Open in app ↗

Medium Search

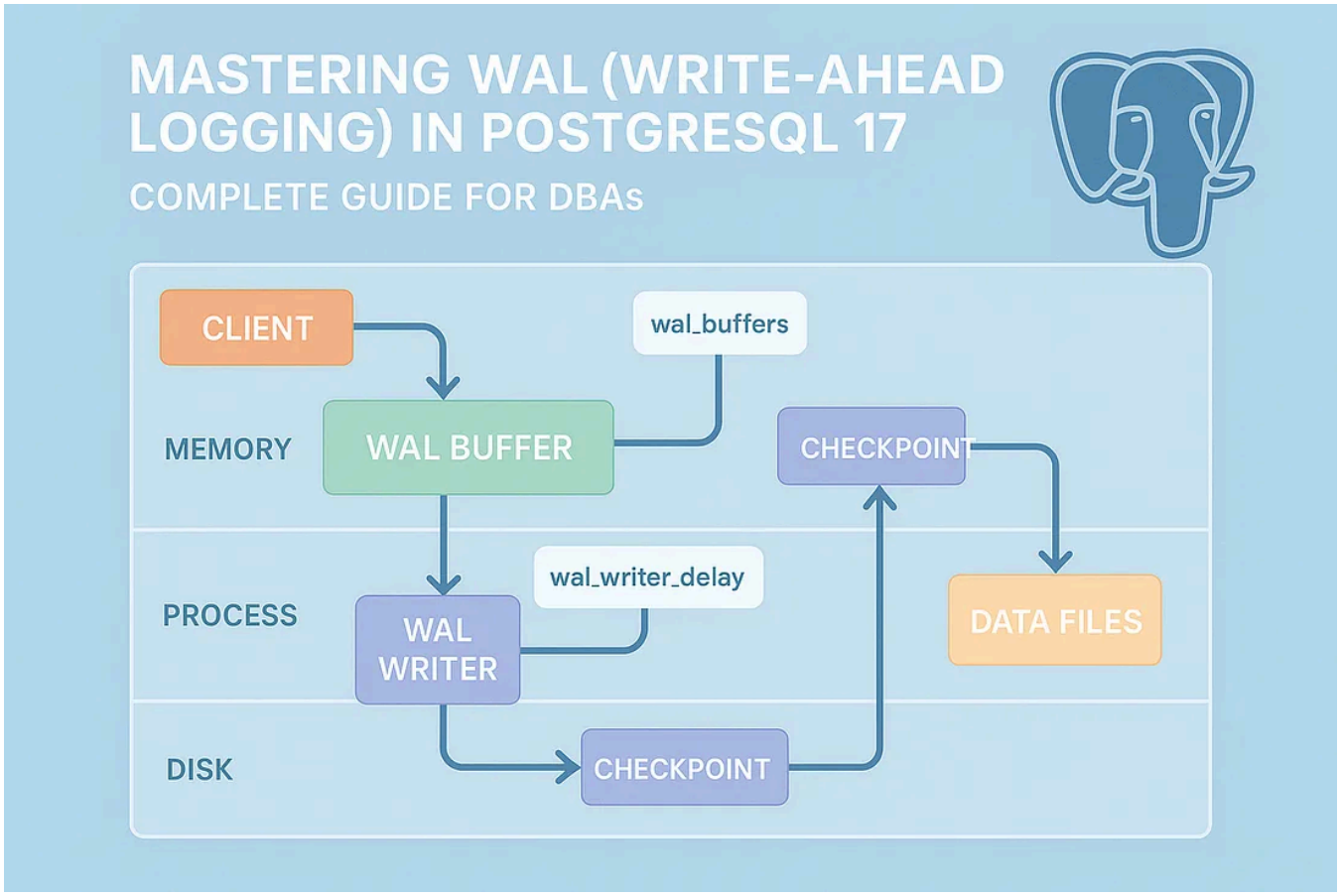
Get unlimited access to the best of Medium for less than $1/week. Become a member

Mastering WAL (Write-Ahead Logging) in PostgreSQL 17: Complete Guide for DBAs

16 min read · Jun 27, 2025

J Jeyaram Ayyalusamy Following

Listen Share More



PostgreSQL's **Write-Ahead Logging (WAL)** is one of its most powerful — and often misunderstood — features. It operates quietly behind the scenes but plays a crucial role in maintaining database integrity and reliability.

At its core, WAL ensures **data durability**, meaning that once a transaction is committed, it will not be lost — even in the event of a system crash. WAL achieves this by writing all changes to a **sequential log file** before they are written to the main data files. This guarantees that changes can always be reconstructed, no matter what.

WAL also **enables crash recovery**. If the database shuts down unexpectedly, PostgreSQL reads the WAL logs on startup and replays any changes that were not yet flushed to the main data files. This automatic recovery process ensures a consistent state without data loss.

Moreover, WAL is the backbone of **replication** in PostgreSQL. Streaming replication and logical replication both rely on WAL to track and transmit changes from the primary to one or more standby servers. This makes high availability and read scalability possible with minimal latency.

Another significant capability that WAL unlocks is **Point-In-Time Recovery (PITR)**. By combining WAL logs with base backups, administrators can restore the database to an exact moment before an error, corruption, or unwanted change occurred. This is invaluable for businesses needing robust disaster recovery strategies.

In PostgreSQL 17, WAL continues to evolve with improved performance, tunability, and compression options. In this article, we'll take a deep dive into how WAL works in PostgreSQL 17, how to configure it, manage it, and monitor its behavior — empowering you to fully leverage one of the database's most critical components.

🔍 What is WAL in PostgreSQL?

PostgreSQL's **Write-Ahead Logging (WAL)** is one of the most critical components in ensuring that your data remains safe, consistent, and recoverable — even in the face of crashes or hardware failures.

At its core, **WAL is a sequential log** of all changes made to the database. Before any data is physically written to the database files, the changes are **first written to the WAL log**. This might seem counterintuitive at first — why write the same data twice? But this simple design provides massive advantages in terms of performance, durability, and reliability.

✅ WAL Guarantees Four Key Capabilities

1. Durability

In database systems, *durability* means that once a transaction is committed, its results must persist, even if the system crashes immediately afterward. PostgreSQL achieves this by writing changes to the WAL before confirming the commit to the client.

Once the WAL record is flushed to disk, the system guarantees that the transaction can be replayed during crash recovery, even if the actual data pages haven't been written yet. This ensures zero data loss for committed transactions.

2. Crash Recovery

If your PostgreSQL server crashes unexpectedly — due to a power failure, kernel panic, or system reboot — WAL becomes the safety net. When the database restarts, PostgreSQL reads the WAL files to identify all the changes that were recorded but not yet applied to the data files.

It then **replays these changes** to bring the database back to a consistent state. This automatic recovery happens without user intervention and is one of the key reasons PostgreSQL is considered enterprise-grade.

3. Replication

PostgreSQL supports **physical (streaming) replication** using WAL. In a replicated setup, the **primary server continuously sends WAL data to the standby server(s)**. These standby servers replay the WAL records in the same order as the primary, keeping them in sync.

This method ensures high availability and horizontal scalability by distributing read workloads across replicas.

4. Point-in-Time Recovery (PITR)

One of the most powerful recovery features PostgreSQL offers is **Point-in-Time Recovery**. This allows you to restore the database to **an exact moment in time**, down to the second.

Using a combination of a base backup and archived WAL files, you can undo mistakes like accidental `DROP TABLE` commands or data corruption. PITR is especially valuable in regulated industries like finance or healthcare, where data integrity and traceability are paramount.

> **In Summary:** WAL is *not optional* in PostgreSQL — it's a built-in, non-negotiable part of how the database works. It ensures that every transaction is durable, recoverable, and replicable. Without WAL, PostgreSQL could not guarantee ACID compliance.

📦 Where Are WAL Files Stored?

WAL files aren't scattered throughout your system — they're neatly organized in a specific directory under your PostgreSQL data directory.

🗂️ Default Location

By default, WAL files are stored here:

```
<data_directory>/pg_wal
```

This is an internal subdirectory that PostgreSQL manages for all ongoing WAL segments.

You can check your actual PostgreSQL data directory by running:

```
SHOW data_directory;
```

💡 Example Output

```
     data_directory
------------------------
 /var/lib/pgsql/17/data
(1 row)

postgres=#
```

So the full path to WAL files would be:

```
/var/lib/postgresql/17/main/pg_wal
```

This directory contains the **WAL segment files** — binary logs that capture every change made to the database. Each segment is typically 16MB by default, and PostgreSQL cycles through them over time.

🧠 What Happens Inside `pg_wal`?

- WAL segments are created sequentially and named with hexadecimal identifiers.
- Active WAL files are constantly written to during normal operation.
- Old WAL files are either recycled (reused) or archived, depending on your settings.

This directory is **mission-critical**. If you lose the contents of `pg_wal`, you risk losing committed transactions — even if your data files seem intact. That's why it's essential to:

- Monitor disk space in this directory,
- Archive WAL segments properly for backup and PITR,
- Protect it with proper file system permissions and replication.

✅ Final Thoughts

WAL is often invisible to the average PostgreSQL user, but it's the *engine room* of the database. It powers everything from transactional integrity to replication and recovery. Understanding where WAL files live, how they work, and why they matter will make you a more effective and confident PostgreSQL administrator or developer.

⚙️ Key WAL Configuration Parameters in PostgreSQL: Explained

PostgreSQL's **Write-Ahead Logging (WAL)** system is designed to ensure data durability, crash recovery, replication, and point-in-time recovery. But to get the most out of WAL, database administrators (DBAs) must understand how to configure it effectively.

This section dives deep into the most important WAL-related settings found in the `postgresql.conf` file. These parameters govern how much WAL data PostgreSQL retains, where it's stored, how often it's archived, and how replication systems interact with it.

Let's explore each one in detail.

1️⃣ `wal_keep_segments` (Deprecated) → `wal_keep_size` (Modern)

In PostgreSQL versions prior to 13, the `wal_keep_segments` parameter was used to control **how many old WAL segment files to retain** in the `pg_wal` directory, mainly to support streaming replication.

What's changed?

Starting from **PostgreSQL 13**, `wal_keep_segments` was replaced with a more intuitive setting: `wal_keep_size`.

```
wal_keep_size = '160MB'
```

This means PostgreSQL will keep at least **160 megabytes worth of WAL files** in the `pg_wal` directory, even if those files are no longer needed for crash recovery. This retained size is useful for **replication scenarios**, where standby servers might lag behind the primary. If WAL files are deleted too soon, the standby will fall out of sync and need to reinitialize, which is expensive.

> ✅ Practical Tip: With the default WAL segment size of 16MB, 160MB corresponds to roughly **10 WAL files** retained for standby servers.

If you're running a streaming replica and want to avoid replication interruptions, ensure `wal_keep_size` is sized based on network latency and replica lag.

2️⃣ `max_wal_size`

This parameter defines the **maximum size** that WAL data can grow to on disk **before PostgreSQL triggers a checkpoint.**

```
max_wal_size = '1GB'
```

A **checkpoint** is a critical process that writes all dirty pages (modified data in memory) to disk. Checkpoints help maintain consistency and control the size of WAL files.

Setting a larger `max_wal_size` allows PostgreSQL to **delay checkpoints**, which reduces disk I/O during periods of heavy write activity. However, the downside is that **crash recovery may take longer**, since more WAL data will need to be replayed.

> ⚠️ Balancing act: A higher value improves performance during peak loads but increases recovery time. A lower value speeds up crash recovery but may increase disk I/O during normal operations.

3️⃣ `archive_mode`

WAL files aren't just used for crash recovery and replication — they're also essential for **backups and point-in-time recovery (PITR).** To enable these features, you must **turn on archiving.**

```
archive_mode = on
```

When `archive_mode` is enabled, PostgreSQL begins **archiving completed WAL segments.** However, this setting alone does nothing unless paired with a proper `archive_command`.

> ✅ Note: This setting is essential if you're implementing a disaster recovery strategy using PITR or storing backups on external storage (e.g., NFS, S3, tape backup systems).

4️⃣ `archive_command`

The `archive_command` tells PostgreSQL **how to store completed WAL files.** It runs every time a WAL file is ready for archiving — typically after it's been filled or when a timeout occurs (see `archive_timeout` below).

```
archive_command = 'cp %p /mnt/nfs/%f'
```

- `%p` : Full path to the WAL segment in `pg_wal`
- `%f` : Just the filename of the WAL segment

This command **copies the WAL file to an NFS-mounted storage directory**, which could then be picked up by a backup tool or stored offsite for compliance purposes.

> 🧠 Best Practice: Validate the success of your archive commands. If `archive_command` fails, WAL files pile up in `pg_wal` , eventually filling your disk. Use logging, monitoring, and integrity checks in production.

5️⃣ `archive_timeout`

WAL files are typically archived only when they're full (16MB by default). But what if your database is mostly idle or experiences low write volume?

That's where `archive_timeout` helps.

```
archive_timeout = '1h'
```

This forces PostgreSQL to **switch and archive a WAL file at least once every hour**, even if the file isn't full. This is crucial in environments where you want **regular and predictable WAL archiving**, such as:

- Backup tools that expect frequent WAL segments
- Standby servers that need a steady stream of WAL files
- PITR systems that rely on consistent archiving intervals

> 🔁 Example: If you set `archive_timeout = '10min'`, you're guaranteed a new WAL file at least every 10 minutes, regardless of write activity.

🔍 Why These Parameters Matter

These configuration settings directly affect:

- 📦 **Disk usage**: Uncontrolled WAL growth can fill up your storage.
- 🔄 **Replication reliability**: Incorrect `wal_keep_size` or archive delays can break streaming replication.
- 🛡️ **Backup consistency**: Without `archive_mode` and `archive_command`, PITR and logical backups may fail.
- ⚡ **Performance tuning**: `max_wal_size` and checkpoints impact throughput and recovery time.

Tuning them properly allows you to balance **performance, safety, and scalability**.

✅ Final Thoughts

PostgreSQL gives DBAs the tools to finely tune how WAL behaves — from how much history is retained to how logs are archived. By understanding and configuring these parameters, you can design systems that are:

- Resilient to failures

- Replication-friendly
- Efficient under heavy loads
- Ready for point-in-time recovery and backups

In the next part, we'll cover how to **monitor WAL activity**, clean up old segments, and prevent storage overload using built-in PostgreSQL tools and scripts.

📁 How to Set Up WAL Archiving in PostgreSQL (Step-by-Step Guide)

In a production PostgreSQL environment, it's not enough to rely solely on crash recovery or periodic backups. To achieve **enterprise-grade disaster recovery** and support **Point-in-Time Recovery (PITR)**, you need to **archive your WAL (Write-Ahead Log) files.**

Archiving WAL files allows you to **capture the history of all transactions**, even after they've been applied to the database. With archived WALs and base backups, you can rewind your database to **any exact point in time** — making it one of the most powerful recovery tools PostgreSQL offers.

Let's walk through **how to set up WAL archiving** with practical commands and real-world explanations.

✅ Step 1 — Enable WAL Archiving

To begin archiving WAL files, you must tell PostgreSQL to enter **archive mode.** This is a fundamental switch that allows the system to archive each completed WAL segment.

Execute the following SQL command:

```
ALTER SYSTEM SET archive_mode = on;
```

This change modifies the `postgresql.auto.conf` file, making the setting persistent across reboots. However, **a restart is required** for the change to take effect, because `archive_mode` is a **postmaster (startup) parameter.**

> 🔍 **Why enable archive mode?** Without it, PostgreSQL won't even attempt to archive WAL segments — making PITR and long-term log retention impossible.

✅ Step 2 — Define `archive_command`

Once archiving is enabled, you must specify how PostgreSQL should copy each completed WAL file to a safe location. This is done using the `archive_command` setting.

```
ALTER SYSTEM SET archive_command = 'cp %p /postgres/logs/archive_wal/%f';
```

Breakdown:

- `%p` : The **full path** of the WAL file that PostgreSQL is archiving.
- `%f` : The **filename only**, used for naming the archived copy.

This example command copies the WAL file to a local path on disk (`/postgres/logs/archive_wal`). You could also modify the command to:

- Copy to NFS/Samba/CIFS shared folders
- Upload to AWS S3 (using tools like `aws s3 cp`)
- Push to a remote server using `scp`

> ⚠️ **Important:** If the `archive_command` fails (returns non-zero), PostgreSQL will **retry indefinitely** and not delete the WAL file, which may eventually fill the disk.

✅ Step 3 — Create the Archive Directory

PostgreSQL doesn't automatically create the destination directory. You need to do this manually and ensure the database has the correct permissions.

Run the following commands on your server:

```
sudo mkdir -p /postgres/logs/archive_wal
sudo chown postgres:postgres /postgres/logs/archive_wal
```

This ensures:

- The folder exists (`-p` means create parent folders if needed)
- The `postgres` user (under which PostgreSQL runs) has ownership and write access

> 💡 You can also mount an NFS share or external disk at this path for safer long-term storage.

✅ Step 4 — Restart PostgreSQL

After setting `archive_mode` and `archive_command` , you **must restart the PostgreSQL service** for the changes to take effect.

Use the following system command (on Linux systems using `systemd`):

```
sudo systemctl restart postgresql-17
```

You can confirm the service restarted successfully by checking its status:

```
sudo systemctl status postgresql-17
```

> 🚀 Once restarted, PostgreSQL will begin archiving completed WAL segments based on your defined `archive_command`.

✅ Step 5 — Verify WAL Configuration

After restarting, it's critical to verify that your WAL archiving configuration is correct and active. Use these SQL commands:

```
SHOW wal_level;
SHOW max_wal_size;
SHOW archive_mode;
SHOW archive_command;
SHOW archive_timeout;
```

What to look for:

- `wal_level` should be at least `replica` (or `logical` if you're using logical replication)
- `archive_mode` should return `on`
- `archive_command` should show your `cp` or custom command
- `archive_timeout` (if configured) ensures regular WAL rotation

If everything looks good, PostgreSQL will now safely store every completed WAL segment — enabling PITR and supporting incremental backup strategies.

🔄 Manually Switching WAL Files

There are situations where you may want to **force PostgreSQL to close the current WAL file** and start writing to a new one — even if the current file isn't full. This is often useful:

- Just before taking a physical or logical backup

- After large batch jobs or data loads
- To test archiving and replication setups

Use the following SQL function:

```
SELECT pg_switch_wal();
```

> 📌 **What it does:** This command instructs PostgreSQL to immediately switch to a new WAL segment. The previous one is then marked ready for archiving and copied according to the `archive_command`.

Practical Use Case:

If you're using `pg_basebackup` to take a base backup and want to ensure all WAL segments are archived correctly, call `pg_switch_wal()` **just before or after the backup** to capture all recent transactions.

🧩 Final Thoughts

WAL archiving is not just a backup feature — it's a **lifeline** for production databases. Whether you're building high-availability architectures or need the ability to roll back to a specific time (PITR), proper WAL archiving is essential.

By following these steps:

- You ensure **recoverability** in the event of corruption, mistakes, or deletion.
- You enable **incremental backup workflows** using tools like `pgBackRest` or `Barman`.
- You reduce the risk of replication lag or failure by keeping a clean, consistent archive of past WAL activity.

📊 Deep Dive into WAL Monitoring, PITR, and Best Practices in PostgreSQL

PostgreSQL's Write-Ahead Logging (WAL) mechanism plays a foundational role in ensuring data durability, crash recovery, and replication. However, **without proper monitoring and management, WAL can become a liability** — consuming disk space, slowing down replication, and putting your backups at risk.

In this post, we'll take a deep look into:

- How to monitor WAL activity
- How WAL enables Point-in-Time Recovery (PITR)
- The difference between WAL archiving and replication slots
- What happens when WAL is mismanaged

Let's explore each of these in detail.

📊 Monitoring WAL Activity in PostgreSQL

Monitoring WAL activity is not just good practice — it's essential for keeping your PostgreSQL instance healthy, especially in production.

Unchecked WAL file growth can:

- Fill up disk space on the primary server
- Cause standby replication lag or failure
- Lead to missed backups or incomplete PITR coverage
- Affect overall database performance

PostgreSQL provides built-in tools to help monitor WAL in real time.

🔎 1. Check Current WAL Write Pointer

You can monitor the current WAL position (also called LSN — Log Sequence Number) using the following SQL command:

```
SELECT pg_current_wal_lsn();
```

This shows the **current WAL write pointer** — i.e., the point at which PostgreSQL is writing new changes to the WAL log. As transactions occur, this value increases steadily.

Monitoring this value over time gives you an understanding of:

- How much WAL your system generates (WAL write rate)
- Whether spikes in activity are occurring (e.g., bulk inserts)
- The need to adjust WAL size or checkpointing frequency

> 💡 Combine this with a monitoring tool like **Prometheus + Grafana**, **pg_stat_monitor**, or **pgBadger** to visualize WAL activity trends.

🔎 2. Check WAL Replay Status on Standby

On a standby (replica) server, you can measure how far behind it is compared to the primary by checking the WAL replay location:

```
SELECT pg_last_wal_replay_lsn();
```

This returns the LSN of the **last WAL record successfully replayed on the standby**. By comparing this with the primary's `pg_current_wal_lsn()`, you can calculate **replication lag**.

> 📉 A growing gap between write and replay LSN indicates slow replication. This may be due to disk I/O bottlenecks, slow network, or underpowered standby hardware.

🧪 Data Recovery & PITR with WAL

WAL doesn't just support crash recovery or replication — it also unlocks one of PostgreSQL's most powerful recovery features: **Point-in-Time Recovery (PITR).**

🔄 What Is PITR?

Point-in-Time Recovery allows you to **rewind your entire PostgreSQL database to a precise moment in the past.** This is invaluable when:

- Someone accidentally drops a critical table
- A bad migration corrupts data
- An application bug introduces invalid values
- You need to meet **audit or regulatory** requirements to restore data as it was at a specific time

To enable PITR, you must:

1. Take regular **base backups** of your database (e.g., with `pg_basebackup`, `pgBackRest`, or `Barman`)
2. Continuously **archive WAL files** using `archive_mode` and `archive_command`

During PITR:

- PostgreSQL restores the base backup
- Replays archived WAL files **up to a specified timestamp or transaction**
- Stops right before the damaging operation occurred

> 🛡️ PITR provides **granular, second-by-second recovery**, making it a must-have feature for production databases with strict uptime, data protection, and audit requirements.

🚩 WAL Archiving vs. Replication Slots

PostgreSQL supports two different strategies for retaining WAL data — **WAL Archiving** and **Replication Slots**. While they serve different purposes, they can (and often do) coexist.

🗂️ WAL Archiving

- **Purpose:** Store completed WAL segments on disk or external storage
- **Used for:**
- Backups
- PITR (Point-in-Time Recovery)
- Long-term retention
- **Mechanism:** PostgreSQL copies WAL files to a specified location using `archive_command`

> ✅ Best for disaster recovery and restoring to any point between backups

🔌 Replication Slots

- **Purpose:** Ensure a **streaming replica never misses WAL records**, even if it temporarily goes offline
- **Used for:**
- Streaming replication setups (physical or logical)
- High availability clusters
- **Mechanism:** PostgreSQL keeps WAL files until the replica acknowledges receipt — controlled by a replication slot

> 🛑 **Caution:** If a replication slot is active but the replica is down, WAL files will accumulate **indefinitely**, potentially filling the disk.

✅ Can You Use Both?

Yes, and in many setups you should. For example:

- Use **replication slots** to ensure fast, near real-time streaming to hot standbys
- Use **WAL archiving** to support PITR, full backups, and disaster recovery

These two methods **complement each other** in high-availability and compliance-driven environments.

🛑 WAL Mismanagement = Serious Trouble

If not managed properly, WAL can go from PostgreSQL's strongest feature to a system-breaking problem. Here's how:

⚠️ Disk Space Consumption

If you:

- Enable `archive_mode` but fail to configure `archive_command` correctly, or
- Create replication slots for replicas that go offline and never reconnect

… then WAL files will accumulate indefinitely in the `pg_wal` directory, quickly consuming all available disk space. Once full, PostgreSQL **cannot write new transactions**, effectively halting all write operations.

⚠️ Frequent WAL Switches

WAL files are typically 16MB in size. If your database switches WAL files too often, it might indicate:

- Excessive write volume (e.g., bulk loads, unindexed logging)
- Aggressive checkpoint settings (e.g., low `max_wal_size`)
- Poor autovacuum tuning

Frequent WAL switches can stress disk I/O and slow down backups or replication.

⚠️ Manual Deletion of WAL Files

Never manually delete files in the `pg_wal` directory — even if disk space is running low. Doing so can:

- Break replication
- Corrupt crash recovery
- Make PITR impossible
- Require full database reinitialization

> ❗ Instead, fix the root cause: configure archiving correctly, manage replication slots, and monitor WAL growth.

✅ Final Thoughts

PostgreSQL's WAL system is one of the database's greatest strengths — but only when managed wisely.

Here's a quick recap:

- Monitor WAL LSNs (`pg_current_wal_lsn` , `pg_last_wal_replay_lsn`) regularly
- Set up archiving for PITR and regulatory recovery
- Understand and balance the roles of WAL archiving vs replication slots
- Never manually delete WAL files — always use supported tools

By proactively managing WAL, you can build a **resilient, recoverable, and high-performing PostgreSQL environment.**

🏁 Conclusion

PostgreSQL has earned its reputation as one of the most reliable open-source relational databases in the world — and **Write-Ahead Logging (WAL)** is one of the core reasons why.

WAL acts as the **lifeline of durability**, ensuring that no committed transaction is ever lost, even in the event of a system crash or power failure. But WAL isn't just about crash recovery — it also empowers **streaming replication**, **disaster recovery**, and **Point-in-Time Recovery (PITR)**, all of which are essential for running PostgreSQL in real-world production environments.

Understanding how WAL works is the first step. But more importantly, **proper configuration and ongoing management** are what ensure WAL helps — not harms — your infrastructure. This means:

- Enabling WAL archiving and setting up `archive_command` correctly
- Defining retention strategies through `wal_keep_size` and checkpoints
- Monitoring LSN activity and replication lag to avoid data loss or drift
- Avoiding common pitfalls, like manual deletion or unmonitored replication slots

> 💡 Whether your goal is simple data durability or a fully redundant high-availability (HA) architecture, mastering WAL is **non-negotiable** for every serious PostgreSQL DBA, DevOps engineer, or platform architect.

By treating WAL not just as a background feature, but as a **core design consideration**, you can build a PostgreSQL environment that is not only performant, but also resilient, secure, and ready for recovery at any time.

📢 Stay Updated with Daily PostgreSQL & Cloud Tips!

If you've been finding my blog posts helpful and want to stay ahead with daily insights on PostgreSQL, Cloud Infrastructure, Performance Tuning, and DBA Best Practices — I invite you to subscribe to my Medium account.

🔔 Subscribe here 👉 https://medium.com/@jramcloud1/subscribe

Your support means a lot — and you'll never miss a practical guide again!

🔗 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 | AWS | Open Source

Following

Written by Jeyaram Ayyalusamy

62 followers · 2 following

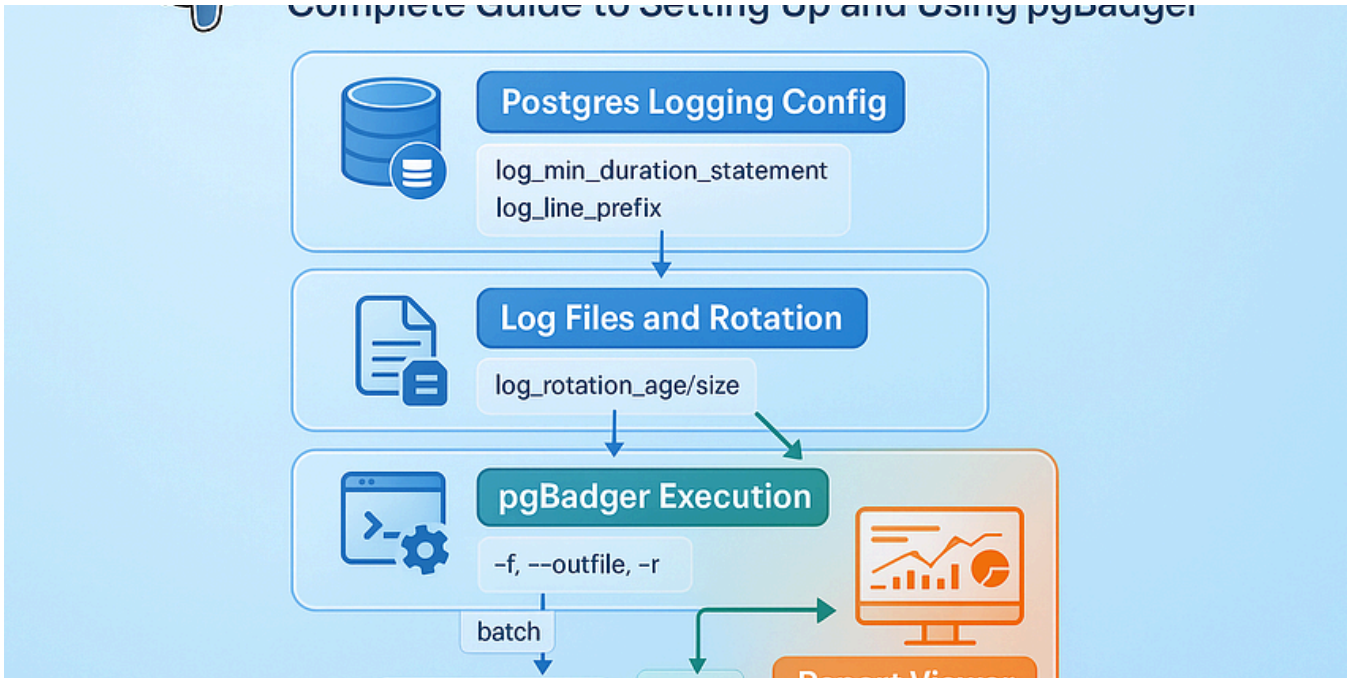
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

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...

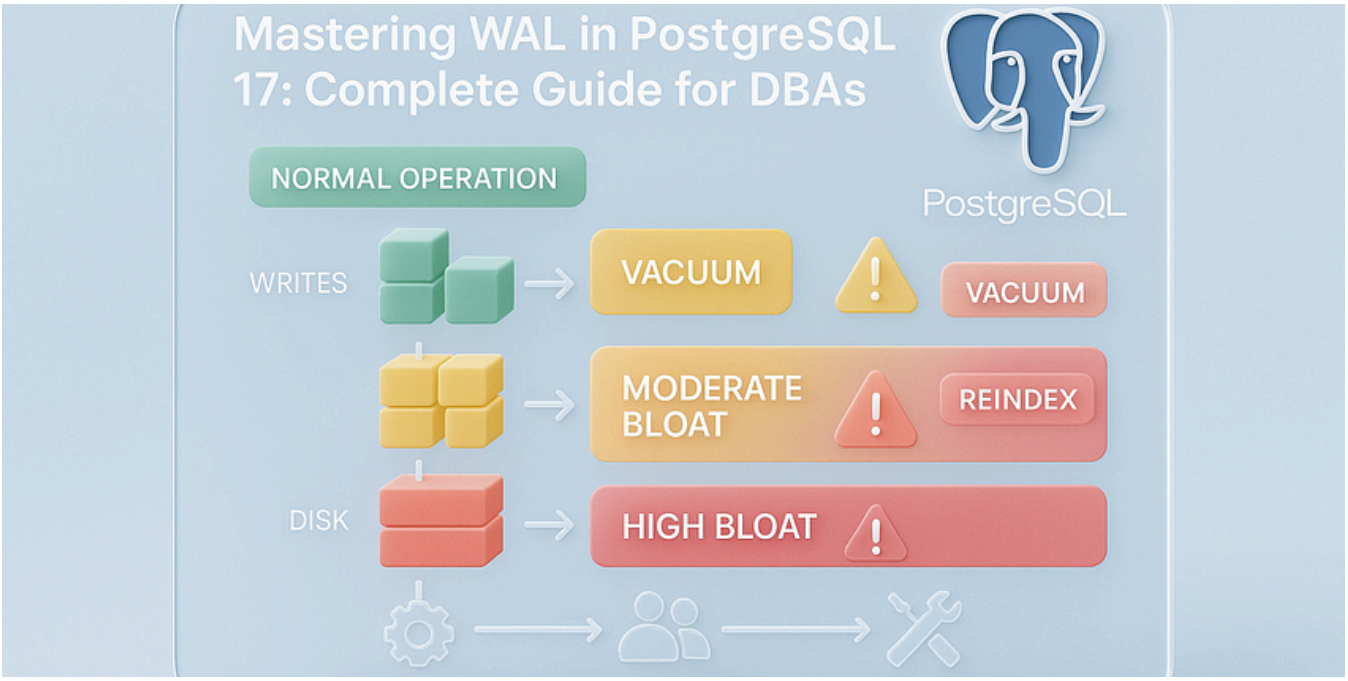
Jun 23 52

Jeyaram Ayyalusamy

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

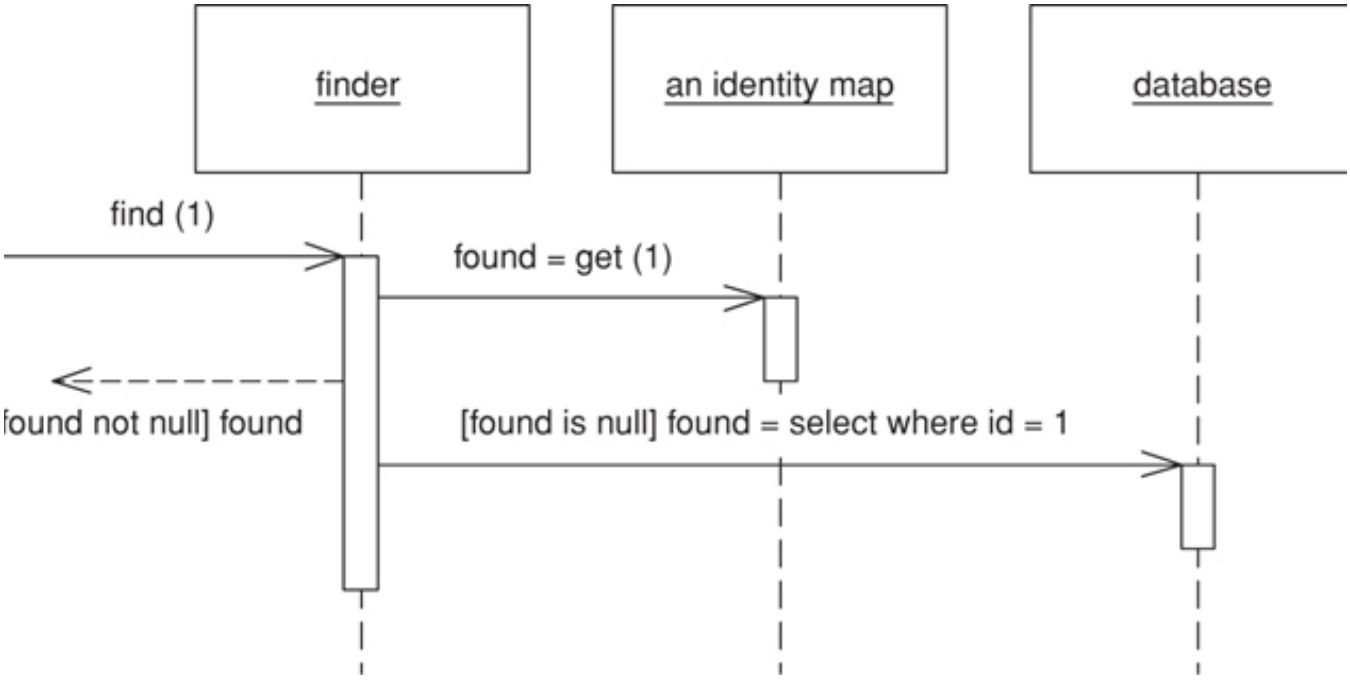
Recommended from Medium

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

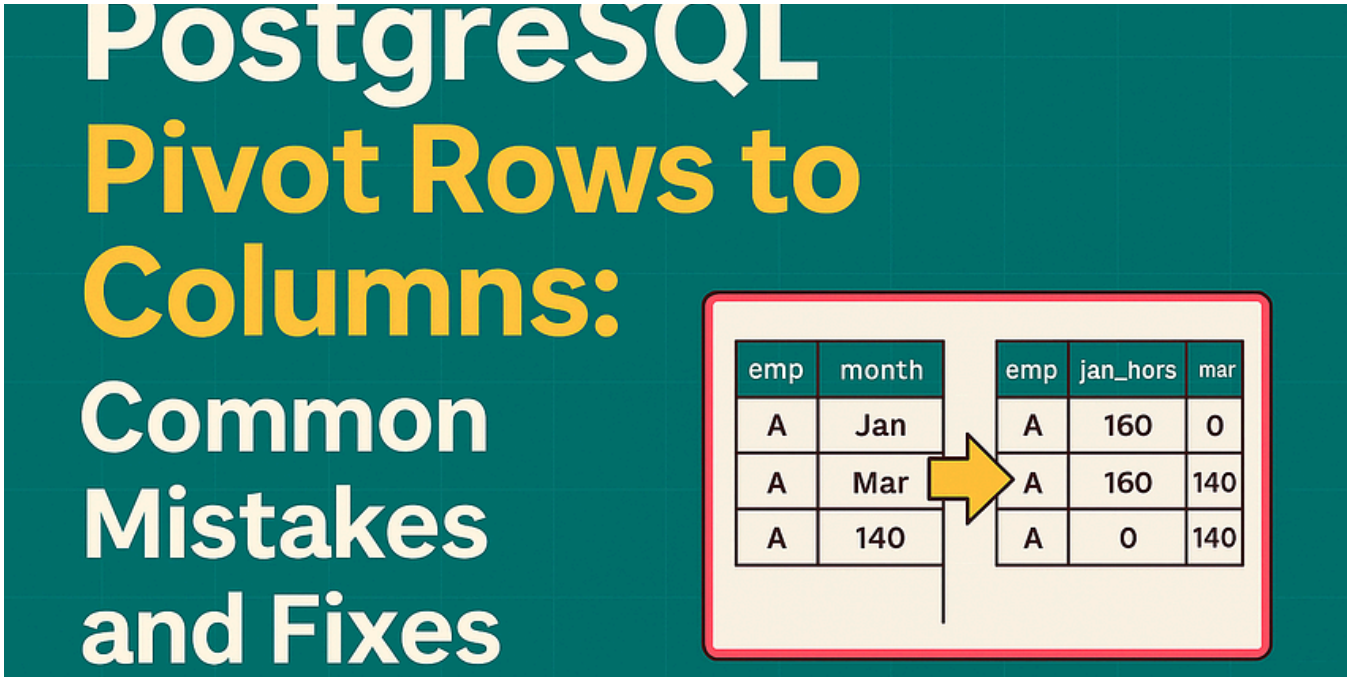


Harishsingh

The Forgotten SQL Pattern That Reduced Our Query Time by 90%

Like many teams, we assumed our SQL queries were “good enough” because they returned the correct results. But when a simple dashboard…

6d ago 4



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 25 793 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

In AlgoMart by Yash Jain

SQL Server vs MySQL vs PostgreSQL—Picking the Right DB Like a Dev

If you've ever had to choose between SQL Server, MySQL, and PostgreSQL, you know it's not just about syntax or what the job post says. It's...

Jun 26 43

See more recommendations