions can be blocked waiting for locks or other resources. Identify blockers and blocked processes using:

SELECT blocked.pid AS blocked\_pid,  
 blocked.query AS blocked\_query,  
 blocking.pid AS blocking\_pid,  
 blocking.query AS blocking\_query  
FROM pg\_stat\_activity blocked  
JOIN pg\_locks blocked\_locks  
 ON blocked\_locks.pid = blocked.pid AND NOT blocked\_locks.granted  
JOIN pg\_locks blocking\_locks  
 ON blocking\_locks.locktype = blocked\_locks.locktype  
 AND blocking\_locks.database IS NOT DISTINCT FROM blocked\_locks.database  
 AND blocking\_locks.relation IS NOT DISTINCT FROM blocked\_locks.relation  
 AND blocking\_locks.page IS NOT DISTINCT FROM blocked\_locks.page  
 AND blocking\_locks.tuple IS NOT DISTINCT FROM blocked\_locks.tuple  
 AND blocking\_locks.pid != blocked\_locks.pid  
JOIN pg\_stat\_activity blocking  
 ON blocking.pid = blocking\_locks.pid;

## **✅ What This Reveals:**

* Which session is blocked
* Who is blocking it
* What queries are involved on both sides

🧩 Use this insight to resolve lock contention and prevent cascading performance issues.

## **🔄 Kill Problematic Sessions (With Caution)**

To terminate a session that is blocking others or running a problematic query:

SELECT pg\_terminate\_backend(<pid>);

Replace <pid> with the actual process ID from your diagnostic queries.

⚠️ Warning: Only terminate sessions after reviewing their impact and confirming that they are not part of critical transactions.

## **🖥️ Automate with Cron or Monitoring Tools**

Integrate your pg\_stat\_activity queries with:

* ****Cron jobs**** to log or alert on long-running queries
* ****pgAdmin dashboards**** for real-time visualization
* ****Monitoring stacks**** like Prometheus + Grafana (via PostgreSQL exporters)

💡 Tip: Maintain audit logs of terminated sessions for compliance and root-cause analysis.

## **📊 Sample Monitoring Strategy**

Use Case Query Type Frequency Active session audit All active queries Every minute Long-running query detection Duration > 5 mins Every 5 mins Blocking session detection Lock wait analysis via joins On alert only Session termination pg\_terminate\_backend(pid) Manual review Logging & Reporting Save results to table or log file Nightly

## **✅ Best Practices for Using**pg\_stat\_activity

* ****Filter by state = 'active'**** to focus only on running queries.
* ****Use now() - query\_start**** to calculate execution duration.
* ****Monitor wait\_event\_type and wait\_event**** to diagnose blocking and IO issues.
* ****Avoid terminating sessions blindly**** — always investigate the context.
* ****Combine with pg\_locks**** for full lock conflict analysis.

## **🔍 Monitoring PostgreSQL in Real Time with**pg\_stat\_activity**: Advanced Diagnostics for DBAs**

PostgreSQL is widely recognized for its performance, extensibility, and reliability in handling modern application workloads. However, as database complexity increases, so does the need for visibility into ****live query activity****, ****session management****, and ****transaction monitoring**** — especially in production environments where uptime and performance are critical.

Fortunately, PostgreSQL provides a powerful system view called ****pg\_stat\_activity****, which offers real-time access to vital session-level details such as active queries, client connections, session states, and lock conditions.

In this article, we will take an in-depth look at how to use pg\_stat\_activity to build a ****comprehensive PostgreSQL monitoring framework****, complete with detailed SQL queries and best practices for real-time database diagnostics.

## **🧰 Why Use**pg\_stat\_activity**?**

The pg\_stat\_activity system view captures runtime information about every session connected to the PostgreSQL server. It is one of the most powerful diagnostic tools available to PostgreSQL DBAs and developers.

## **🔍 What It Exposes:**

* ****Current database connections**** (database name, user name, process ID)
* ****Client details**** (IP address, application name, port)
* ****Running SQL queries**** per session
* ****Query state and start time****
* ****Transaction and backend start times****
* ****Wait events and lock statuses****

## **🧠 Why It Matters:**

Using this live view, you can:

* ****Detect long-running or resource-intensive queries**** before they cause application slowdowns
* ****Trace blocked sessions**** and investigate lock contention
* ****Identify idle-in-transaction sessions**** that delay vacuuming and increase table bloat
* ****Proactively terminate problematic backends**** in high-load scenarios
* ****Audit active users and client tools**** interacting with the database

📘 *Access to pg\_stat\_activity requires superuser privileges or specific role-level permissions (e.g., pg\_monitor).*

## **🚦 Key Monitoring Queries Using**pg\_stat\_activity

The following queries are designed to give you immediate visibility into various aspects of PostgreSQL runtime behavior. These can be run directly in psql, scripted for automation, or integrated into dashboards and alerts.

## **1️⃣ View Connected Users and Client IPs**

Understanding who is connected and from where is the foundation of session management.

echo "View the connected users and their clients"  
psql -c "  
SELECT  
 datname,  
 usename,  
 client\_addr,  
 client\_port  
FROM pg\_stat\_activity;"

View the connected users and their clients  
 datname | usename | client\_addr | client\_port  
----------+----------+-------------+-------------  
 postgres | postgres | | -1  
 postgres | postgres | | -1  
 | | |  
 | postgres | |  
 | | |  
 | | |  
 | | |  
(7 rows)  
  
[postgres@ip-172-31-89-173 ~]$

### **🔍 Explanation:**

* datname: The name of the database the user is connected to.
* usename: Username used by the client.
* client\_addr: IP address of the client.
* client\_port: Port on the client side.

✅ ****Use Case:****  
Identify all active sessions and validate legitimate access, especially when investigating security events or session spikes.

🔒 *Look for unknown IPs or users connected during off-hours as part of routine audits.*

## **2️⃣ View Currently Active Queries**

Quickly inspect which SQL statements are being executed in real time.

echo "View active queries"  
psql -c "  
SELECT  
 datname,  
 usename,  
 query  
FROM pg\_stat\_activity  
WHERE state != 'idle';"

View active queries  
 datname | usename | query  
----------+----------+---------------------------------------  
 postgres | postgres | INSERT INTO big\_table (data) +  
 | | SELECT repeat('x', 1000) +  
 | | FROM generate\_series(1, 10000000000);  
 postgres | postgres | +  
 | | SELECT +  
 | | datname, +  
 | | usename, +  
 | | query +  
 | | FROM pg\_stat\_activity +  
 | | WHERE state != 'idle';  
(2 rows)  
  
[postgres@ip-172-31-89-173 ~]$  
[postgres@ip-172-31-89-173 ~]$

### **🔍 Explanation:**

* Filters out connections in an idle state.
* Displays only sessions actively executing SQL commands.

✅ ****Use Case:****  
Live query monitoring during application load testing or peak usage hours to assess system throughput and detect unusual queries.

🧩 Use this query to populate lightweight dashboards or auto-refresh terminal views.

## **3️⃣ Identify Long-Running Queries**

Track queries that may be stuck, looping, or consuming excessive resources.

echo "Long running queries"  
psql -c "  
SELECT  
 current\_timestamp - query\_start AS runtime,  
 datname,  
 usename,  
 query  
FROM pg\_stat\_activity  
WHERE state != 'idle'  
ORDER BY runtime DESC;"

Long running queries  
 runtime | datname | usename | query  
----------+----------+----------+-----------------------------------------------  
 00:00:00 | postgres | postgres | +  
 | | | SELECT +  
 | | | current\_timestamp - query\_start AS runtime,+  
 | | | datname, +  
 | | | usename, +  
 | | | query +  
 | | | FROM pg\_stat\_activity +  
 | | | WHERE state != 'idle' +  
 | | | ORDER BY runtime DESC;  
(1 row)  
  
[postgres@ip-172-31-89-173 ~]$

### **🔍 Explanation:**

* Calculates the duration each query has been running.
* Sorts sessions with the longest runtimes at the top.

✅ ****Use Case:****  
Detect and isolate slow-performing queries in batch jobs, user-generated reports, or ad hoc workloads.

⚠️ *Establish performance thresholds (e.g., 5–10 minutes) to automatically flag or terminate unusually long queries.*

## **4️⃣ Detect Blocked Processes and Lock Contention**

PostgreSQL uses row-level and table-level locking mechanisms. Unresolved locks can stall entire workflows.

echo "Blocked only due to lock waits"  
psql -c "  
SELECT  
 pid,  
 backend\_type  
FROM pg\_stat\_activity  
ORDER BY backend\_type, pid;"

Blocked only due to lock waits  
 pid | backend\_type  
------+------------------------------  
 1769 | autovacuum launcher  
 1766 | background writer  
 1765 | checkpointer  
 1824 | client backend  
 1963 | client backend  
 1770 | logical replication launcher  
 1768 | walwriter  
(7 rows)  
  
[postgres@ip-172-31-89-173 ~]$

### **🔍 Explanation:**

* Lists session process IDs grouped by backend type (e.g., autovacuum, client backend).
* Helps correlate backend activity with lock types.

✅ ****Use Case:****  
Identify which backend processes are involved in locking and which queries may be causing congestion or deadlocks.

🔗 Combine with pg\_locks and pg\_blocking\_pids() for a full lock wait graph.

## **5️⃣ Measure Transaction Durations**

Open transactions that are not closed promptly can interfere with vacuum operations and contribute to table bloat.

echo "Processes ordered by current txn\_duration"  
psql -c "  
SELECT  
 datname,  
 pid,  
 application\_name,  
 state,  
 query,  
 now() - xact\_start AS txn\_duration  
FROM pg\_stat\_activity  
ORDER BY txn\_duration DESC;"

Processes ordered by current txn\_duration  
 datname | pid | application\_name | state | query | txn\_duration  
----------+------+------------------+--------+--------------------------------------+-----------------  
 | 1770 | | | |  
 | 1766 | | | |  
 | 1768 | | | |  
 | 1769 | | | |  
 | 1765 | | | |  
 postgres | 1824 | psql | active | INSERT INTO big\_table (data) +| 00:00:07.472628  
 | | | | SELECT repeat('x', 1000) +|  
 | | | | FROM generate\_series(1, 1000000); |  
 postgres | 1968 | psql | active | +| 00:00:00  
 | | | | SELECT +|  
 | | | | datname, +|  
 | | | | pid, +|  
 | | | | application\_name, +|  
 | | | | state, +|  
 | | | | query, +|  
 | | | | now() - xact\_start AS txn\_duration+|  
 | | | | FROM pg\_stat\_activity +|  
 | | | | ORDER BY txn\_duration DESC; |  
(7 rows)  
  
[postgres@ip-172-31-89-173 ~]$

### **🔍 Explanation:**

* Calculates how long each transaction has been active.
* Identifies applications and users associated with those transactions.

✅ ****Use Case:****  
Detect “idle in transaction” problems and prevent scenarios where autovacuum skips cleanup because of lingering transactions.

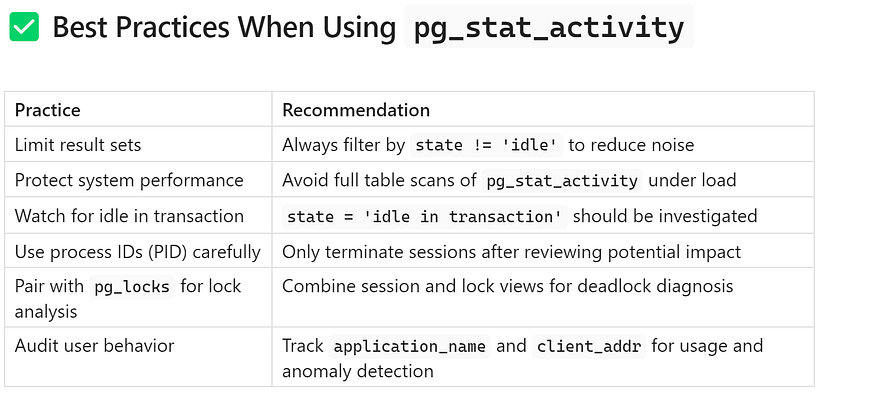
🧠 *Regularly review long transaction durations to ensure application logic commits or rolls back properly.*

## **🧪 Sample Automation Integration**

Integrate these queries into your system monitoring pipeline:

* ****Cron-based reporting:****
* Run and log every 15 minutes.
* Email alerts if query runtime or txn duration exceed thresholds.
* ****Prometheus & Grafana Dashboards:****
* Use PostgreSQL exporters with queries tied to visual thresholds.
* ****Custom admin views in pgAdmin or DataGrip:****
* Embed queries into user-defined dashboards.

Press enter or click to view image in full size



## **🏁 Conclusion**

Monitoring PostgreSQL in real time is essential for maintaining performance, ensuring stability, and responding proactively to issues. The pg\_stat\_activity view is a built-in observability tool that provides immediate insight into session-level behavior, enabling you to:

* 🎯 Detect problematic queries before they disrupt service
* 🔒 Identify blocking and waiting sessions during lock contention
* ⏱ Pinpoint long transactions that delay cleanup or introduce risk
* 👨‍💻 Trace user activity across client applications

By incorporating the diagnostic queries shown above into your ****daily monitoring practices****, ****automated alerts****, or ****real-time dashboards****, you equip yourself with a reliable framework to manage PostgreSQL health at scale.

📌 Start small with manual checks, scale up with automation, and integrate deeply with your team’s operational workflows.

## **🛠️ Advanced PostgreSQL Monitoring: Terminating Sessions, Logging Insights, and Performance Simulations**

PostgreSQL is an enterprise-class open-source database trusted by thousands of organizations to power transactional workloads. To ensure smooth operations in high-concurrency and high-throughput environments, it’s crucial for DBAs and engineers to proactively monitor database activity, manage long-running queries, and configure systems for observability.

This article focuses on ****actionable monitoring techniques**** beyond passive observation — such as ****terminating runaway sessions****, ****enabling full logging****, and ****generating test datasets for benchmarking****. These are best practices every PostgreSQL administrator should be familiar with.

## **🔧 Managing Problematic Queries in PostgreSQL**

While monitoring tools like pg\_stat\_activity provide visibility into what’s happening, you sometimes need to ****take corrective action****—especially when queries block others, run longer than expected, or consume excessive resources.

## **📌 How to Terminate a Session**

You can terminate a backend session safely using the pg\_terminate\_backend() function.

echo "Terminate the idle transaction. Replace <pid> with the PID of the process to terminate."  
psql -c "SELECT pg\_terminate\_backend(1824);"

⚠ Replace <pid> with the actual process ID of the session, which you can get from pg\_stat\_activity.

psql -c "SELECT pg\_terminate\_backend(1824);"  
Terminate the idle transaction. Replace <pid> with the PID of the process to terminate.  
 pg\_terminate\_backend  
----------------------  
 t  
(1 row)  
  
[postgres@ip-172-31-89-173 ~]$

postgres=# INSERT INTO big\_table (data)  
SELECT repeat('x', 1000)  
FROM generate\_series(1, 1000000);  
INSERT 0 1000000  
postgres=#  
postgres=#  
postgres=# select count(\*) big\_table;  
FATAL: terminating connection due to administrator command  
server closed the connection unexpectedly  
 This probably means the server terminated abnormally  
 before or while processing the request.  
The connection to the server was lost. Attempting reset: Succeeded.  
postgres=#  
postgres=#

## **🔍 When to Use It:**

* A query is stuck in a join, loop, or cartesian explosion
* An application holds a lock on a critical table and won’t release it
* Sessions are idle in transaction, preventing autovacuum from running
* A rogue analytics dashboard launches resource-intensive queries

## **✅ Best Practices:**

* Always inspect the query and session metadata first (e.g., user, query text, backend start time)
* If possible, notify the user or log the termination for auditing
* Avoid terminating autovacuum workers unless absolutely necessary

## **🖥️ Configuring PostgreSQL for Deeper Monitoring and Logging**

A default PostgreSQL installation does not include verbose logs or detailed query tracking. As part of ****observability hardening****, DBAs should configure the following settings:

## **📝 Enable Full Logging for Session Context**

PostgreSQL logs can be customized using log\_line\_prefix to add context like timestamps, users, client IPs, and application names. This is useful for ****debugging, auditing, and post-incident forensics****.

psql -c "ALTER SYSTEM SET log\_line\_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h';"

****Log Breakdown:****

* %t = Timestamp
* %p = Process ID
* %u = User
* %d = Database
* %a = Application name
* %h = Client hostname or IP

💡 After applying this change, reload the configuration with SELECT pg\_reload\_conf(); or restart the PostgreSQL service.

## **✅ Benefits:**

* Associate slow queries with specific users or applications
* Identify IP addresses generating connection floods
* Provide context-rich logs for security and compliance

## **📊 Enable**pg\_stat\_statements**for Historical Query Analytics**

The pg\_stat\_activity view provides real-time insight, but it does not store ****historical query statistics****. For that, you can enable the pg\_stat\_statements extension.

### **🔌 Installation & Setup:**

psql -c "ALTER SYSTEM SET shared\_preload\_libraries = 'pg\_stat\_statements';"  
sudo systemctl restart postgresql-17  
psql -c "CREATE EXTENSION pg\_stat\_statements;"

[postgres@ip-172-31-89-173 ~]$ psql -c "ALTER SYSTEM SET shared\_preload\_libraries = 'pg\_stat\_statements';"  
ALTER SYSTEM  
[postgres@ip-172-31-89-173 ~]$  
[postgres@ip-172-31-89-173 ~]$ sudo systemctl restart postgresql-17  
[postgres@ip-172-31-89-173 ~]$  
[postgres@ip-172-31-89-173 ~]$ psql -c "CREATE EXTENSION pg\_stat\_statements;"  
CREATE EXTENSION  
[postgres@ip-172-31-89-173 ~]$

### **🔍 Query Example:**

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

query | calls | total\_exec\_time | mean\_exec\_time  
------------------------------------------------------------------+-------+-------------------+---------------------  
 SELECT COUNT(\*) FROM big\_table +| 1 | 17092.417026 | 17092.417026  
 WHERE data LIKE $1 | | |  
 SELECT COUNT(\*) FROM big\_table | 1 | 131.835335 | 131.835335  
 CREATE EXTENSION pg\_stat\_statements | 1 | 14.776395 | 14.776395  
 SELECT query, calls, total\_exec\_time, mean\_exec\_time +| 12 | 4.253539000000001 | 0.35446158333333333  
 FROM pg\_stat\_statements +| | |  
 ORDER BY total\_exec\_time DESC +| | |  
 LIMIT $1 | | |  
 ALTER SYSTEM SET shared\_preload\_libraries = 'pg\_stat\_statements' | 1 | 3.04161 | 3.04161  
(5 rows)  
  
postgres=#

## **✅ Use Cases:**

* Identify ****frequently executed queries****
* Detect ****queries with the highest total or mean execution time****
* Perform ****query normalization and optimization****
* Understand ****query patterns across multiple users or applications****

🧠 Best Practice: Periodically reset statistics during maintenance windows using SELECT pg\_stat\_statements\_reset(); to keep the view manageable.

## **🧪 Test Dataset for Performance Simulation**

In development or testing environments, you’ll often need to simulate real-world data volumes to validate indexing, tuning, or application behavior under load.

## **🧱 Generate 1 Million Rows Using**generate\_series()

PostgreSQL’s built-in generate\_series() function makes it easy to create synthetic datasets.

psql -h localhost -U pguser -d pgdb \  
 -c "CREATE TABLE products AS SELECT \* FROM generate\_series(1, 1000000);"

## **✅ Purpose:**

* Simulate ****large datasets**** for benchmarking
* Test ****index creation**** and performance
* Analyze ****query planner behavior****
* Validate ****data migration tooling****

## **📝 Tip:**

To better mimic real-world schemas, wrap generate\_series() with additional columns, such as timestamps, status fields, or JSON payloads.

## **📈 Simulating Read and Write Workloads**

Testing how PostgreSQL behaves under load is a critical part of performance validation. Once you’ve created test data, perform the following operations:

## **🔍 Count Rows (Read Test)**

psql -h localhost -U pguser -d pgdb \  
 -c "SELECT COUNT(\*) FROM products;"

## **✅ Use Case:**

* Monitor how PostgreSQL utilizes indexes or sequential scans
* Validate caching effectiveness (repeated queries)
* Observe I/O and CPU behavior during reporting scenarios

## **➕ Insert New Rows (Write Test)**

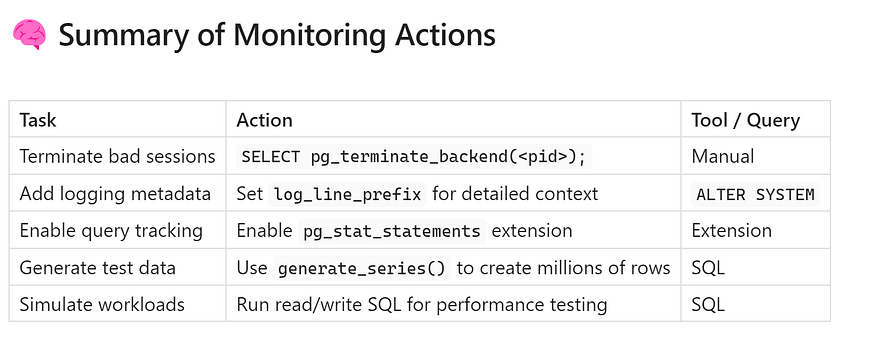
psql -h localhost -U pguser -d pgdb \  
 -c "INSERT INTO products SELECT \* FROM generate\_series(1000001, 1200001);"

## **✅ Use Case:**

* Stress test write throughput
* Observe behavior of WAL (Write-Ahead Logging)
* Validate performance of partitioned tables or bulk ingestion patterns

📘 Wrap in a transaction if needed and use EXPLAIN ANALYZE for deeper insight into execution plans.

Press enter or click to view image in full size



## **🏁 Conclusion**

PostgreSQL monitoring isn’t just about dashboards — it’s about having the ability to ****act decisively****, ****log intelligently****, and ****simulate workloads**** that reflect production complexity. By incorporating the techniques shared in this guide, you can:

* 🛠 Terminate sessions that threaten database stability
* 📋 Improve log clarity and postmortem analysis
* 📊 Gain visibility into normalized, historical query patterns
* 🔬 Perform realistic performance tests and tuning exercises

These are ****core competencies**** for PostgreSQL DBAs, Site Reliability Engineers, and performance-conscious developers.

Start applying these monitoring techniques today to proactively manage your PostgreSQL workloads with precision and confidence.

## **🔍 Putting It All Together: Real-Time PostgreSQL Monitoring with**pg\_stat\_activity

PostgreSQL is a powerful open-source relational database that supports a wide range of workloads. One of its most valuable features for operational monitoring is the pg\_stat\_activity view—a real-time window into what is happening inside your database. When combined with diagnostic scripts and structured practices, it becomes an essential part of a DBA’s performance and stability toolkit.

This section summarizes the core benefits and operational outcomes of using pg\_stat\_activity and related utilities for efficient database health checks.

## **🧰 Why It Matters: End-to-End Monitoring Workflow**

With the right set of queries and scripts, PostgreSQL administrators and performance engineers can gain actionable insights and maintain high database uptime. Specifically, the capabilities outlined allow you to:

## **✅ Monitor Real-Time SQL Activity**

You can continuously observe how your database is being used by tracking active and idle sessions. This visibility allows administrators to confirm whether PostgreSQL is handling expected workloads correctly or if there are anomalies requiring further investigation.

## **✅ Investigate Slow or Blocked Queries**

Long-running or blocked queries can severely affect throughput and user experience. Having visibility into these issues lets you drill down into the cause — whether it’s inefficient SQL, missing indexes, or session-level locks — and enables quicker remediation.

## **✅ Identify Sessions to Terminate Safely**

Not all sessions are equal. Some may be idle in transaction, consuming locks or preventing autovacuum from cleaning up dead tuples. Others may be generating excessive load or holding up business-critical processes. With sufficient monitoring data, problematic sessions can be identified and safely terminated without impacting the broader system.

## **✅ Simulate Load and Test Performance Tuning**

Query performance and database throughput should not only be observed in production but also validated in staging or dev environments. By simulating large datasets and transactional behavior, teams can test indexing strategies, validate query plans, and ensure tuning changes are effective before rolling them out.

## **✅ Build Automation Around PostgreSQL Health Checks**

Once reliable patterns are identified, they can be automated. Scheduled health check scripts, alerting mechanisms, and dashboards can help detect anomalies earlier — freeing up engineers from constant manual monitoring and reducing time-to-resolution for incidents.

## **🎯 Final Thoughts**

The pg\_stat\_activity view in PostgreSQL remains one of the most ****underutilized but powerful tools**** available to DBAs.

## **✅ Real-Time + Historical = Complete Observability**

By combining pg\_stat\_activity with tools like pg\_stat\_statements and enhanced logging, teams can move beyond ad-hoc monitoring and achieve a full-stack observability model. This allows for both real-time diagnostics and historical performance analysis.

## **✅ Proactive Monitoring Prevents Downtime**

Waiting until a performance issue causes user impact is already too late. Teams that proactively monitor query behavior — before it becomes a bottleneck — are better equipped to maintain application availability, user satisfaction, and business continuity.