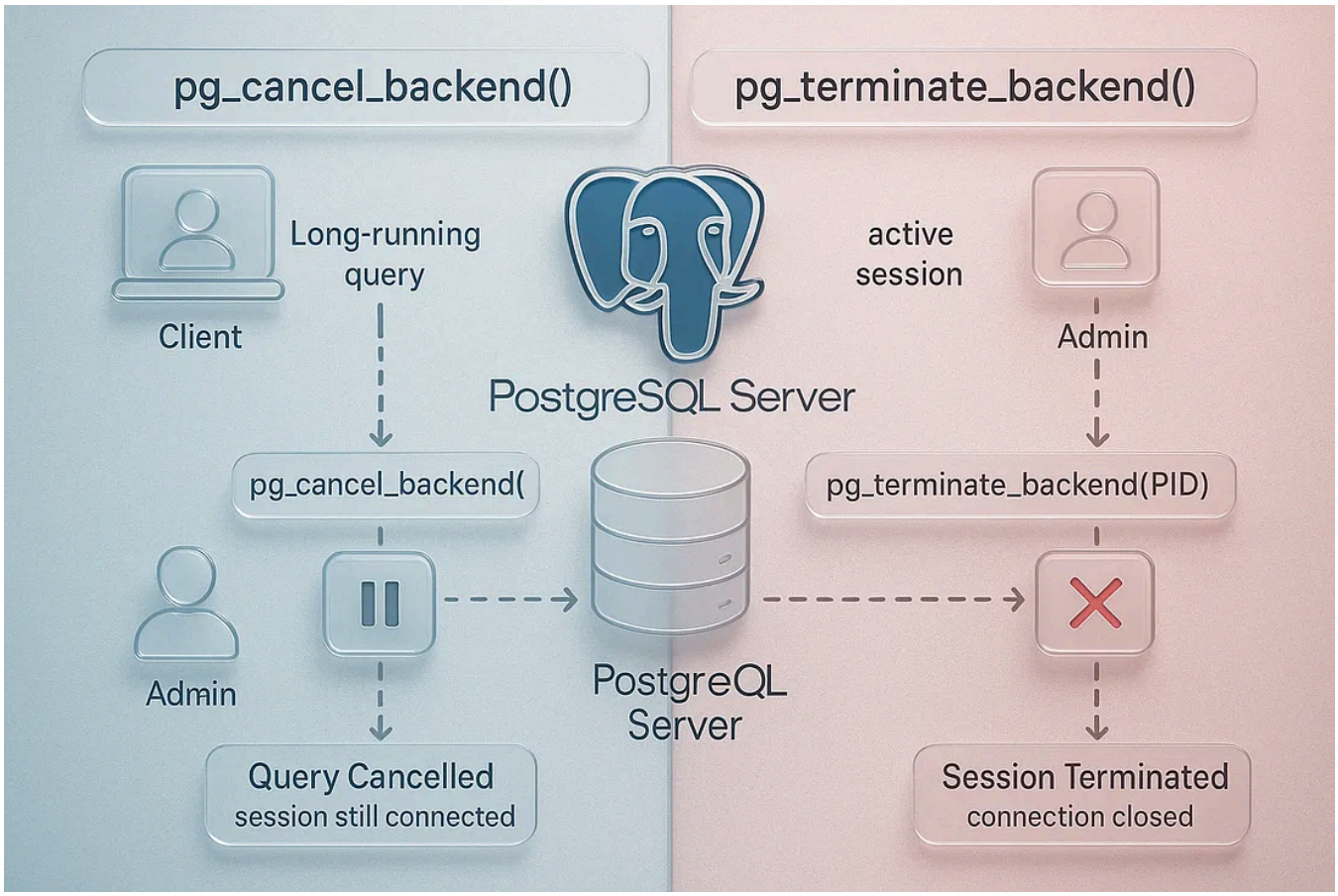
How to Safely Terminate PostgreSQL Sessions: `pg_cancel_backend()` vs `pg_terminate_backend()`

16 min read · Jun 18, 2025

Jeyaram Ayyalusamy Following



As a PostgreSQL database administrator, one of your key responsibilities is maintaining performance and stability — especially in production environments. However, no matter how optimized your setup is, there will be times when certain

sessions run queries that consume too many resources, cause locking issues, or block other critical operations.

PostgreSQL offers two built-in functions to help DBAs manage these situations:

- `pg_cancel_backend()` – Used to **cancel a currently running query** without disconnecting the user session.
- `pg_terminate_backend()` – Used to **forcefully disconnect** a user session.

Both tools are essential for safe and effective database administration, but it's crucial to understand when and how to use each one. Let's start with the safer and more conservative option: `pg_cancel_backend()`.

1️⃣ `pg_cancel_backend()` – Safely Cancel Queries Without Disconnecting Users

✅ What Is `pg_cancel_backend()`?

`pg_cancel_backend()` is a PostgreSQL function that allows you to **cancel the currently running SQL query** in a backend process. It does **not terminate the session** — the user remains connected to the database and can continue working after the query is stopped.

In other words, the session stays alive, and only the active SQL command is interrupted. This makes it a safe and non-disruptive way to handle problematic queries in multi-user systems.

🧠 How It Works

To use `pg_cancel_backend()`, you need the **process ID (PID)** of the backend that's executing the problematic query. PostgreSQL stores this information in the `pg_stat_activity` system view, which contains details about all active sessions and queries.

🔍 Step 1: Identify the Process

Run the following query to see all active backend sessions and their states:

```
SELECT pid, usename, state, query, backend_start, query_start
FROM pg_stat_activity
WHERE state = 'active';
```

This will give you a list of all currently running queries, who is running them, and when they started. Look for:

- Queries that are running unusually long
- Queries that are blocking other sessions
- Resource-heavy operations you suspect are causing performance problems

⛔ Step 2: Cancel the Query

Once you identify the offending process, run:

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

Replace `<pid>` with the actual process ID of the session running the long or blocking query.

⚠️ What Happens When You Cancel a Query?

Here's what you can expect:

- Only the **current SQL statement** is canceled.
- The session remains **open and active.**
- If the query was part of a larger transaction, the transaction **remains in progress** unless manually rolled back or committed by the user.
- There is **no loss of connection** to the client application.

This makes `pg_cancel_backend()` **ideal for gently intervening** when something goes wrong without causing panic or data loss for the end user.

📘 Example Scenario

Let's imagine a scenario where a developer is testing on production (bad idea, but it happens) and runs this unfiltered query:

```
SELECT * FROM orders;
```

The `orders` table has 100 million rows. This query now causes high I/O, slows down performance, and other users complain of slowness.

As a DBA, you do the following:

1. Run:

```
SELECT pid, usename, query, state FROM pg_stat_activity WHERE state = 'active';
```

1. Find the offending query with PID `23456` .
2. Cancel it:

```
SELECT pg_cancel_backend(23456);
```

The heavy query stops immediately. The developer is still connected and can continue working without needing to log in again or restart their tool.

🔐 Permissions Required

To use `pg_cancel_backend()`, your database role must have:

- Superuser privileges **OR**
- The ability to cancel sessions owned by your own user

This means only superusers or roles with sufficient rights can cancel queries run by others.

🧩 When Should You Use It?

Use `pg_cancel_backend()` when:

- You want to **avoid disrupting** a user session.
- A query is running too long and needs to be stopped.
- A blocking query is **holding locks** and preventing others from progressing.
- You want to act **quickly but safely** to restore performance.

🔚 Summary

`pg_cancel_backend()` is an essential PostgreSQL function that every DBA should have in their toolkit. It's:

- 🛡️ Safe — Only cancels the current query, not the session.
- 🧘 Non-intrusive — Doesn't force users to reconnect.
- ⚡ Effective — Instantly halts slow or blocking queries.
- 👷 User-friendly — Lets users continue without losing their work.

In environments where uptime and stability are key, it's often your **first line of defense** before resorting to more aggressive actions like `pg_terminate_backend()`.

2️ `pg_terminate_backend()` – Forcefully Kill a PostgreSQL Session

In a PostgreSQL environment, performance and system stability are critical, especially when serving multiple users or applications. While `pg_cancel_backend()` is useful for safely stopping a running query, there are times when you need to **go one step further** and **terminate the entire session.**

This is where `pg_terminate_backend()` becomes essential.

✅ What Is `pg_terminate_backend()` ?

`pg_terminate_backend()` is a PostgreSQL function that **forcibly disconnects** a user session by **killing its backend process.** It doesn't just stop the current query — it ends the whole session and removes it from the system.

When used:

- The **client connection is closed immediately.**
- Any **uncommitted work is rolled back.**
- All associated locks and resources are released.

This is a **powerful and irreversible operation**, and should be used with care.

💥 When Should You Use It?

There are several scenarios where `pg_terminate_backend()` is the right choice:

- A session is **frozen or unresponsive**, and `pg_cancel_backend()` fails to stop it.
- A session is **idle in a transaction** but **holding locks** that block others.
- A backend process is **consuming excessive CPU or memory.**
- A client application crashed, but its connection remains active.

- You need to quickly restore system responsiveness during peak hours.

In all these cases, terminating the backend is often the fastest way to fix the issue.

🧠 How Does It Work?

🔍 Step 1: Identify the Offending Session

To use `pg_terminate_backend()`, you must first locate the PID (process ID) of the session you want to terminate. You can retrieve it from the `pg_stat_activity` system view:

```
SELECT pid, usename, application_name, state, query
FROM pg_stat_activity
WHERE state <> 'idle';
```

This query shows all active or running sessions, including the user, application name, query, and current state.

🔨 Step 2: Terminate the Session

Once you identify the PID of the problematic session (e.g., 78901), you can run:

```
SELECT pg_terminate_backend(78901);
```

This will:

- Kill the backend process with PID 78901.
- Immediately disconnect the session.
- Release any locks or resources it was holding.

The session will disappear from `pg_stat_activity`, and any blocked operations will now be free to proceed.

⚠️ What Are the Effects?

Here's what happens when you use `pg_terminate_backend()`:

- 🛑 The session is **terminated without warning.**
- 🔄 **Uncommitted transactions are rolled back** — so any data changes in progress are lost.
- 📤 The client (e.g., psql, pgAdmin, a web app) will receive an error like:

```
FATAL: terminating connection due to administrator command
```

- 🧹 All locks, memory, and temp resources used by the session are freed.

This command is **more aggressive** than `pg_cancel_backend()` and should be used carefully — especially in production environments.

🔐 Who Has Permission to Use It?

- Only **superusers** can run `pg_terminate_backend()` on other users' sessions.
- Regular users can only terminate their own sessions (if they have a separate login).
- This restriction helps **prevent accidental misuse** or sabotage in shared systems.

📘 Real-Life Example

Suppose an analytics dashboard runs a complex query that takes too long. You try:

```
SELECT pg_cancel_backend(56789);  -- But it doesn't stop
```

The session is stuck, continues using CPU, and blocks other users.

You then run:

```
SELECT pg_terminate_backend(56789);
```

The session is killed. The dashboard disconnects and shows an error. But your system performance recovers instantly, and blocked transactions resume.

🧩 Best Practices

Before using `pg_terminate_backend()`, always:

- Try `pg_cancel_backend()` first.
- Confirm the session is truly causing an issue.
- Communicate with the user if possible (in dev/test environments).
- Monitor system logs to track usage of this command (audit purposes).

🔚 Final Thoughts

`pg_terminate_backend()` is an essential administrative tool for DBAs working with PostgreSQL. It provides a **last-resort option** to maintain performance, resolve deadlocks, and eliminate stuck sessions. When used responsibly, it can prevent serious outages and keep your system running smoothly.

> But remember: with great power comes great responsibility. Use it wisely.

🛠️ Step-by-Step: How to Terminate a PostgreSQL Session

🔎 Step 1: Identify Active Sessions Using `pg_stat_activity`

Before you can cancel or terminate a PostgreSQL session, you first need to identify which sessions are currently connected to the database and what they are doing. This is a critical step for any database administrator to ensure that only **problematic or unresponsive sessions** are targeted — not regular or system-critical ones.

PostgreSQL provides a **system view called `pg_stat_activity`**, which acts like a **real-time dashboard** showing all active connections to the database. You can use this view to monitor performance, troubleshoot locking/blocking issues, and gather the exact **process ID (PID)** required to cancel or kill a session.

🧠 Why Is This Step Important?

Imagine your PostgreSQL database is slowing down. Users report that the application is stuck, or reports aren't loading. A likely cause is a session that is:

- Running a **very long or resource-heavy query**
- Holding a **lock** on an important table
- Left **idle in transaction**, preventing maintenance jobs like `VACUUM` from running
- Completely **unresponsive** due to client-side failure (e.g., a user closed their app while a session was open)

In such cases, blindly terminating sessions is dangerous. Instead, you need to **pinpoint the exact session** causing trouble — and `pg_stat_activity` helps you do exactly that.

🧪 Check the Structure of `pg_stat_activity`

To see the structure of the `pg_stat_activity` view — including all columns it offers — you can run:

```
\d pg_stat_activity
```

This command reveals useful fields like:

- `datname` : the name of the connected database
- `pid` : the backend process ID (what you'll use to cancel or terminate)
- `usename` : the connected user
- `application_name` : what app or tool initiated the connection
- `state` : what the session is doing (active, idle, etc.)
- `query` : the current SQL query being executed

📥 Run This Query to List All Active Sessions

Now let's put this into practice. Run the following query to view detailed information about every session connected to your database:

```
SELECT
    datname,            -- Database name
    pid,                -- Backend process ID (needed for termination)
    usename,            -- Username of the client
    application_name,   -- Tool or software used (psql, pgAdmin, BI tool)
    state,              -- What the session is doing right now
    query               -- SQL query being executed (or last run)
FROM
    pg_stat_activity;
```

🔍 What Each Field Tells You

Column	Meaning
datname	The name of the database the session is connected to (e.g., `salesdb`, `inventorydb`).
pid	The unique process ID for that session. You will use this value with `pg_cancel_backend()` or `pg_terminate_backend()`.
usename	The name of the user who connected. Helpful for identifying which team or person is responsible.
application_name	Shows the application used to connect (e.g., pgAdmin, psql, a reporting tool, or a custom app).
state	Describes the current activity of the session. Common values include:• `active`: running a query right now• `idle`: connected but not running anything• `idle in transaction`: not active, but still holding a transaction open• `idle in transaction (aborted)`: the session hit an error and hasn't exited the transaction
query	The SQL query the session is running (or the last one it ran). This can help you judge if the session is misbehaving.

🖼️ Sample Output

Let's say you run the query and get this output:

datname	pid	usename	application_name	state	query
mydb	24501	alice	pgAdmin	active	SELECT * FROM huge_sales_table;
mydb	24503	bob	BI_Dashboard	idle in transaction	BEGIN;
finance	24506	system	monitoring_tool	idle	

From this, you might conclude:

- Session **24501** is running a long query — maybe too heavy for current resources.

- Session **24503** is idle in transaction — meaning it's not doing anything but still locking resources.
- Session **24506** is idle and not running anything currently.

You can now decide which session you want to cancel or terminate.

✅ Pro Tips

1. **Filter only active or idle-in-transaction sessions** to focus on potentially harmful activity:

```
SELECT pid, usename, state, query
FROM pg_stat_activity
WHERE state IN ('active', 'idle in transaction');
```

1. **Exclude your own session** to avoid accidentally terminating your admin connection:

```
SELECT pid, usename, query
FROM pg_stat_activity
WHERE pid <> pg_backend_pid();
```

1. **Order by longest-running sessions** (optional but useful in performance tuning):

```
SELECT pid, usename, query_start, state, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY query_start ASC;
```

🔚 What's Next?

Once you've identified the problematic session and obtained its PID, you're ready to **take action:**

- Use `pg_cancel_backend(pid)` to gently stop the query
- Use `pg_terminate_backend(pid)` to forcefully end the session

🛠️ Step 2: 🚦 Choose How to Terminate the Session

After identifying the session that's causing problems in your PostgreSQL database (using `pg_stat_activity` in Step 1), the next step is to **decide how to terminate it.**

PostgreSQL provides **two different functions** for this purpose — one that tries to **gracefully stop the query**, and another that **forcefully kills the session.** Choosing the right one depends on the situation.

Let's walk through both options in detail:

✅ Option A: Cancel the Running Query

Function: `pg_cancel_backend(pid)`

This is the **safest** and **least disruptive** method. It attempts to **cancel only the SQL query** that is currently running in the specified session. The user stays connected, and the session remains active.

🔧 **Syntax:**

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

🧪 **Example:**

```
SELECT pg_cancel_backend(3674);
```

In this example, `3674` is the process ID (PID) of the session you want to cancel. Replace it with the actual PID from `pg_stat_activity`.

📝 **What It Does:**

- Attempts to stop the query currently being executed by the session.
- Leaves the connection open — the user will **not be disconnected**.
- The session can still be used to run more queries afterward.
- Any open transaction **remains in progress** unless the user explicitly commits or rolls it back.

✅ **Use Case:**

This method is ideal when:

- A user runs a **long-running query** by mistake.
- A query is **causing performance issues** but doesn't need to kill the whole session.
- You want to **avoid interrupting the user** (especially in production environments).
- You want to be **non-intrusive** — e.g., in shared databases with many users.

⚠️ **Limitations:**

- It only works if the query is **currently running** (i.e., session is in `'active'` state).
- If the session is idle or stuck in a non-cancellable state (e.g., `idle in transaction` or system wait), this command **may not have any effect**.
- It does **not release locks** held by the session — the transaction remains active.

💥 Option B: Terminate the Entire Session

Function: `pg_terminate_backend(pid)`

This is a **more aggressive approach** that **completely ends the session.** It forcefully **disconnects the user**, rolls back any in-progress work, and releases all locks held by the session.

🔧 **Syntax:**

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

💉 **Example:**

```
SELECT pg_terminate_backend(5983);
```

📝 **What It Does:**

- Instantly **kills the backend process** of the session.
- **Disconnects** the user or client application.
- **Rolls back any uncommitted transactions.**
- **Releases all locks** held by that session.
- Removes the session from `pg_stat_activity`.

✅ **Use Case:**

Use `pg_terminate_backend()` when:

- A session is **unresponsive** (e.g., the user closed the app but the session is still active).
- A session is **idle in transaction** and **blocking others.**
- `pg_cancel_backend()` **fails to cancel the query.**
- The session is **holding critical locks** and needs to be ended immediately.
- The client process has **crashed**, leaving behind a “zombie” session.

⚠️ **Things to Be Careful About:**

- It **forces the user to reconnect** — this may cause application errors.

- **Unsaved work is lost** — because any open transaction is rolled back.
- If the session belongs to a background process (like an ETL or report), it may **interrupt important operations.**

🔍 Comparison: When to Use Each Method

Situation	Recommended Function	Why?
Long-running SELECT slowing down system	`pg_cancel_backend()`	Stops the query safely, user stays connected
Query won't stop with cancel	`pg_terminate_backend()`	Forcefully ends session
Session is idle but holding locks	`pg_terminate_backend()`	Cancels session and frees up resources
User forgot to commit and left session open	`pg_terminate_backend()`	Ends session, rolls back transaction
You want to avoid user disruption	`pg_cancel_backend()`	Safe, non-intrusive

🧯 Best Practices

- **Start with** `pg_cancel_backend()` whenever possible — it's a softer approach.
- **Use** `pg_terminate_backend()` only when cancellation fails or a session is clearly causing harm.
- Always double-check the **PID** before running either command — terminating the wrong session can lead to lost data or service disruption.
- If managing critical production databases, make sure you have **proper logging** and **alerts** set up to track these actions.

📌 Permissions Required

To use either of these functions:

- You must be a **superuser** or a user with appropriate privileges.
- Regular users **can only cancel or terminate their own sessions.**

✅ Step 3: Verify That the Session Is Gone

After using `pg_cancel_backend()` or `pg_terminate_backend()` to deal with a problematic session, it's important to confirm that your action was successful. This final step ensures that the **query has stopped** (if canceled) or that the **session has been completely removed** (if terminated).

In this step, you'll learn how to **re-check PostgreSQL's live session status** and confirm that everything is back to normal.

🔄 How to Verify the Result

PostgreSQL keeps track of all running sessions in a special system view called **`pg_stat_activity`**. This view shows you:

- Which users are connected
- What queries are running
- Which session IDs (PIDs) are active
- What applications are being used
- What state each session is in (e.g., active, idle, idle in transaction)

To verify that a session or query has been stopped, run the following SQL command:

```
SELECT datname, pid, usename, application_name, state, query
```

```
FROM pg_stat_activity;
```

This gives you a **live snapshot** of all current connections and what they're doing.

🔍 What to Look For

After running the command above:

- If you used `pg_cancel_backend(PID)`, then:
- The query under that PID should **no longer appear** in the `query` column.
- The session **will still be listed**, but it will likely show a new or empty query with a state like `idle`.
- If you used `pg_terminate_backend(PID)`, then:
- The **entire PID will be gone** — it should not appear in the output at all.
- This confirms that the session has been **fully terminated**, all associated locks are released, and the user is disconnected.

🧪 Practical Example: End-to-End Session Management

Let's go through a real-world scenario where you simulate PostgreSQL session activity and manage them step by step.

🧬 Step 1: Create a Test Database with Simulated Load

To practice managing sessions, you can use PostgreSQL's built-in benchmarking tool — `pgbench`.

This tool creates a test database and simulates active user sessions.

```
psql -c "CREATE DATABASE apps;"
pgbench -i apps                  # Initializes the database with test tables
pgbench -c50 -t10000 apps        # Starts 50 client sessions, each running 10,0
```

This setup simulates a busy database environment where you may need to cancel or terminate sessions.

👁️ Step 2: View Current Sessions

After running `pgbench`, your PostgreSQL server will have multiple active sessions.

You can list them with:

```
psql -c "SELECT datname, pid, usename, application_name, state, query FROM pg_s
```

You'll now see many rows, one for each session, showing:

- The process ID (`pid`)
- The username (`usename`)
- The name of the client app (`application_name`)
- The current session state (`state`)
- The current or last executed SQL command (`query`)

Look for:

- One PID that is running a query (state = 'active')
- One PID that is idle or stuck (state = 'idle in transaction')

⛔ Step 3: Cancel a Query (Graceful Action)

Let's assume PID `3674` is running a long query you want to stop.

Use the following command:

```
psql -c "SELECT pg_cancel_backend(3674);"
```

This sends a polite signal to that backend process to **stop the currently running SQL command**, but it **won't disconnect the session.**

💣 Step 4: Terminate a Session (Forceful Action)

Now, assume PID `5983` is in a locked or idle state and not responding.

To disconnect it completely and release any held resources, run:

```
psql -c "SELECT pg_terminate_backend(5983);"
```

This will:

- Forcefully kill the session
- Roll back any uncommitted changes
- Remove the session from PostgreSQL

✅ Step 5: Confirm the Result

To check that your commands worked, re-run the session query:

Now:

- The session with **PID** `3674` should still be listed, but **its query should no longer be running** — the state might now be `idle`.
- The session with **PID** `5983` should be **gone** — no longer appearing in the list. This confirms that the session has been successfully terminated.

✨ Done! You've Managed PostgreSQL Sessions Successfully

You've just completed the full process of:

1. Monitoring PostgreSQL activity in real time
2. Identifying sessions that needed attention
3. Gracefully canceling a long-running query
4. Forcefully terminating a stuck or idle session
5. Verifying that your changes worked

🎯 Final Thoughts

Being able to monitor and manage PostgreSQL sessions is one of the most important skills for a database administrator. It helps you:

- Keep the system responsive
- Prevent long query blocks
- Handle unresponsive clients safely
- Protect users from performance degradation

Whether you're managing a test environment or a critical production database, knowing how to **observe, act, and verify** session behavior will make you a more effective PostgreSQL operator.

⚠️ Important Considerations

Before using powerful commands like `pg_cancel_backend()` and `pg_terminate_backend()`, it's important to understand the **risks and behavior** of these actions. These commands directly affect live sessions in your PostgreSQL database, so mistakes can lead to serious issues. Let's go over the most important things to keep in mind.

🛑 Always Double-Check the PID

Every session in PostgreSQL has a unique **Process ID (PID)**. This number is what you use to cancel or terminate a session.

But here's the catch:
If you accidentally cancel or terminate the wrong PID — like one used by your production app or a critical background process — you might:

- Interrupt live users
- Cause an application error
- Lose unsaved data
- Create system downtime

✅ What to do:

Before taking any action, always run a query on `pg_stat_activity` to see full details about the session:

- Who is using it
- What it's doing

- What application it's from

This helps you **make sure you're targeting the right session.**

🧹 Resource Cleanup and Rollbacks

When you use `pg_terminate_backend()`, PostgreSQL will:

- Immediately disconnect the session
- Roll back any **uncommitted transactions**
- Release any **locks** or **temporary files**

This can be helpful when a session is stuck or blocking others. But it also means:

- Any **unsaved work will be lost**
- If the session was doing something important (like updating many records), that work will be undone

So it's important to ask:

> "Is it safe to terminate this session now?"

🕐 Delays with pg_cancel_backend()

The `pg_cancel_backend()` function tries to **gracefully stop the currently running query**, but sometimes it may not work instantly.

For example:

- The query might be **waiting on a lock** or **I/O operation** (like reading from disk)
- It might be inside a part of PostgreSQL that doesn't allow interruption
- The query might look stuck but is actually finishing something important

In these cases, `pg_cancel_backend()` may take **a few seconds**, or may **fail to cancel** the query at all.

✅ If that happens, you can wait and try again — or if necessary, **switch to** `pg_terminate_backend()` to force the session to stop.

📝 Conclusion

Managing sessions is one of the most important jobs for anyone responsible for a PostgreSQL database. When your system slows down or sessions get stuck, you need to act quickly and carefully.

PostgreSQL gives you two powerful tools:

🧰 Your Two Main Options

- `pg_cancel_backend()` – This is the **safe, gentle option.** It stops a query without disconnecting the user.
- `pg_terminate_backend()` – This is the **aggressive option.** It disconnects the session immediately and rolls back anything it was doing.

🚀 Why These Tools Matter

By learning how to use these two functions properly, you can:

- Keep your database **running smoothly**
- **Prevent performance issues** during peak usage
- **Avoid downtime** without restarting the server
- Act fast during incidents without needing a full database restart

In high-stakes environments — especially in production — knowing how to handle sessions **safely and quickly** makes you a better DBA or developer.

✅ So always:

- Check the session carefully
- Cancel if possible
- Terminate only when necessary

With these tools and the knowledge of when to use them, you now have **full control over PostgreSQL session management.**

👉 *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 MySQL Oracle AWS Sql

Following

Written by Jeyaram Ayyalusamy

51 followers · 2 following

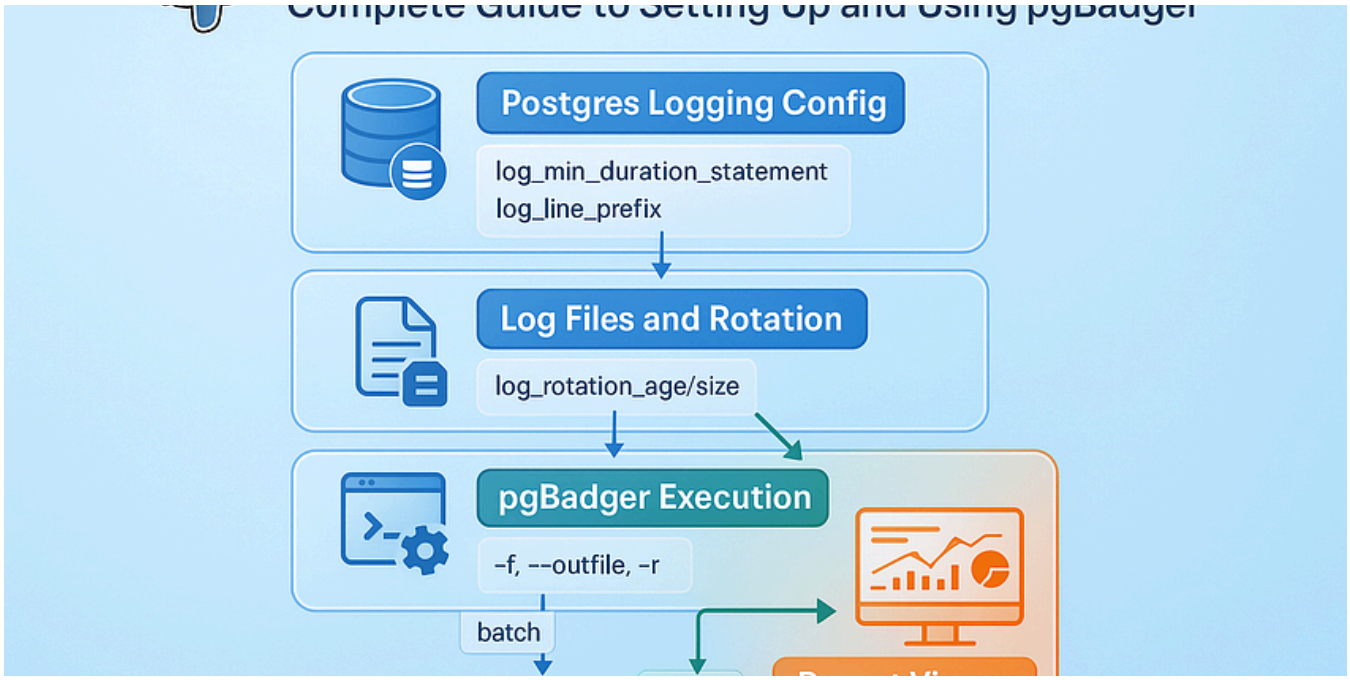
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

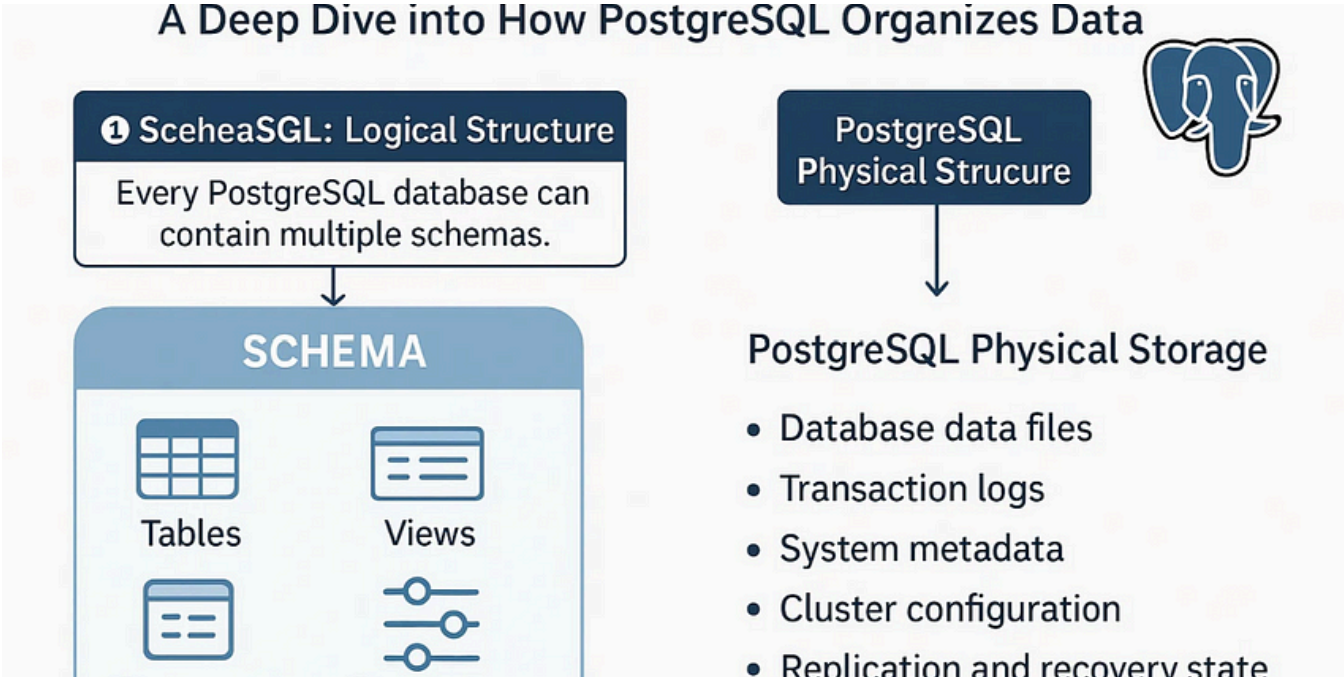


Jeyaram Ayyalusamy

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

The Internal Structure of PostgreSQL: A Deep Dive into How PostgreSQL Organizes Data

PostgreSQL is one of the most powerful and popular open-source relational database systems used in production today. But while most people...

Jun 1 2

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

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

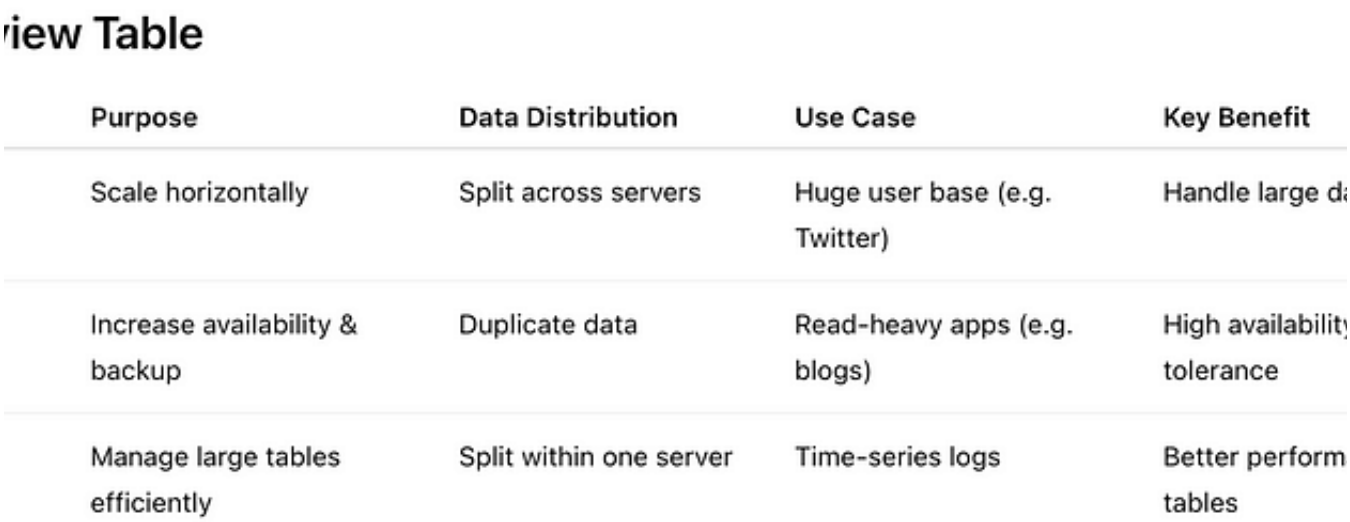
Recommended from Medium

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.

5d ago 37



iew Table

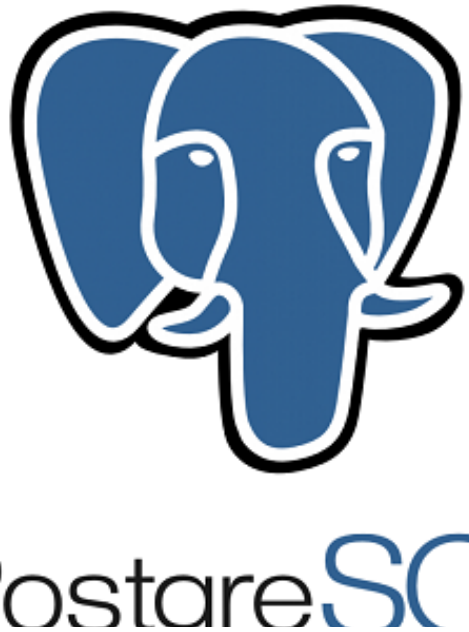
	Purpose	Data Distribution	Use Case	Key Benefit
	Scale horizontally	Split across servers	Huge user base (e.g. Twitter)	Handle large da
	Increase availability & backup	Duplicate data	Read-heavy apps (e.g. blogs)	High availabilit tolerance
	Manage large tables efficiently	Split within one server	Time-series logs	Better perform tables

Ravi Jaisawal

Sharding vs Replication vs Partitioning in Databases

Differences between Sharding, Replication, and Partitioning in databases — with practical examples and real-world use cases

Jun 11 4

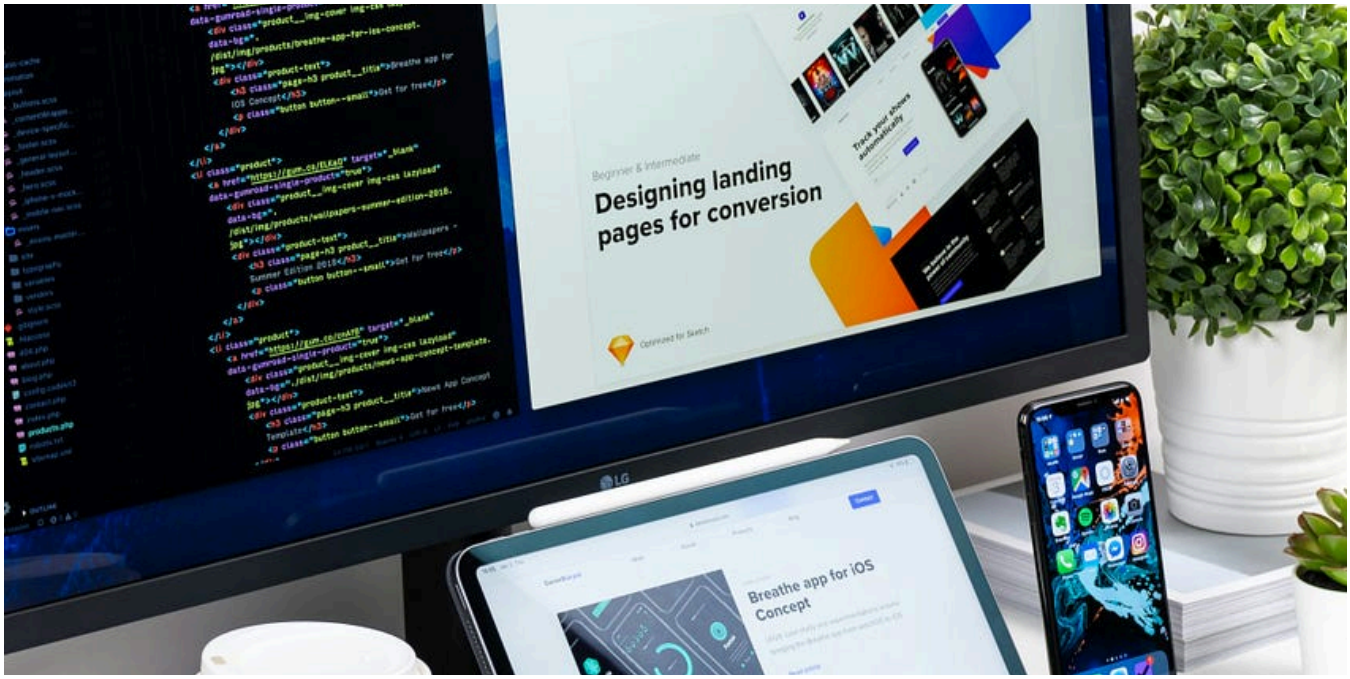


Sohail Saifi

Postgres Hidden Features That Make MongoDB Completely Obsolete (From an Ex-NoSQL Evangelist)

For six years, I was that developer.

May 26 203 11



Sandesh I DevOps | CI/CD | K8

“Why PostgreSQL is Beating MongoDB in 2025 (And Why I Switched Back)”

(Spoiler: My NoSQL experiment cost us 3 months of debugging headaches)

Jun 11 10 1

Rizqi Mulki

PostgreSQL + Node.js: How We Achieved 50K Transactions per Second

Connection Pooling, Write-Ahead Log Tuning, and Avoiding N+1 Queries

Apr 19 23

In Level Up Coding by Daniel Craciun

Stop Using UUIDs in Your Database

How UUIDs Can Destroy SQL Database Performance

6d ago 122 8