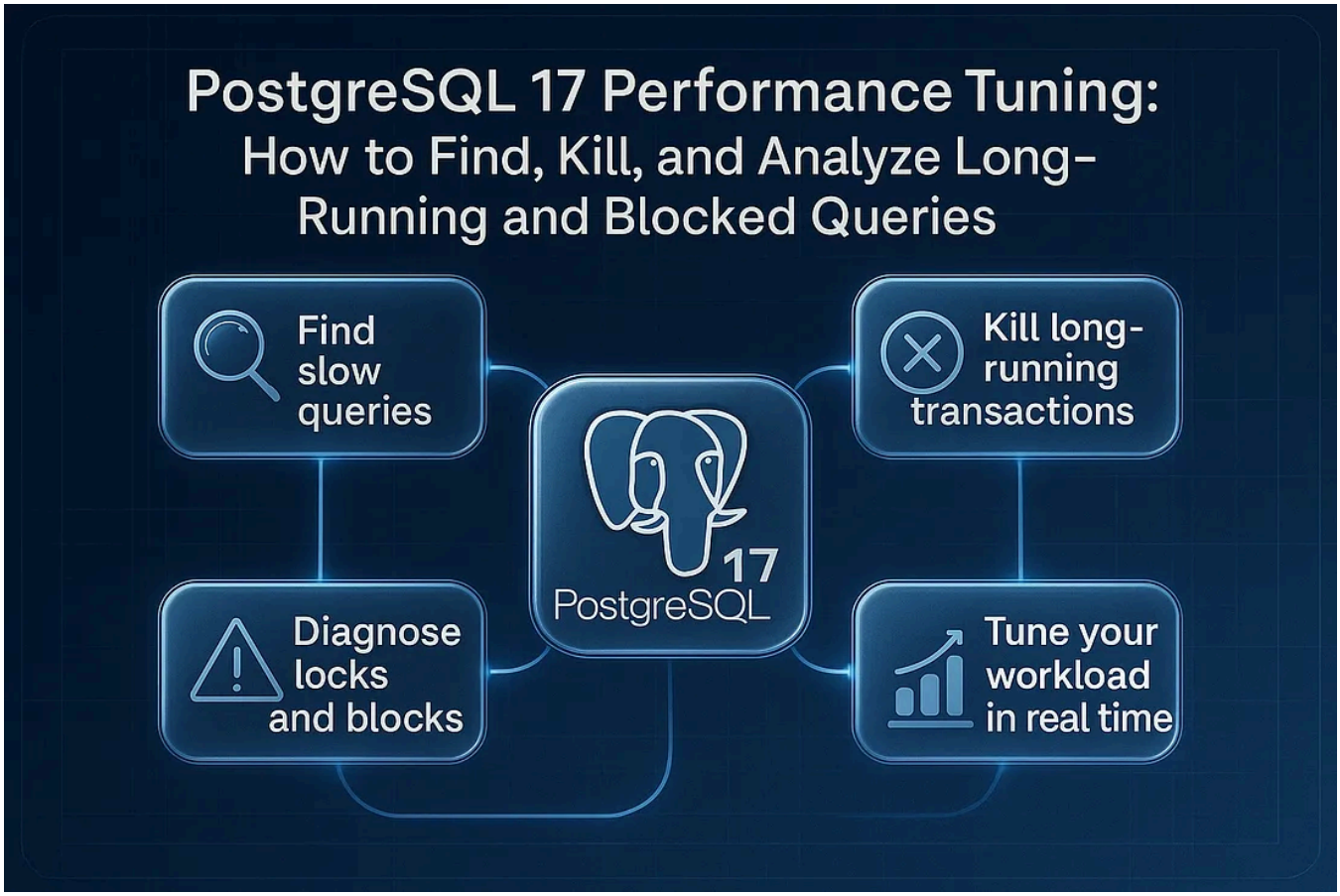
PostgreSQL 17 Performance Tuning: How to Find, Kill, and Analyze Long-Running and Blocked Queries

19 min read · Jun 14, 2025

Jeyaram Ayyalusamy Following



Keeping your PostgreSQL database fast, efficient, and healthy isn't magic — it's all about **proactive monitoring**.

PostgreSQL provides a rich set of internal system views that act like windows into the database engine. If you're managing production workloads, two of the most powerful views you should get comfortable with are:

- `pg_stat_activity`
- `pg_locks`

Together, these views can help you:

✅ Identify slow or stuck queries
✅ Terminate runaway sessions
✅ Understand locking behavior
✅ Tune performance bottlenecks in real time

Let's break this down.

🔍 What is `pg_stat_activity`?

`pg_stat_activity` shows **current activity in the PostgreSQL database.** It reveals what's running, who's running it, from where, and how long it's been running. It's your go-to view for live session diagnostics.

Sample Query:

```
SELECT pid, usename, application_name, state, query, wait_event_type, wait_ever
FROM pg_stat_activity
WHERE state != 'idle'
ORDER BY query_start;
```

Key columns:

- `pid` : Process ID of the backend
- `usename` : Connected user

- `state` : e.g., `active` , `idle in transaction` , etc.
- `query` : The actual SQL text being executed
- `wait_event_type` / `wait_event` : Useful to detect what it's waiting on (e.g., I/O, lock)

Why it matters:
This view tells you **who is doing what, for how long, and where the database is waiting**. For instance, spotting a query running for 25 minutes? That's a likely candidate for tuning or termination.

🛠 Killing a Long-Running Query

Sometimes you need to take action.

```
SELECT pg_terminate_backend(<pid>);
```

Use with care: this forcefully ends the session. A safer option for idle-in-transaction sessions is:

```
SELECT pg_cancel_backend(<pid>);
```

🔒 What is `pg_locks` ?

`pg_locks` provides visibility into all **row-level, relation-level, and transaction-level locks**. It is invaluable when you're facing **blocking issues** or **deadlocks**.

Sample Query to View Blocking:

```
SELECT blocked.pid AS blocked_pid,
       blocked.query AS blocked_query,
       blocking.pid AS blocking_pid,
       blocking.query AS blocking_query
FROM pg_locks blocked_locks
JOIN pg_stat_activity blocked ON blocked_locks.pid = blocked.pid
JOIN pg_locks blocking_locks ON blocking_locks.locktype = blocked_locks.locktyp
    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.transactionid IS NOT DISTINCT FROM blocked_locks.transac
    AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
    AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
    AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
    AND blocking_locks.pid != blocked_locks.pid
JOIN pg_stat_activity blocking ON blocking_locks.pid = blocking.pid
WHERE NOT blocked_locks.granted;
```

Use case:
If users are complaining that their transactions are “stuck,” this view shows **who is blocking whom**, letting you trace and resolve the issue.

⚙️ Putting It All Together

When performance issues strike, running a quick check on these views can save hours of debugging. Imagine this:

1. Check `pg_stat_activity` to find long-running queries
2. Use `pg_locks` to detect if they’re being blocked
3. Resolve the blocker using `pg_cancel_backend()` or `pg_terminate_backend()`

These are **essential DBA moves** that separate reactive firefighting from strategic, high-impact database operations.

📘 Final Thoughts

PostgreSQL gives you the tools — you just need to know where to look. `pg_stat_activity` and `pg_locks` aren't just diagnostic tools; they are your **real-time observability dashboard** for PostgreSQL health and performance.

If you're building production-grade PostgreSQL systems, start by mastering these two views.

🔍 PostgreSQL Statistics Collector 101: The Heart of Database Observability

PostgreSQL isn't just a powerful database engine — it's also incredibly self-aware.

Behind the scenes, PostgreSQL continuously collects vital runtime metrics through its **built-in statistics collector subsystem**. This subsystem is what makes PostgreSQL's observability tools so powerful. Whether you're tuning performance, debugging locks, or monitoring table usage, this is the engine doing the heavy lifting.

Let's take a closer look at what the statistics collector does — and why every DBA should love it.

💡 What is the PostgreSQL Statistics Collector?

The **statistics collector** is an internal PostgreSQL process that **gathers and stores runtime information** about the database system. This includes:

- 🔄 **Query execution patterns**
- 📊 **Table and index access frequency**
- ⚙️ **Client connection and backend activity**
- 🧮 **Lock contention and wait events**

This isn't just about pretty dashboards — it's foundational for **query tuning, capacity planning, and performance troubleshooting.**

🧬 How It Works

The collector constantly receives updates from the server processes and stores them in shared memory. These statistics are then made available through a rich set of system views (think: virtual tables). You can query these views just like any table using SQL.

The statistics data is reset either manually (e.g., via `pg_stat_reset()`) or when the database restarts, so it's meant for **short- to medium-term analysis**, not long-term historical tracking (unless you integrate with tools like Prometheus or pgBadger).

📖 Key System Views You Should Know

Here are some of the most essential views provided by the statistics collector:

🔎 `pg_stat_activity` : Real-Time Query Monitoring

Shows all current connections and the queries being executed. Crucial for detecting long-running or blocked queries.

```
SELECT pid, usename, query, state, query_start FROM pg_stat_activity WHERE stat
```

🔒 `pg_locks` : Lock Tracking

Provides insights into what locks are held or waiting. Essential when diagnosing deadlocks or performance issues due to contention.

```
SELECT * FROM pg_locks WHERE NOT granted;
```

📊 `pg_stat_database` : Global Stats Per Database

Shows aggregate statistics at the database level — tuples fetched, committed, rolled back, blocks hit, and more.

```
SELECT datname, numbackends, xact_commit, blks_hit FROM pg_stat_database;
```

🧮 **Other Useful Views:**

- `pg_stat_user_tables` : Stats per user table (e.g., seq scans, inserts, updates)
- `pg_stat_user_indexes` : Index usage patterns (e.g., idx scans, idx fetches)
- `pg_statio_user_tables` : I/O patterns (heap reads, blocks read vs. hit)
- `pg_stat_bgwriter` : Background writer activity (checkpoints, buffers written)

These views give DBAs and developers an **x-ray view of what's really happening** inside their PostgreSQL database.

🔧 Why It Matters

Without the statistics collector:

- You'd be **guessing at performance issues**
- You couldn't **optimize queries based on real usage**
- You'd have no visibility into **locks, deadlocks, or wait events**
- Scaling decisions would be **blind bets**

With it, however, you get a **powerful toolkit** to guide indexing strategies, rewrite inefficient queries, manage connection pooling, and stay ahead of potential bottlenecks.

📌 Final Thoughts

The PostgreSQL statistics collector is the unsung hero behind every tuned query and every well-optimized system. By exposing real-time and cumulative performance data, it allows DBAs, developers, and SREs to make **data-driven decisions** instead of relying on gut feeling or guesswork.

So the next time you're chasing down a slow query or planning to scale your infrastructure, start by querying the stats. PostgreSQL already has the answers — **you just need to ask the right views.**

🎯 Monitoring Queries with `pg_stat_activity`

When you're managing a PostgreSQL database — especially in a production environment — **real-time visibility into what queries are running** is absolutely critical.

That's where `pg_stat_activity` comes in.

This powerful system view gives you a **live snapshot of all active connections** to your PostgreSQL instance. Whether you're investigating a performance slowdown, tracking suspicious behavior, or simply monitoring user activity, `pg_stat_activity` is your go-to tool.

📸 What is `pg_stat_activity`?

`pg_stat_activity` is a **system catalog view** that shows information about:

- All current client connections
- The state of each backend (e.g., active, idle)
- The SQL statement being executed
- When the query started
- Who issued it and from where

Think of it as **PostgreSQL's process list** — a real-time dashboard of database activity at the session level.

📄 View All Running Queries

To get the full picture of what's happening in your database, this is the simplest and most informative starting point:

```
SELECT * FROM pg_stat_activity;
```

This query returns details such as:

- `pid` : Process ID of the session
- `usename` : Username of the connected client
- `datname` : Database name
- `client_addr` : IP address of the client
- `query` : SQL being executed
- `state` : `active` , `idle` , `idle in transaction` , etc.
- `query_start` : When the current query started

💡 **Tip:** Focus on queries with long durations and those in `active` or `idle in transaction` states—they often indicate performance issues or open transactions that need attention.

🔢 Count Total Active Connections

Want to see how many users are currently connected? Use this simple count query:

```
SELECT COUNT(*) AS total_conns FROM pg_stat_activity;
```

This is particularly useful for:

- **Monitoring load** on your PostgreSQL instance
- **Enforcing connection limits**
- **Detecting spikes** during traffic surges or abnormal usage

You can also compare this number to `max_connections` in your `postgresql.conf` to ensure you're not close to hitting your connection ceiling.

🌐 Count Connections from a Specific IP

Need to check if one particular client is overloading the server or making too many connections? Filter by IP:

```
SELECT COUNT(*) FROM pg_stat_activity WHERE client_addr = '192.168.1.10';
```

This query helps:

- Detect **abuse or anomalies** from a single source
- Enforce **client connection policies**
- Debug issues tied to specific **application servers**

You can even extend this query to group and rank IPs by the number of connections:

```
SELECT client_addr, COUNT(*) AS conn_count
FROM pg_stat_activity
```

```
GROUP BY client_addr
ORDER BY conn_count DESC;
```

🚨 Common Use Cases for `pg_stat_activity`

- 🔍 **Identifying slow or stuck queries**
- 🧯 **Killing long-running sessions** via `pg_terminate_backend(pid)`
- 🧭 **Tracing application behavior** during incidents
- 📊 **Monitoring resource consumption per user or application**

🧠 Final Thoughts

`pg_stat_activity` is more than just a monitoring tool—it's a **critical observability asset** for PostgreSQL DBAs and developers. With a few simple SQL queries, you can diagnose bottlenecks, monitor live traffic, and take immediate action to ensure the health and performance of your database.

Next time something feels “off” with your database, start with `pg_stat_activity`. It often tells you exactly what you need to know.

🚩 Find Long-Running Queries in PostgreSQL

When your PostgreSQL database starts feeling sluggish, one of the first culprits to investigate is **long-running queries**.

These queries can:

- Hold locks for too long
- Block other transactions
- Consume CPU and memory

- Slow down overall system performance

The good news? PostgreSQL gives you the tools to **catch them in real time.**

🕵️ SQL to Catch Long-Running Queries

Here's a simple query that identifies any session where the query has been running for **more than 5 minutes**:

```
SELECT
  pid,
  usename,
  query_start,
  now() - query_start AS query_time,
  query,
  state,
  wait_event_type,
  wait_event
FROM pg_stat_activity
WHERE now() - query_start > interval '5 minutes';
```

This query leverages the `pg_stat_activity` view to show everything about sessions that may be hogging resources.

📌 What You're Seeing

Let's break down the important columns:

- **`pid`** : Process ID of the session. You'll use this with functions like `pg_terminate_backend()` to stop a query if needed.
- **`usename`** : The database user who is running the query.
- **`query_start`** : Timestamp when the query started.

- **`query_time`** : How long the query has been running (calculated as `now() - query_start`).
- **`query`** : The actual SQL text. This helps you understand what's causing the issue.
- **`state`** : Whether the session is `active`, `idle`, or `idle in transaction`.
- **`wait_event_type`** / **`wait_event`** : Shows if the query is waiting on a lock, I/O, or another resource—great for diagnosing why it's stuck.

💡 **Pro tip:** If `wait_event` indicates something like `Lock`, the session might be **blocked by another transaction.**

📊 See Database-Wide Activity Snapshot

Want a **big-picture overview** of who's connected to your PostgreSQL database, what databases they're using, and what state their sessions are in?

This query provides a grouped summary:

```
SELECT
  client_addr,
  usename,
  datname,
  state,
  COUNT(*)
FROM pg_stat_activity
GROUP BY client_addr, usename, datname, state
ORDER BY COUNT(*) DESC;
```

📋 What This Shows You:

- **`client_addr`** : The IP address of the connecting client
- **`usename`** : The database user
- **`datname`** : The name of the database
- **`state`** : Active, idle, idle in transaction, etc.

- `COUNT(*)` : Number of sessions in this category

This view is ideal for:

- Monitoring **connection patterns**
- Spotting **overloaded applications**
- Understanding **how many clients are connected and in what state**

For example, if you see a high number of `idle in transaction` connections from a specific IP, that could indicate a **poorly managed connection pool** or an app not committing its transactions.

🧠 Wrapping Up

PostgreSQL provides deep introspection tools, and with just a few queries, you can:

- Find and fix long-running queries
- Monitor connection load and patterns
- Pinpoint performance bottlenecks

By mastering `pg_stat_activity` , you gain the visibility needed to maintain a **healthy, performant database**—without guesswork.

Stay tuned for the next post, where we'll dive into **pg_stat_statements** for historical query performance tracking!

🔒 Detect Locks and Blocked Queries in PostgreSQL Like a Pro

When a PostgreSQL database starts slowing down and **nothing obvious appears to be wrong**, there's a good chance the issue is related to **locks.**

Locks are essential for maintaining **data consistency** and **transactional integrity**, but when mismanaged, they can silently wreak havoc on your system — causing

delays, timeouts, and user frustration.

In this section, we'll walk through how to detect and trace locks using `pg_locks` and identify blocked queries using `pg_stat_activity`.

🔍 Detecting Locks with `pg_locks`

The PostgreSQL system view `pg_locks` gives you visibility into all types of locks held or awaited by sessions in the database. These include:

- Row-level locks
- Table-level locks
- Transaction-level locks

To view all currently held and waiting locks, run:

```
SELECT * FROM pg_locks;
```

This gives you low-level details, including:

- **`locktype`** : The type of lock (relation, transaction, etc.)
- **`mode`** : The kind of lock (e.g., RowExclusiveLock, AccessShareLock)
- **`granted`** : Whether the lock is currently granted (`true`) or the process is still waiting for it (`false`)
- **`pid`** : The process ID holding or waiting on the lock

📑 Find Locks with Table Names

Raw lock data is hard to interpret without knowing which **tables** are involved. Use this query to join `pg_locks` with `pg_class` and `pg_stat_activity` to get the **table**

names and related queries:

```
SELECT
  relname AS relation_name,
  query,
  pg_locks.*
FROM pg_locks
JOIN pg_class ON pg_locks.relation = pg_class.oid
JOIN pg_stat_activity ON pg_locks.pid = pg_stat_activity.pid;
```

This version gives you:

- ✅ Table name (`relname`)
- ✅ The **SQL query** responsible for holding or waiting on the lock
- ✅ All lock metadata from `pg_locks`

💡 This is especially useful when diagnosing **contention on specific tables** — such as when an `UPDATE` or `DELETE` operation is blocking other transactions.

🚦 Identify Blocked Queries and Their Blockers

Want to find out which queries are blocked — and **exactly who's blocking them**? This powerful query shows both sides of the problem:

```
SELECT
  activity.pid,
  activity.usename,
  activity.query,
  blocking.pid AS blocking_id,
  blocking.query AS blocking_query
FROM pg_stat_activity AS activity
JOIN pg_stat_activity AS blocking
  ON blocking.pid = ANY(pg_blocking_pids(activity.pid));
```

📌 What This Tells You:

- `activity.pid` : The session that is currently **blocked**
- `activity.query` : The SQL query that is waiting
- `blocking.pid` : The session that is **causing the block**
- `blocking.query` : The SQL query currently holding the lock

This is one of the most **actionable diagnostic tools** in PostgreSQL.

You can trace lock chains and:

- Notify or terminate the blocking session (`pg_terminate_backend(pid)`)
- Optimize long-running write operations
- Rethink transaction scopes in your application

🧠 Why Locks Matter (And Why You Should Watch Them)

- Locks are necessary, but when **left uncommitted** or **poorly indexed**, they block critical operations.
- A single `idle in transaction` session can **freeze a table** for everyone else.
- The longer a lock waits, the more **backpressure** it puts on your entire system.

By actively monitoring `pg_locks` and blocked queries, you'll keep your PostgreSQL environment responsive, reliable, and user-friendly.

✅ Final Thoughts

Locks don't have to be mysterious. With the power of `pg_locks` , `pg_stat_activity` , and smart queries, you can:

- Detect contention before users feel it
- Trace the root cause of blocking chains

- Take informed action to free up system resources

The next time you're staring at a sluggish dashboard or stuck migration, check your locks. The problem — and the fix — might already be waiting for you in `pg_stat_activity`.

⚠️ Cancel or Kill Long-Running Queries in PostgreSQL

In any PostgreSQL system — especially those in production — **long-running queries** can become a major performance bottleneck. They can:

- Hold locks for extended periods
- Block other queries
- Consume memory and CPU
- Delay user-facing responses or batch jobs

When that happens, you may need to intervene and **stop the query manually**. Fortunately, PostgreSQL provides two powerful tools to help: `pg_cancel_backend()` and `pg_terminate_backend()`.

But the choice between the two matters — let's break them down.

✅ Graceful Cancel (Preferred Method)

If you want to **stop the query but not disconnect the user session**, use the `pg_cancel_backend()` function. This method **requests the server to cancel the currently running query**, allowing the client connection to stay open.

```
SELECT pg_cancel_backend(pid)
FROM pg_stat_activity
WHERE query LIKE '%your_problem_query%';
```

🔍 **What It Does:**

- **Attempts a clean interruption** of the running query
- Leaves the session **alive**, so the user or application can continue sending new queries
- Ideal for **interactive sessions**, dashboards, or apps that you don't want to fully disconnect

🧠 **Pro Tip:** This is your **first line of defense.** Always try canceling before resorting to a forceful termination.

🛑 Force Kill (Use with Caution)

If canceling the query doesn't work — or if the session is stuck in a bad state — you can escalate with `pg_terminate_backend()`. This **immediately terminates the entire backend process** handling the query.

```
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE query LIKE '%your_problem_query%';
```

⚠️ **What It Does:**

- **Kills both the query and the database connection**
- Rolls back any open transactions associated with the session
- Can be disruptive if the session is part of a larger transaction block or a connection pool

💥 **Warning:** Use this method only when you're certain the session **must be terminated** — such as in cases of deadlocks, runaway queries, or system-threatening behavior.

🧠 When Should You Cancel vs. Kill?

Situation	Recommended Action
Long query but responsive session	✅ `pg_cancel_backend()`
Idle-in-transaction holding a lock	🚫 `pg_terminate_backend()`
App is retrying a stuck query	✅ Start with cancel
Session is unresponsive or blocking many	🔥 Force kill with care
Zombie connection from an app crash	🔥 Kill required

✅ Final Thoughts

PostgreSQL gives you fine-grained control over managing queries — but with great power comes great responsibility. Understanding the **difference between canceling and killing** is key to maintaining stability without accidentally dropping active users or services.

Whenever possible:

- **Start with cancel**
- **Monitor lock wait events**
- **Only kill when absolutely necessary**

And always log the actions you take, especially in production environments.

🖥️ Use pgAdmin to Manage Queries — No SQL Needed

If you're managing PostgreSQL databases but don't feel comfortable typing SQL commands — or you just prefer a visual interface — **pgAdmin** is your best friend.

pgAdmin is the official graphical user interface (GUI) tool for PostgreSQL, and it makes monitoring and managing queries as easy as a few clicks. Whether you're a

beginner or a seasoned DBA looking for convenience, pgAdmin helps you avoid memorizing SQL commands for tasks like tracking queries, canceling problematic sessions, or terminating runaway connections.

Let's walk through how to use it step-by-step.

🚀 Step-by-Step: Monitor and Manage Queries Using pgAdmin

1️⃣ Open pgAdmin

Launch pgAdmin on your local machine or access it via the browser if it's hosted on a server. Log in with your credentials and connect to your PostgreSQL server instance.

2️⃣ Select Your Database

In the **left-hand tree panel**, expand the connection to your PostgreSQL server. Navigate to the database you want to monitor.

3️⃣ Go to: Dashboard → Activity → Sessions

Once inside your selected database:

- Click on the **Dashboard** tab (top panel).
- Select the **Activity** tab.
- Choose **Sessions** to view a list of all current database sessions and queries.

This panel displays:

- Connected users
- Running queries
- Session states (`active` , `idle` , `idle in transaction`)
- Backend process IDs (PIDs)
- Application names
- Client IPs
- Query durations

4️⃣ Easily Sort, Cancel, or Terminate Queries

From the Sessions panel, you can **interact directly with any running query**:

- ✅ **Sort by query duration** to spot long-running processes
- ❌ **Right-click → Cancel** to gracefully stop a query
- 🔥 **Right-click → Terminate** to forcefully kill a session (use with caution)

This is especially useful when:

- You notice a query blocking others
- A user left a session open and it's holding locks
- You want to free up system resources without writing SQL

💡 Why Use pgAdmin for Query Management?

- **No need to write or remember SQL commands**
- **Visual clarity** makes it easy to spot performance issues
- Ideal for **quick interventions** during incidents
- Great for **non-DBAs** who need to manage PostgreSQL occasionally

Whether you're managing a development environment or troubleshooting production, pgAdmin offers a **user-friendly alternative** to command-line tools like `psql`.

✅ Final Thoughts

pgAdmin puts powerful PostgreSQL monitoring tools right at your fingertips — no need to be a SQL wizard. With a few clicks, you can sort, cancel, and kill queries, giving you real-time control over your database sessions.

If you're just getting started with PostgreSQL or managing multiple databases, integrating pgAdmin into your workflow can save time and reduce errors.

📊 Key Columns in `pg_stat_activity` — What They Mean and Why They Matter

When you're monitoring your PostgreSQL database using `pg_stat_activity`, it's essential to understand the meaning of the key columns. These fields give you valuable insights into what's happening inside your database at any given moment.

Here's a breakdown of the most important columns you'll encounter:

Column	Description
`pid`	Process ID of the session (used to cancel or terminate it, if needed)
`usename`	The name of the user running the query
`datname`	The database that the session is connected to
`query_start`	Timestamp of when the current query began execution
`state`	The current state of the connection (e.g., `active`, `idle`, `idle in transaction`)
`wait_event`	What the backend is currently waiting on (e.g., lock, I/O, or timeout)

These columns are especially useful when diagnosing slow queries, connection bottlenecks, or blocked transactions.

🔎 Extra Monitoring Queries — Supercharge Your PostgreSQL Visibility

Once you're familiar with `pg_stat_activity`, you can start running targeted queries to get deeper insights. Here are some **practical monitoring queries** to help you stay ahead of issues:

✅ View All Active Connections

```
SELECT * FROM pg_stat_activity;
```

This is your general-purpose snapshot of all current sessions, including users, IPs, queries, and more. Use this as a starting point to understand overall activity.

🔢 Count Total Connections

```
SELECT COUNT(*) FROM pg_stat_activity;
```

This gives you a quick total of all current sessions. It's useful for:

- Monitoring system load
- Spotting connection pool issues
- Ensuring you're within limits of `max_connections`

🚦 Find Queries That Are Waiting

```
SELECT * FROM pg_stat_activity
WHERE wait_event IS NOT NULL
AND backend_type = 'client backend';
```

This filters out any backend processes that are **waiting on something** — typically due to locks or I/O delays. This is a powerful way to identify:

- Blocked queries
- Lock contention issues

- Queries stuck due to resource unavailability

🧠 **Tip:** Combine this with lock-tracing queries for deeper diagnostics.

⏱️ Find Queries Running for Over 1 Second

```
SELECT
  client_addr,
  usename,
  datname,
  now() - query_start AS query_age,
  state,
  query
FROM pg_stat_activity
WHERE (now() - query_start) > interval '1 second'
AND state <> 'idle'
ORDER BY query_start;
```

This helps you catch **long-running or suspiciously slow queries**, even if they aren't holding locks. It's great for spotting:

- Performance bottlenecks
- N+1 query problems
- Queries not using indexes effectively

📌 Focus on queries with high `query_age` and non-idle states for optimization opportunities.

🧠 Final Thoughts

Understanding and using `pg_stat_activity` effectively is a game-changer for PostgreSQL monitoring. By mastering its key columns and combining them with powerful query patterns, you can:

- Monitor active sessions in real time
- Detect and resolve performance issues faster
- Ensure your system stays responsive and stable

The beauty of PostgreSQL is that it offers **deep introspection tools** without needing any external add-ons. With just SQL and a bit of know-how, you're already in control.

🔁 Advanced Lock Chain Visualization in PostgreSQL

Unravel complex deadlocks like a pro

When you're dealing with PostgreSQL performance issues, sometimes it's not enough to just see which queries are blocked — you need to understand **how the blocking chain unfolds**. This is especially important in complex environments where **multiple sessions block each other in cascading sequences**, potentially leading to deadlocks.

That's where this **recursive lock chain visualization query** becomes your secret weapon. It gives you a **tree-like view** of who is blocking whom — and how deep the blockage goes.

🧠 Why You Need This

In real-world workloads:

- One user might be waiting on a lock held by another.
- That second user might be waiting on a third.
- Before you know it, you're in a **multi-level lock chain** or even a deadlock.

This recursive query helps you trace that chain **step-by-step**, making it easier to pinpoint the root blocker and clean up the jam.

🔍 What This Query Does

Here's what each part of the query is doing:

🔄 `WITH RECURSIVE l AS (...)`

This part prepares a list of **locks**, capturing key lock identifiers like `relation`, `page`, `transactionid`, etc., and wraps them into a structured `obj` row. This makes it easy to compare locks later.

🔗 `pairs AS (...)`

This builds **waiter → locker pairs**. It checks which sessions are **waiting** on the same lock object that is **already held** by another session. It filters out self-pairs and ensures only **granted vs. waiting** locks are linked.

🌳 `tree AS (...)`

This is the heart of the recursion. It:

- Starts with the **root blockers** — sessions that are **not being blocked** by anyone.
- Then recursively adds all sessions that are **waiting on these blockers**.
- Builds a path (`1.2.3`) and level (`lvl`) to form a visual tree structure.
- Prevents cycles using an `ARRAY` to track seen PIDs.

📤 Final Output: A Visual Blocking Chain

```
SELECT
  (clock_timestamp() - a.query_start)::interval(3) AS query_age,
  a.datname,
  tree.pid,
  a.usename,
  a.client_addr,
  lvl,
  REPEAT(' .', lvl) || ' ' || LEFT(REPLACE(query, E'\n', ' '), 80) AS query
FROM tree
JOIN pg_stat_activity a USING (pid)
ORDER BY path;
```

🗒️ What You'll See:

Column Description `query_age` How long the query has been running `datname` Database name `pid` Process ID of the session `usename` User running the query `client_addr` IP address of the client `lvl` Level in the blocking tree (0 = root blocker) `query` Truncated, formatted SQL query with indentation based on tree level

Indentation (`REPEAT(' .', lvl)`) visually illustrates which sessions are **blocking others** and how deep the chain goes.

🧰 When to Use This

Use this recursive query when:

- You suspect a **deadlock** or **multi-session blocking**
- A single long-running transaction is blocking **dozens of others**
- You need to **identify the root blocker** for resolution
- You want to **visualize query dependencies** during peak loads

It's incredibly useful in **OLTP systems**, batch data loads, or any environment with heavy write contention.

✅ Final Thoughts

This recursive query is like having **x-ray vision** into PostgreSQL's lock mechanics. While tools like `pg_locks` and `pg_stat_activity` give you the pieces, this view **connects the dots** and reveals the full lock chain in one elegant output.

Whether you're debugging a production incident or proactively tuning performance, mastering lock chain visualization sets you apart as a PostgreSQL expert.

🏁 Conclusion: Master Your PostgreSQL Like a Pro

Managing a high-performing PostgreSQL database isn't just about reacting to problems after they occur — **it's about staying ahead of them.**

The good news? PostgreSQL gives you everything you need to do just that.

✅ Real-Time Visibility, Right at Your Fingertips

PostgreSQL's introspection tools, like `pg_stat_activity` and `pg_locks`, provide **live insights into the heart of your database.** You don't need to guess which query is slow, which user is blocked, or where contention is happening—PostgreSQL tells you.

From query durations to session states and lock conflicts, these tools let you:

- **Track real-time activity**
- **Spot bottlenecks early**
- **Act before users notice a problem**

✅ Master `pg_stat_activity` and `pg_locks`

These two system views form the **core of any PostgreSQL monitoring strategy.**

- Use `pg_stat_activity` to find **long-running or stuck queries**, track user behavior, and monitor session health.
- Use `pg_locks` to **trace lock contention**, identify deadlocks, and map blocking chains with advanced queries.

Together, they give you **360° control over query behavior and system performance.**

✅ Go Beyond Queries: Build Dashboards and Alerts

Don't stop at manual monitoring — **automate your visibility:**

- Integrate with tools like **Grafana**, **Prometheus**, or **pgBadger** for visual dashboards
- Set up **alerting** when queries run too long, lock chains form, or connection limits are reached
- Use scheduled jobs or background workers to **log and audit activity patterns**

This shifts you from **reactive** DBA work to a **proactive performance guardian.**

🚀 Final Thought: Prevention is the New Cure

The best DBAs don't just fix problems — they **prevent them.**

By mastering PostgreSQL's built-in monitoring features, you'll gain:

- Faster diagnostics
- Smoother operations
- Happier end users
- Fewer 2 AM incident calls

PostgreSQL is more than a database engine — it's a platform that empowers you to run stable, high-performance systems. All you need to do is **tap into the visibility it already provides.**

👉 *If you found this guide helpful, follow me(**medium**) for more practical PostgreSQL tutorials, database architecture guides, and hands-on DBA content.*

🔗 Let's Connect!

If you enjoyed this post or would like to connect professionally, feel free to reach out to me on LinkedIn:
👉 Jeyaram Ayyalusamy

I regularly share content on **PostgreSQL, database administration, cloud technologies, and data engineering.** Always happy to connect, collaborate, and discuss ideas!

Postgresql Sql Oracle MySQL AWS

Following

Written by Jeyaram Ayyalusamy

62 followers · 2 following

Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

Responses (1)

Gvadakte

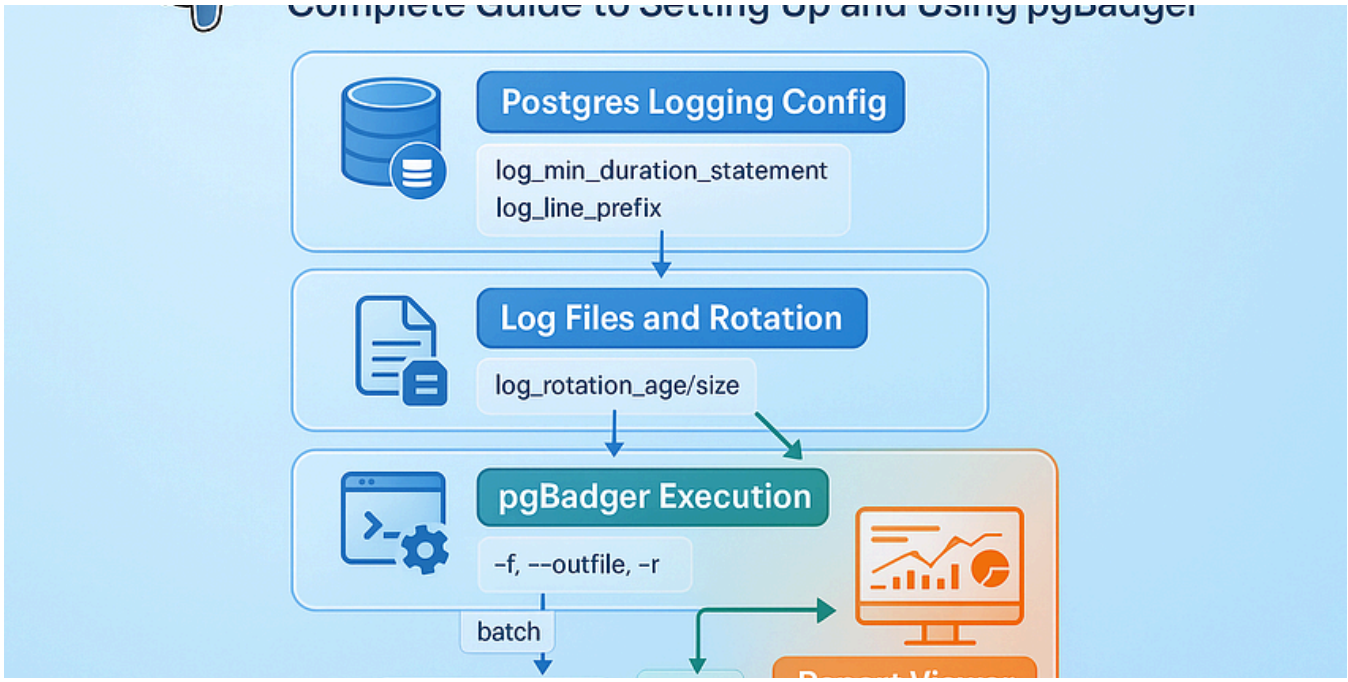
What are your thoughts?

Chigozie Damian Okali
2 days ago

Quite an elaborate and educative write up on PostgreSQL database management.

Reply

More from Jeyaram Ayyalusamy



Jeyaram Ayyalusamy

PostgreSQL 17 Log Analysis Made Easy: Complete Guide to Setting Up and Using pgBadger

Managing a production-grade PostgreSQL instance isn't just about keeping data safe—it's about monitoring performance, identifying...

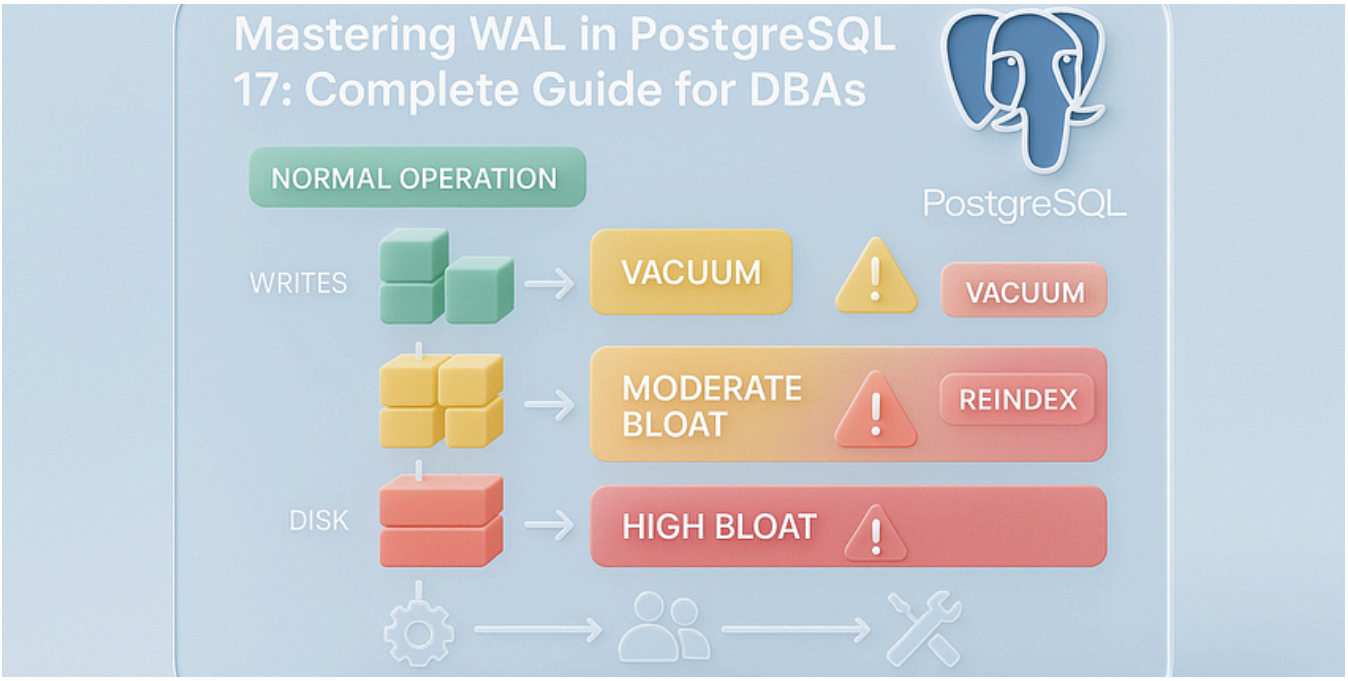
Jun 23 52

Jeyaram Ayyalusamy

PostgreSQL 17 Administration: Mastering Schemas, Databases, and Roles

PostgreSQL has evolved into one of the most feature-rich relational databases available today. With version 17, its administrative...

Jun 9 3



Jeyaram Ayyalusamy

Understanding and Managing Bloating in PostgreSQL: A Complete Guide for DBAs

PostgreSQL is a powerful, reliable, and feature-rich open-source relational database system. It's praised for its extensibility, ACID...

Jun 25 2

Jeyaram Ayyalusamy

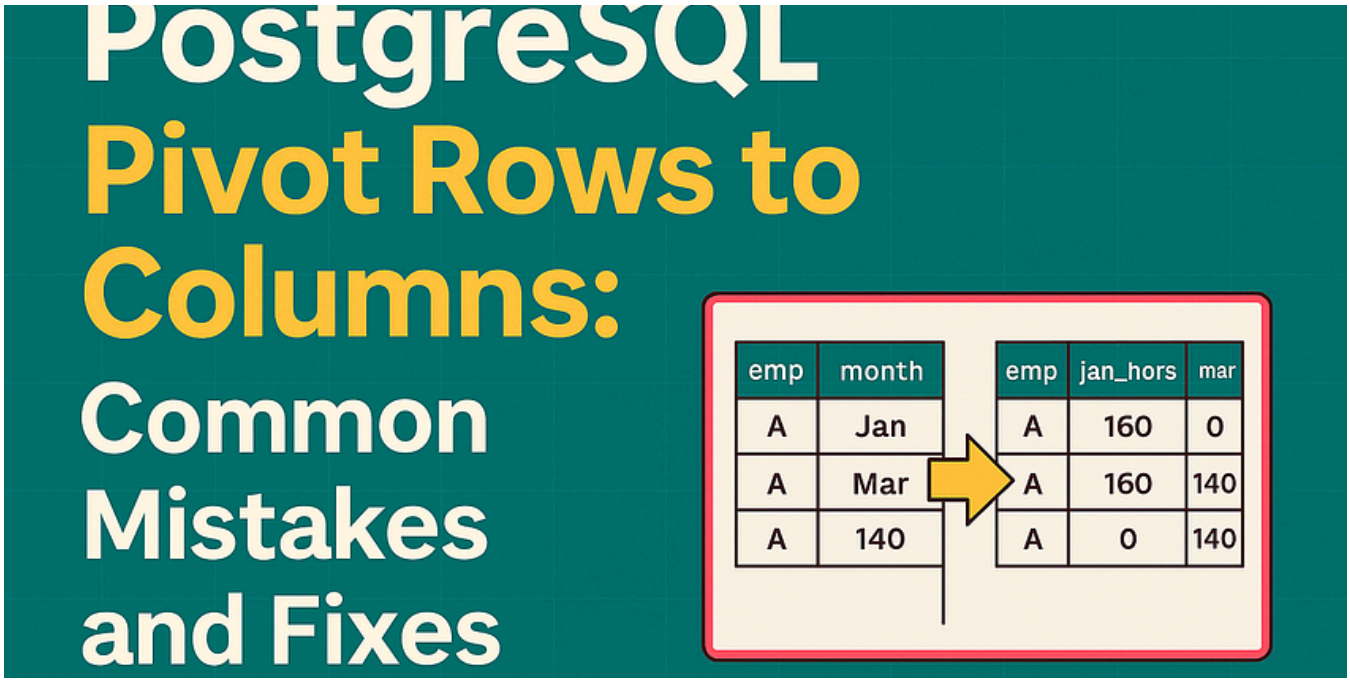
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

PostgreSQL 17 is the latest release of one of the world's most advanced open-source relational databases. If you're using a Red Hat-based...

Jun 3

See all from Jeyaram Ayyalusamy

Recommended from Medium



Ajaymaurya

PostgreSQL Pivot Rows to Columns: Common Mistakes and Fixes

When working with data in PostgreSQL, pivoting rows into columns can feel like a superpower—until something breaks. Whether you're...

Jun 27 15

In Level Up Coding by Daniel Craciun

Stop Using UUIDs in Your Database

How UUIDs Can Destroy SQL Database Performance

Jun 25 791 50

Mojtaba Azad

How We Migrated a 100M Row PostgreSQL Table With Zero Downtime (and Survived)

Here's how we pulled off a 100-million-row table migration in PostgreSQL without a single second of downtime, using only native tools.

Jun 7 56 2

ThreadSafe Diaries

PostgreSQL 18 Just Rewrote the Rulebook, Groundbreaking Features You Can’t Ignore

From parallel ingestion to native JSON table mapping, this release doesn’t just evolve Postgres, it future-proofs it.

Jun 27 477 4

In DevOps.dev by M Mahdi Ramadhan, M. Si

Handling PostgreSQL Connection Pooling

PostgreSQL: connection limit exceeded.

Jun 15 54

Oz

Understanding PostgreSQL Page (Block) Structure

PostgreSQL stores data on disk in fixed-size units called pages or blocks. In this post, we'll explore how PostgreSQL organizes data...

May 14 58 1