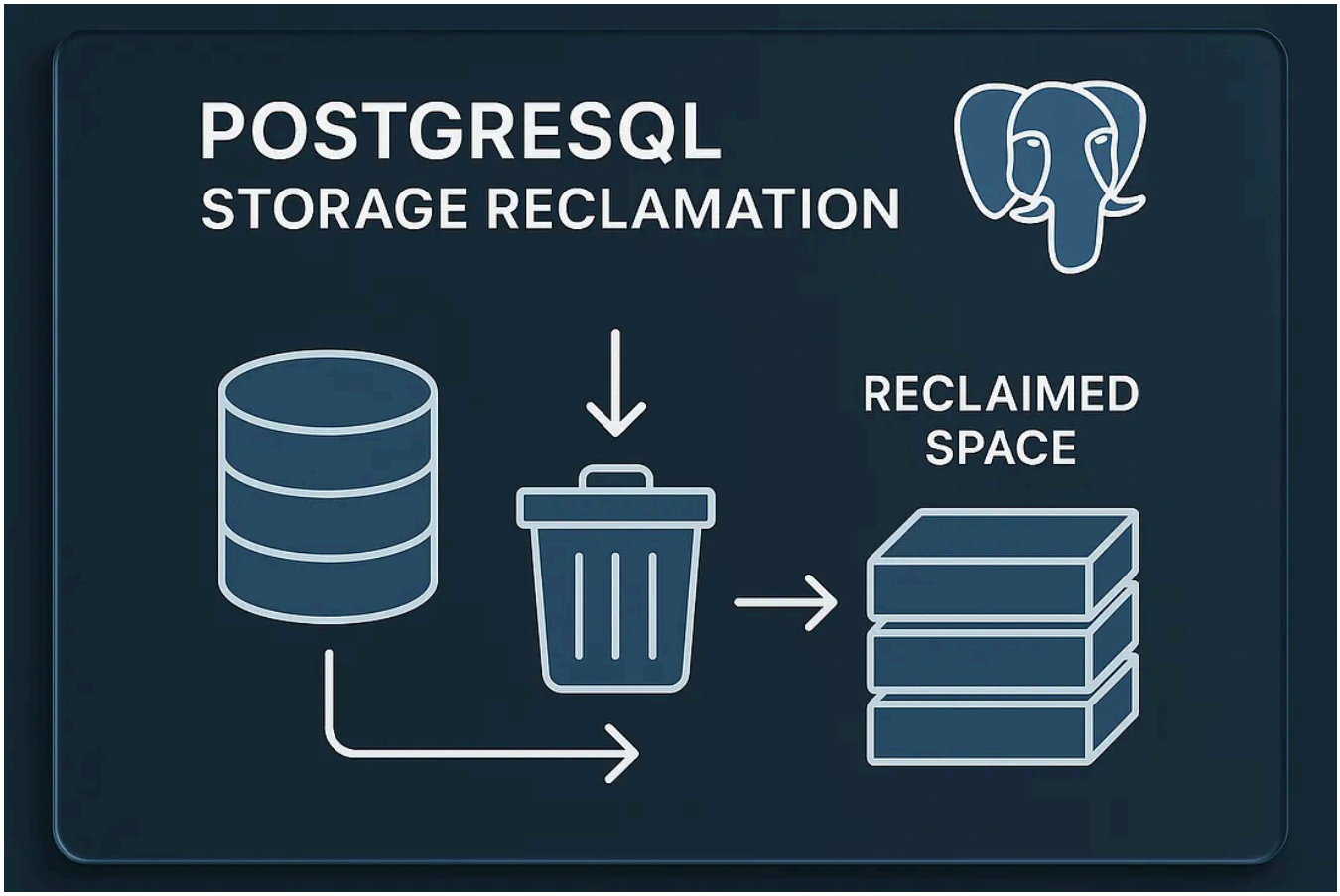
PostgreSQL 17: How to Drop a Database and Fully Reclaim Disk Space

15 min read · Jun 12, 2025

J Jeyaram Ayyalusamy Following



Managing storage efficiently is a critical part of PostgreSQL database administration — especially when working with large datasets or limited storage. Dropping a database may seem like a simple task, but fully reclaiming the freed-up space often requires a deeper understanding of PostgreSQL's internal behavior.

In this guide, I'll walk you through how to drop a PostgreSQL database and completely reclaim disk space on **PostgreSQL 17**.

🔨 Dropping a PostgreSQL Database — A Cautionary Guide

When working with PostgreSQL, there may come a time when you need to **delete an entire database** — perhaps to clean up unused test environments or decommission a project. PostgreSQL provides a simple command to do this, but it's **critical to understand the implications** before using it.

🚀 The Simple Command

Dropping a PostgreSQL database can be done in a single command:

```
CREATE DATABASE demodb;
DROP DATABASE demodb;
```

In this example:

- `CREATE DATABASE demodb;` creates a sample database named `demodb`.
- `DROP DATABASE demodb;` **completely and permanently deletes** that database.

✅ What Gets Deleted?

When you execute `DROP DATABASE`, PostgreSQL removes **everything inside** the specified database, including:

- **All tables:** Every table and its data will be erased.
- **Indexes:** Any performance-related indexing structures are also removed.

- **Sequences:** These are used for auto-incrementing values (like IDs) and will be deleted too.
- **Functions:** Any stored procedures or user-defined functions within the database.
- **Data files:** The physical files associated with the database on disk are deleted.

This command is **non-reversible** — PostgreSQL doesn't move the data to a trash bin. It's gone.

⚠️ A Word of Caution

> ❗ **Important:** Once a PostgreSQL database is dropped, there's no way to get it back — **unless you have a backup.**

This is why it's **highly recommended** to:

- Double-check the database name before dropping.
- Ensure no important data is stored in the database.
- Perform a backup using `pg_dump` or your preferred method before deletion.

🧠 Summary

Dropping a PostgreSQL database is easy, but it's a **destructive operation**. Always think twice before running this command:

```
DROP DATABASE your_database_name;
```

If used responsibly, it's a powerful way to maintain a clean and efficient PostgreSQL environment.

🔐 Who Can Drop a PostgreSQL Database? And How to Do It Safely

Deleting a PostgreSQL database is a powerful operation — and one that comes with a few important restrictions and best practices. Whether you're using SQL or the command line, this guide walks you through **who can drop a database**, the **rules you must follow**, and how to use the `dropdb` utility effectively.

👨‍💼 Who Can Drop a Database?

Not everyone has the authority to remove a database. PostgreSQL enforces strict permission rules to ensure only the right users can perform this sensitive operation.

✅ You **must be one of the following:**

- **The database owner** — the user who originally created the database.
- **A PostgreSQL superuser** — a user with elevated administrative privileges across the entire PostgreSQL instance.

If you are not either of these, PostgreSQL will block the drop operation.

❌ Restrictions While Dropping a Database

One common mistake many developers encounter is trying to drop the **currently connected database.** PostgreSQL doesn't allow this for good reason — you can't remove the floor you're standing on.

🔒 Rule:

> **You cannot drop the database you are currently connected to.**

🛠️ Solution:

To drop a database, you first need to **switch your connection** to another existing database — typically `postgres` or `template1`.

Here's how to do it from the PostgreSQL interactive terminal (`psql`):

```
\c postgres
DROP DATABASE demodb;
```

- `\c postgres` switches your session to the default `postgres` database.
- `DROP DATABASE demodb;` then safely removes the `demodb` database.

💻 Using the `dropdb` Shell Utility

PostgreSQL also provides a convenient **command-line tool** called `dropdb` . This utility simplifies the process of dropping a database without manually launching the SQL console.

📌 Example:

```
dropdb demodb
```

This is functionally equivalent to:

```
DROP DATABASE demodb;
```

✅ Internally, `dropdb` is just a wrapper around the `DROP DATABASE` SQL command—but it saves time for developers who prefer command-line workflows.

🧠 Summary

Dropping a PostgreSQL database isn't just about running a command — it requires the right permissions, the right connection, and sometimes the right tool.

- Only the **database owner** or a **superuser** can drop a database.
- You **must switch to another database** before executing the drop.
- The `dropdb` utility offers a quicker way via the command line.

Handle with care — once dropped, your database and its data are permanently deleted (unless you have backups).

🗄️ Dropping a PostgreSQL Database: But What About Disk Space?

Dropping a PostgreSQL database might seem like a quick fix to free up disk space. And yes, it does **remove all logical database objects** — tables, indexes, sequences, and more. But here's the catch:

> 💡 The **disk space** isn't always immediately returned to the operating system.

Why does that happen? Let's explore the **underlying reasons** and what you can do to truly reclaim space after dropping a PostgreSQL database.

🔍 Why Disk Space May Not Be Immediately Reclaimed

Even after executing `DROP DATABASE`, you may notice that your storage usage hasn't gone down as expected. Here are the key reasons:

🧱 1. Filesystem Behavior

Most operating systems don't instantly release disk space just because a file is deleted. If the **file is still in use** — say, held open by a running process — it remains on disk until it's fully closed.

🔧 **Tip:** Restarting PostgreSQL or ensuring no lingering processes are referencing those files can help.

🔐 2. Open File Handles

PostgreSQL processes (or other tools) may keep file handles open even after a database is dropped. This causes the filesystem to **retain the underlying storage** until those handles are released.

👁️ **Monitor:** Use tools like `lsof` or `pg_stat_activity` to check for open files.

🗂️ 3. Tablespace Management

If your dropped database used **custom tablespaces**, those directories and their contents might not be deleted automatically — especially if they exist **outside PostgreSQL's main data directory**.

⚠️ **Check:** Manually inspect and clean up old tablespace directories, but **only if you're 100% sure** they're no longer in use.

🔄 4. WAL Archiving and Retention

PostgreSQL uses **Write-Ahead Logging (WAL)** for crash recovery and replication. Dropping a database can still leave behind WAL files if:

- Archive mode is enabled
- Replication slots are active
- The retention policy hasn't purged old logs yet

📦 These logs can **consume gigabytes** of space over time.

🛠️ **Solution:**

- Clean up old WAL files using `pg_archivecleanup`
- Adjust retention settings (`wal_keep_size`, `archive_timeout`)
- Remove unused replication slots if no longer needed

✅ How to Truly Reclaim Disk Space

To ensure space is actually released after a drop:

1. **Stop PostgreSQL temporarily** (if possible) to release any open handles.
2. **Verify WAL retention and archive settings.**
3. **Check tablespace directories** and remove unused ones manually.
4. **Monitor disk space** using tools like `du`, `df`, or `ncdu`.

And always — **back up before deleting anything manually.**

🧠 Final Thought

Dropping a PostgreSQL database is just the first step. If you're looking to **reclaim actual disk space**, you'll need to understand the interplay between PostgreSQL, the filesystem, and system-level processes.

> 🔁 Think of it this way: just because a book is removed from a library shelf doesn't mean the shelf has been taken down.

Make space intentionally. Make it count.

🧹 Full Space Reclamation in PostgreSQL: Backup Before Rebuilding

Over time, PostgreSQL databases can accumulate data bloat, old log files, and unused tablespaces that consume significant disk space. In extreme cases —

especially in long-running environments — you may need to **recreate your PostgreSQL cluster entirely** to reclaim space and optimize performance.

But before doing anything destructive, you must ensure **your data and configuration files are safely backed up.** This article covers the first and most crucial steps of a full PostgreSQL cleanup: **backing up your databases and configurations.**

🛑 Why You Must Backup Before Reinitialization

Recreating the PostgreSQL cluster involves deleting the entire `data` directory—which means **losing all databases, roles, configurations, and system metadata.** There's no "undo" button.

To safely proceed, you need two things:

1. A complete backup of **all databases and roles.**
2. A backup of your PostgreSQL **configuration files.**

Let's break down how to do this properly.

1️⃣ Backup All Databases Using `pg_dumpall`

PostgreSQL provides a built-in utility called `pg_dumpall` that exports **every database** in your instance, along with **roles, privileges, and global settings.**

🔧 Run the Command:

```
sudo su - postgres
pg_dumpall | gzip -9 > /var/lib/pgsql/backups/all.dbs.sql.gz
```

🔍 What This Does:

- `sudo su - postgres` : Switches to the PostgreSQL system user.

- `pg_dumpall` : Dumps all databases in a single SQL script.
- `gzip -9` : Compresses the dump at the highest compression level.
- `> /var/lib/pgsql/backups/all.dbs.sql.gz` : Redirects output to a compressed file in the backups directory.

💡 Why Compression Matters:

Databases can be large, and disk space may already be low. Compressing the backup:

- Saves space
- Speeds up transfer and storage
- Reduces risk of disk exhaustion during backup

✅ **Pro Tip**: Verify the dump file by checking its size and optionally grepping for expected schema/table names.

2️⃣ Backup PostgreSQL Configuration Files

Next, you need to preserve the **server configuration files**. These define how PostgreSQL behaves — how much memory it uses, who can connect, logging rules, and replication settings.

🔧 Run the Command:

```
cp /var/lib/pgsql/data/*.conf /var/lib/pgsql/backups/
```

This copies all configuration files from the data directory into your backups folder. Files typically include:

- `postgresql.conf` : The main server settings (memory, parallelism, timeouts, etc.)
- `pg_hba.conf` : Host-based authentication rules

- `pg_ident.conf` : User identity mapping for authentication

✅ **Important:** Custom changes made to these files should be preserved to avoid reconfiguring everything from scratch after cluster recreation.

🧠 Summary

These first two steps — **backing up your databases** and **configuration files** — are absolutely essential for a safe and successful PostgreSQL cluster rebuild.

🔒 What You've Achieved:

- A full export of all data, users, and schemas (`pg_dumpall`)
- A compressed and portable backup file
- Safeguarded configuration settings

> Think of it as creating a blueprint and a time capsule before tearing down and rebuilding your home.

3️⃣ Stop PostgreSQL Safely

Before you touch anything in PostgreSQL's internal data directory, you need to **gracefully stop the PostgreSQL server.** This ensures all transactions are flushed, file handles are closed, and memory is cleared.

🛠 Command:

```
pg_ctl -D /var/lib/pgsql/data stop
```

✅ Why This Step Matters:

- Prevents data corruption
- Ensures files aren't being accessed when you delete them

- Cleanly shuts down all background processes and workers

📌 *Note*: If `pg_ctl` is not in your `PATH`, you may need to use the full path (e.g., `/usr/pgsql-17/bin/pg_ctl` depending on your installation).

4️⃣ Delete the Existing Data Directory

With the server safely stopped, you can now **remove the old data directory**, which includes all your databases, system catalogs, WAL files, and internal metadata.

🧨 Command:

```
rm -rf /var/lib/pgsql/data/*
```

💡 *Tip*: You can check the directory size before deletion using:

```
du -sh /var/lib/pgsql/data
```

Removing this directory frees up disk space and clears any hidden storage consumption from internal files, bloated system catalogs, and unused logs.

5️⃣ Reinitialize the PostgreSQL Cluster

Now that the old cluster is removed, it's time to **reinitialize PostgreSQL** with a fresh system catalog and default configuration.

🛠️ Command:

```
initdb -D /var/lib/pgsql/data
```

🔍 What `initdb` Does:

- Creates a fresh directory structure for PostgreSQL
- Initializes the `postgres` database and default roles
- Sets up internal control files for transaction tracking and logging

✅ This step gives you a **clean, unbloated cluster** — ideal for starting fresh or importing optimized data.

📌 *Optional Flags*: You can use `--locale`, `--encoding`, or `--auth` options during `initdb` if you want to specify default character sets or authentication methods.

6️⃣ Restore PostgreSQL Configuration Files

Your original configuration files, backed up in Part 1, contain all your custom server settings — so now is the time to bring them back.

🛠️ Command:

```
cp /var/lib/pgsql/backups/*.conf /var/lib/pgsql/data/
```

⚙️ Files You're Restoring:

- `postgresql.conf` : Controls memory usage, connections, logging, query planner, and more.
- `pg_hba.conf` : Defines which users can connect from where and how.
- `pg_ident.conf` : Maps system users to PostgreSQL roles.

✅ This ensures your new cluster behaves the same as your previous environment — no need to reconfigure from scratch.

💡 *Tip*: Before copying the files, you can review them and tweak any settings based on performance insights or system changes (e.g., if you're migrating to a machine with more RAM or CPU).

🧠 Summary So Far

You've now completed a **major portion** of the PostgreSQL full space reclamation process:

- ✅ Safely shut down the PostgreSQL server
- ✅ Wiped the old data directory and cleared all bloat
- ✅ Reinitialized a fresh PostgreSQL cluster
- ✅ Restored your configuration files

At this point, your PostgreSQL instance is a **clean slate** — ready to be restarted and repopulated with your original data.

7️⃣ Restart PostgreSQL Server

Once you've reinitialized the data directory and restored the `.conf` files, the next step is to **bring PostgreSQL back online**.

🛠 Command:

```
pg_ctl -D /var/lib/pgsql/data start
```

✅ What This Does:

- Boots the PostgreSQL server using the new, clean data directory.
- Applies your previously restored configuration settings.
- Makes the cluster available to accept connections again.

💡 *Tip*: To verify the server has started successfully, you can run:

```
pg_isready
```

Or view logs (typically in `/var/lib/pgsql/data/pg_log/` or via `journalctl` depending on your OS).

> 🔍 If the server fails to start, check for missing config parameters or permission issues in the new data directory.

8️⃣ Restore the Databases

Now that your PostgreSQL server is running, it's time to **restore all your databases and roles** using the backup created in step 1.

🔧 Step-by-Step:

1. **Decompress the Backup File**

```
gunzip /var/lib/pgsql/backups/all.dbs.sql.gz
```

This unpacks the full SQL dump file (`all.dbs.sql`) that contains every database, schema, role, function, and permission from your previous setup.

2. Restore the Dump into PostgreSQL

```
psql < /var/lib/pgsql/backups/all.dbs.sql
```

This command pipes the entire dump file into the running PostgreSQL instance, effectively recreating:

- All user databases
- Tables, indexes, views

- User-defined functions
- User accounts and access privileges

✅ Depending on the size of your database, this may take some time. Monitor system resources if running on production.

🎉 Success! Your PostgreSQL System Is Fully Restored

At this point, your PostgreSQL server is:

- 🧼 **Fully cleaned** — All old bloat, logs, and unused files removed
- 💾 **Disk space reclaimed** — Especially valuable in constrained environments
- 🔄 **Functionally identical** — Same roles, schemas, and configurations as before
- ⚡ **Performance-optimized** — Leaner and ready for future operations

🧠 Final Thoughts

Rebuilding a PostgreSQL cluster might sound intimidating, but when done methodically, it becomes a powerful tool in your database management toolkit.

> It's like formatting your hard drive after years of use — you start fresh, but retain everything that matters.

This process is especially valuable for:

- Annual maintenance and performance tuning
- Recovering from space exhaustion
- Migrating to new hardware or storage setups
- Cleaning up corruption or misconfigurations

🔎 Extra Space Check After Dropping a Database in PostgreSQL

Have you ever dropped a PostgreSQL database expecting instant disk space recovery — only to find that the space wasn't freed? You're not alone. This behavior is common and can confuse even experienced administrators.

In this post, we'll explore **why dropping a database doesn't always immediately free up space**, and how to investigate the underlying reasons.

🔍 1. WAL Archiving Retention

PostgreSQL uses **Write-Ahead Logging (WAL)** to ensure durability and consistency. Every change to the database is recorded in a WAL file before being applied. This is crucial for crash recovery, replication, and PITR (Point-In-Time Recovery).

Even after a database is dropped, **WAL files are not instantly discarded**. Their retention is governed by several configuration parameters, which may instruct PostgreSQL to keep a backlog of these logs.

You can check your WAL-related settings with this query:

```
SELECT name, setting
FROM pg_settings
WHERE name IN ('archive_mode', 'archive_command', 'wal_level', 'wal_keep_size')
```

What These Settings Mean:

- **`archive_mode`** : If set to `on` , PostgreSQL will retain WAL files until they are archived.
- **`archive_command`** : Defines the command to archive WAL files. If misconfigured or failing, WALs can pile up.
- **`wal_level`** : Higher levels like `replica` or `logical` generate more WAL data.
- **`wal_keep_size`** : Specifies how much old WAL data should be retained on disk.

Key Point: If any of these settings are too aggressive or improperly tuned, PostgreSQL might hold onto WAL files long after the associated data is gone — leading to wasted disk space.

🔍 2. Deleted Files Still Held Open

Another common cause of lingering disk usage is **open file descriptors to deleted files.** PostgreSQL (like many UNIX-based processes) can continue using files even after they've been deleted from the filesystem — *as long as the process keeps them open.*

To check for this, use the `lsof` command:

```
lsof | grep deleted
```

This will return a list of deleted files that are still being held open by processes. If PostgreSQL appears in this list, it means the database engine hasn't fully released those resources yet — so the space is still technically in use.

What to Do:

- If safe to do so, **restart the PostgreSQL service.** This usually closes all file descriptors and releases the disk space.
- For production environments, consider using **rolling restarts** or **failover mechanisms** to avoid downtime.

🧠 Conclusion

Dropping a database doesn't always lead to immediate disk space recovery due to:

- **Retained WAL files** driven by archive settings
- **Deleted files still held open** by running PostgreSQL processes

Understanding and checking both scenarios can help you proactively manage disk space, optimize storage usage, and maintain a cleaner PostgreSQL environment.

✅ *Tip*: Always monitor WAL file growth and regularly audit open file handles in long-running environments. These small steps can prevent big surprises.

🔍 3. Active Sessions Still Holding Locks

Even after a database is dropped, **active connections** or **background processes** may continue to hold file locks or cache file handles in memory. This can **prevent PostgreSQL from releasing resources**, including file space.

To identify if there are any lingering sessions, run:

```
SELECT * FROM pg_stat_activity;
```

This system view provides details on all currently active connections to the PostgreSQL server. Look for:

- Connections with `state = 'idle in transaction'`
- Long-running background tasks
- Sessions still connected to the dropped database (before it was removed)

Why This Matters:

PostgreSQL won't fully clean up data files if a session is still referencing them, even indirectly. This means space won't be reclaimed until all sessions have been closed.

What to Do:

- Terminate unnecessary sessions using `pg_terminate_backend(pid)`

- Restart the PostgreSQL server if cleanup is critical and sessions can't be terminated safely

🔍 4. Tablespace Locations

Another often-overlooked reason for lingering space usage is **external tablespaces.** PostgreSQL allows you to create tablespaces — custom locations on the filesystem where data can be stored outside the default data directory (`$PGDATA`).

To Check Default Base Directory:

Navigate into your PostgreSQL data directory:

```
cd $PGDATA/base
```

Then list and count directories (each corresponding to a database):

```
ls | wc -l
```

To see how much space is being used:

```
du -sh `ls`
```

But What About External Tablespaces?

Tablespaces can be located *outside* of `$PGDATA` . These custom paths won't be automatically cleaned up unless:

- The tablespace is explicitly dropped
- All associated data is removed

Even if the database using that tablespace is dropped, the **filesystem directory may still contain files**, consuming space silently.

How to Identify External Tablespaces:

Run the following SQL to view all configured tablespaces:

```
SELECT spcname, pg_tablespace_location(oid)
FROM pg_tablespace;
```

Then manually inspect those locations on disk and remove any unreferenced data if needed.

🧠 Final Thoughts

If you're managing PostgreSQL at scale, simply dropping a database isn't always enough to reclaim disk space. You also need to:

- Check for **active sessions** holding locks or file handles
- Clean up **external tablespaces** that PostgreSQL no longer manages automatically

These steps ensure your storage stays clean, predictable, and under control.

✅ *Tip*: Consider automating periodic checks on `pg_stat_activity` and tablespace usage in your PostgreSQL maintenance scripts or monitoring tools.

🧹 PostgreSQL Storage Cleanup: Final Thoughts and Pro Tips

After exploring why PostgreSQL might not immediately release disk space after dropping a database, let's summarize what we've learned and provide some

practical guidance to ensure you manage storage efficiently in real-world deployments.

📝 Summary: Key Takeaways

PostgreSQL provides powerful tools for managing databases — but when it comes to storage cleanup, there are a few nuances you must understand.

✅ DROP DATABASE removes data logically — not always physically

Using the `DROP DATABASE` command permanently deletes the database and its schema objects. However, it doesn't **guarantee immediate disk space recovery** on the operating system level.

✅ Disk space may remain occupied after deletion

PostgreSQL may continue holding onto disk space due to:

- Retained WAL files
- Open file descriptors
- Active sessions with locks
- External tablespaces with lingering files

This means **visible database objects are gone**, but physical disk usage might not shrink as expected.

✅ Complete cleanup might require full cluster recreation

If you're aiming for **absolute disk space recovery** — for example, before cloning an environment or releasing a server — you may need to recreate the entire PostgreSQL cluster.

✅ The safest approach: dump → drop → reinitialize → restore

Use the following sequence for a full reset:

1. **`pg_dumpall`** to back up all databases
2. **Delete or archive** the current cluster data directory
3. **Reinitialize** PostgreSQL with `initdb`

4. **Restore** your databases from the dump

This ensures a clean slate and maximum space recovery.

✅ Continuously monitor key storage areas

Even if you're not dropping databases often, keep an eye on the following:

- **WAL file growth** (`pg_wal`)
- **Open deleted files** (`lsof | grep deleted`)
- **Active sessions** (`pg_stat_activity`)
- **External tablespaces** (`pg_tablespace_location()`)

These are the usual suspects behind unexpected disk usage in PostgreSQL environments.

💡 Pro Tip: Make Cleanups Smarter and Safer

Avoid manual interventions that can lead to inconsistency or data loss. Instead, adopt these professional best practices:

✅ Automate full cluster cleanups during low-usage windows

When planning for large cleanup operations or environment resets, schedule them during off-peak hours and automate the workflow using shell scripts or Ansible playbooks. This minimizes disruption and human error.

✅ Validate WAL and tablespace configurations before migrations

Before any environment migration, clone, or backup strategy, audit your WAL retention and tablespace locations. Misconfigured parameters can silently consume disk, complicating your cleanup or migration.

✅ Don't rely solely on `DROP DATABASE` for disk recovery

While `DROP DATABASE` clears logical objects, it's not a silver bullet for physical space reclamation. Use it as part of a broader cleanup strategy that includes file-level checks and cluster-wide resets when needed.

By understanding PostgreSQL's internal mechanics and following structured cleanup practices, you'll ensure your database environments remain clean, efficient, and production-ready.

👉 *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 | Data | Oracle | Mongodb

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

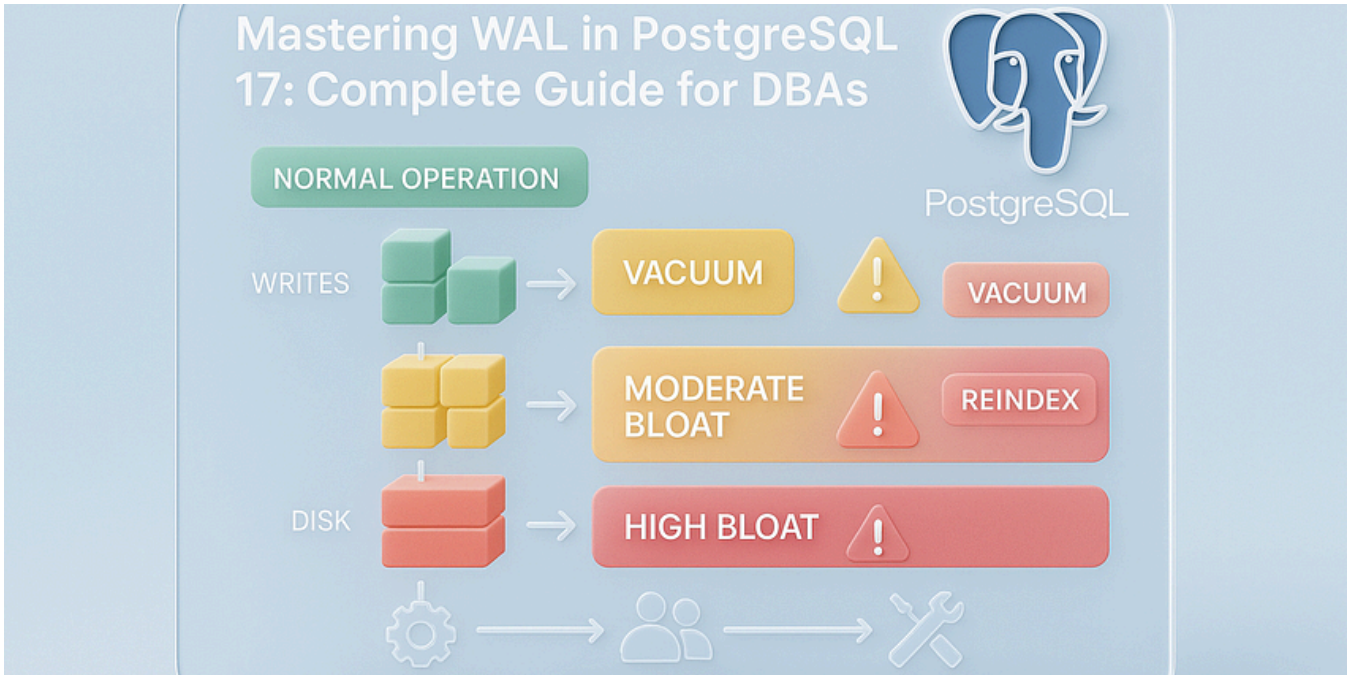
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

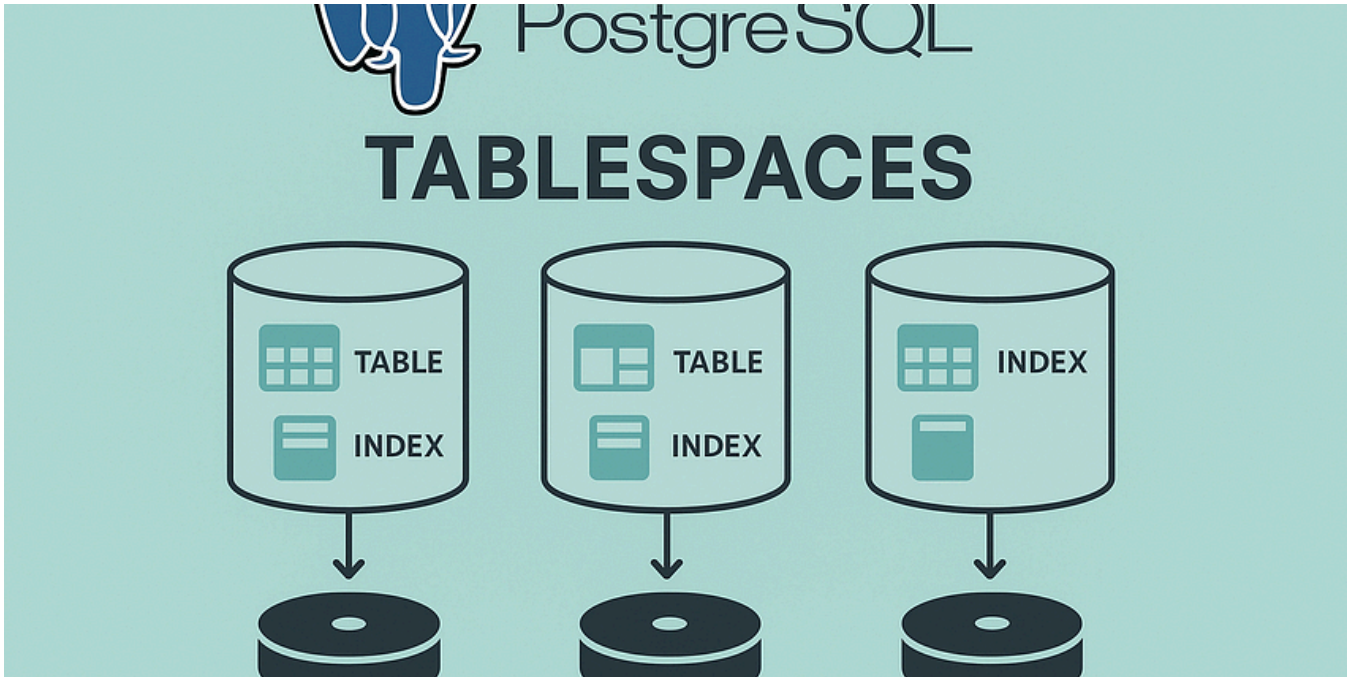


Jeyaram Ayyalusamy

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 52

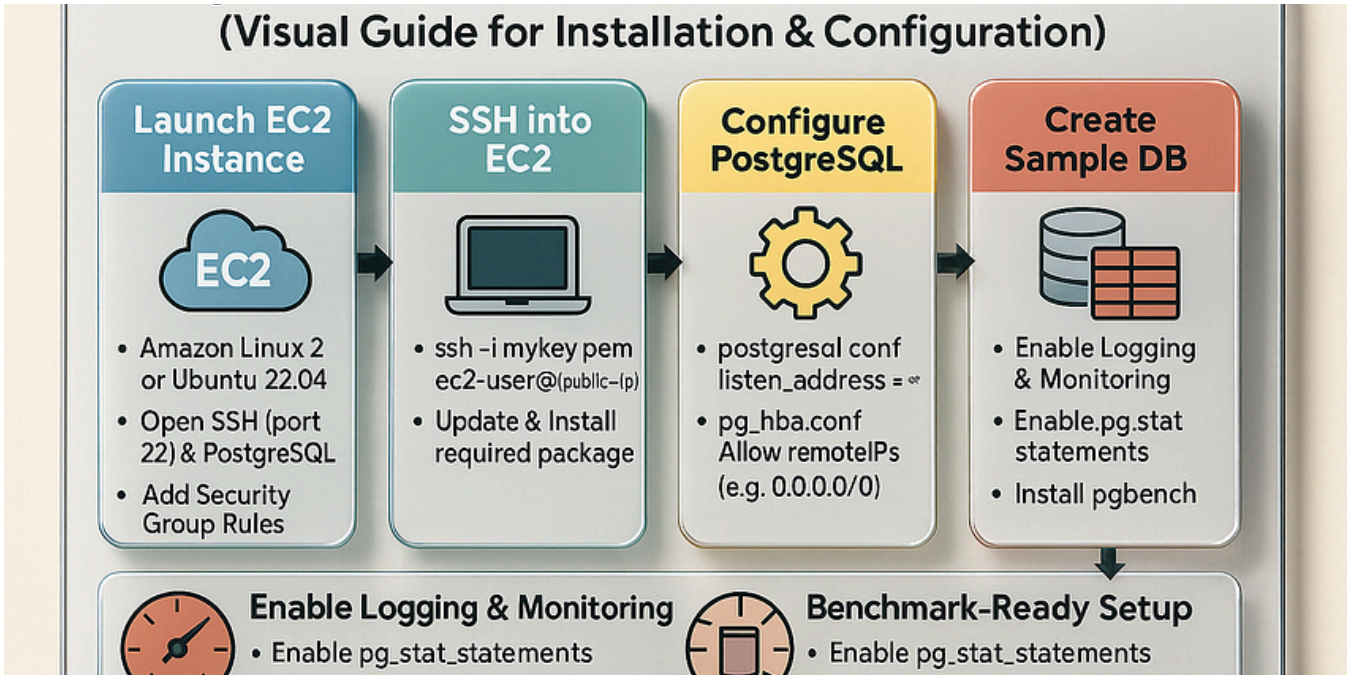


Jeyaram Ayyalusamy

PostgreSQL Tablespaces Explained: Complete Guide for PostgreSQL 17 DBAs

PostgreSQL offers a powerful feature called tablespaces, allowing database administrators to take control of how and where data is...

Jun 12 8



Jeyaram Ayyalusamy

PostgreSQL 17 on AWS EC2—Full Installation & Configuration Walkthrough

Setting up PostgreSQL 17 on AWS EC2 might seem complex at first—but once you break down each component, it becomes an efficient and...

1d ago 50

Jeyaram Ayyalusamy

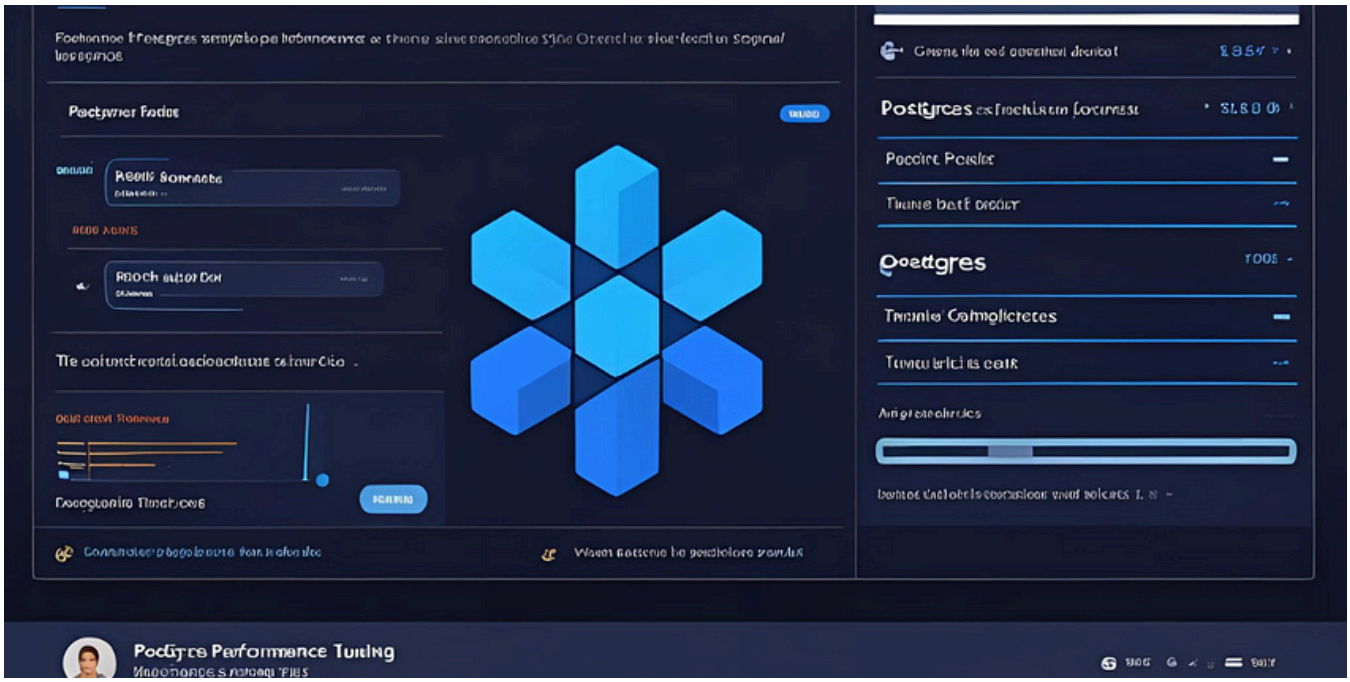
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

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



Rizqi Mulki

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55

Azlan Jamal

Stop Using SERIAL in PostgreSQL: Here's What You Should Do Instead

In PostgreSQL, there are several ways to create a PRIMARY KEY for an id column, and they usually involve different ways of generating the...

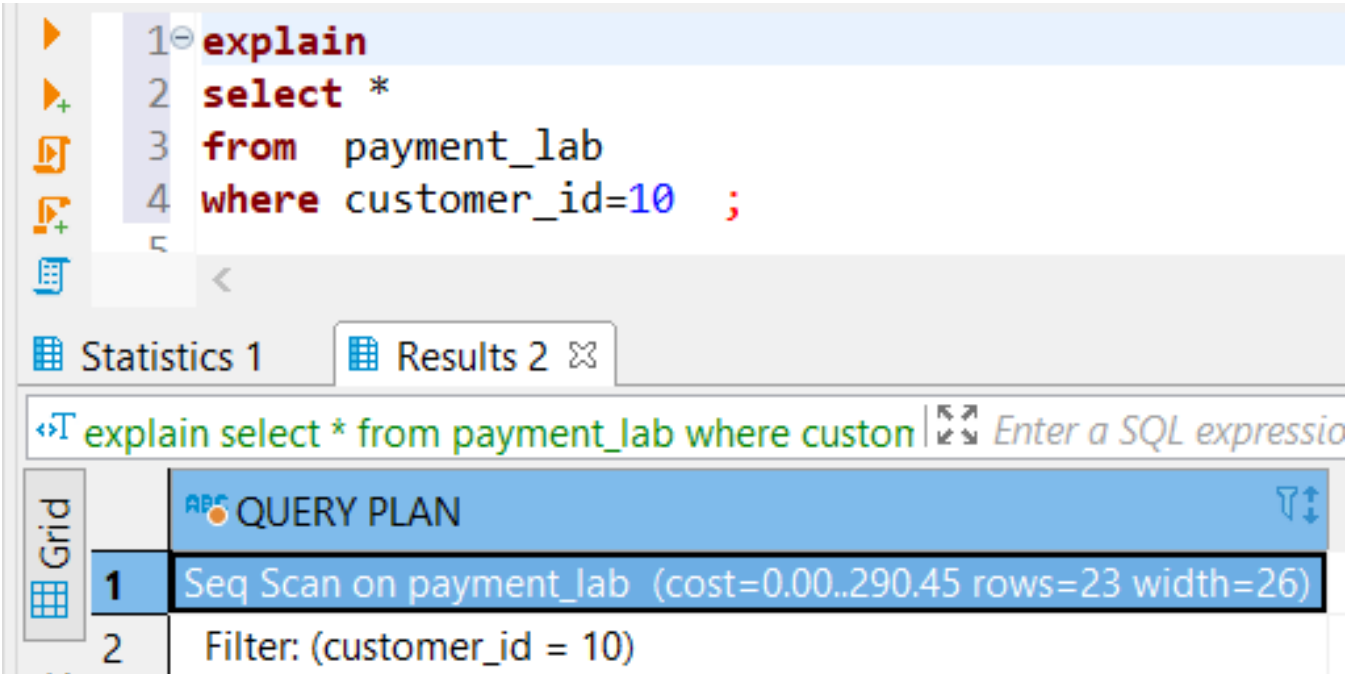
Jul 12 33

techWithNeeru

This SQL Query Took 20 Seconds. Now It Takes 20ms. Here's What I Changed

Last week, I was troubleshooting a performance issue that was driving our team crazy. Our analytics dashboard was timing out, users were...

Jul 10 66



Muhammet Kurtoglu

Postgresql Query Performance Analysis

SQL tuning is critically important for ensuring database performance, reliability, and scalability. In most cases, performance issues in a...

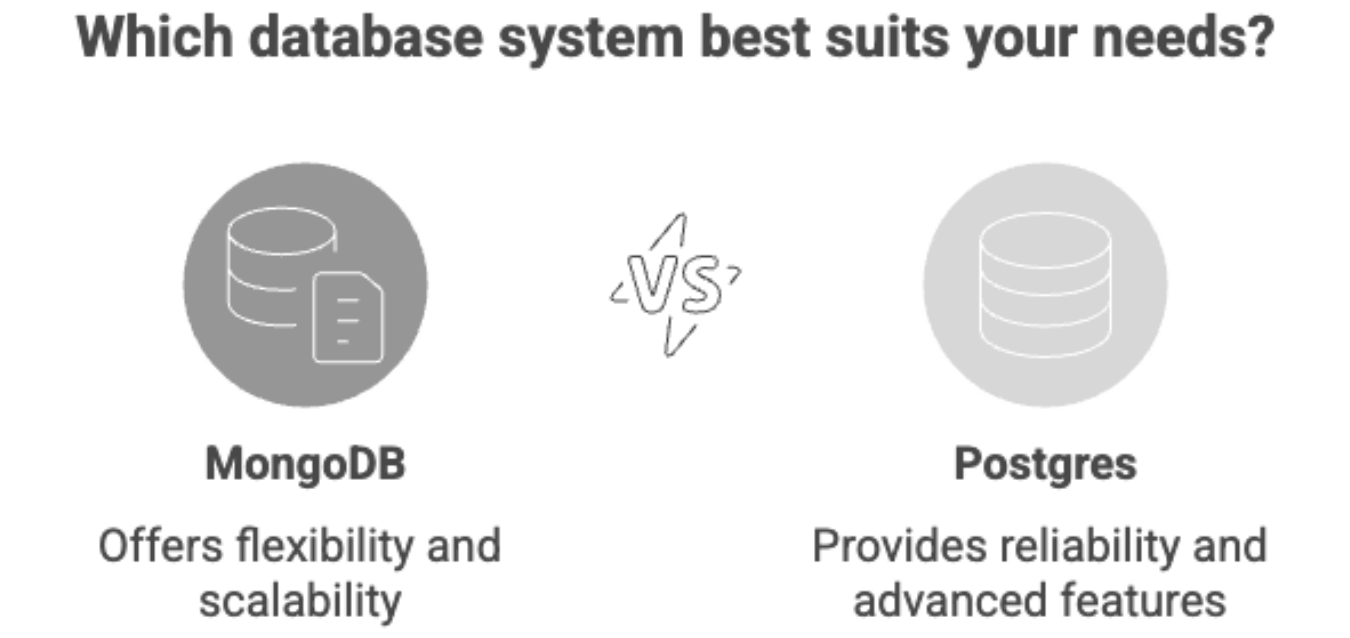
6d ago 10

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



Sachin Kumar

Postgres Killed My Love for MongoDB—And I'm Glad It Did

I used to be a hardcore NoSQL evangelist. MongoDB was my go-to. Schemaless! JSON! Scale! It felt fast. It looked modern. And I swore I'd...

3d ago 5

See more recommendations