# **How to Clean Up WAL Files and Replication Slots in PostgreSQL (Complete Guide for DBAs)**

[](https://medium.com/@jramcloud1?source=post_page---byline--2ca33b3fdb3b---------------------------------------)

[Jeyaram Ayyalusamy](https://medium.com/@jramcloud1?source=post_page---byline--2ca33b3fdb3b---------------------------------------)

Following

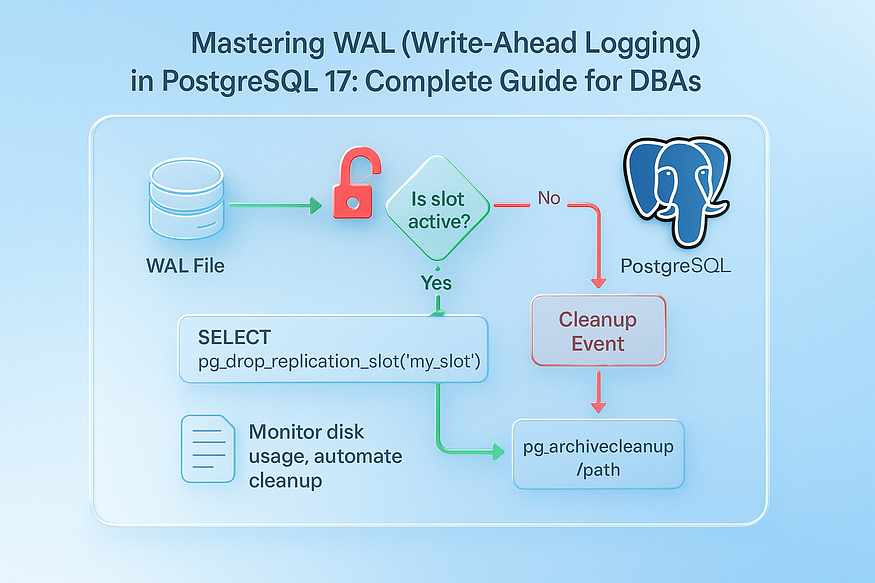
15 min read

·

Jun 27, 2025

13

Press enter or click to view image in full size



PostgreSQL is widely regarded as one of the most powerful and reliable open-source relational database systems. With its advanced feature set and ACID-compliant architecture, it is the backbone of countless enterprise and cloud-native applications. However, this power comes with a caveat: certain internal mechanisms, if left unmanaged, can silently consume disk space and degrade performance.

Two such mechanisms — ****Write-Ahead Logs (WAL)**** and ****replication slots**** — are fundamental to PostgreSQL’s durability and replication capabilities. But without proper attention, they can result in unanticipated disk usage and performance bottlenecks.

## **Why Should You Care About WAL Files and Replication Slots?**

In PostgreSQL, many of the core features that make it robust and reliable — such as durability, crash recovery, and high availability — depend on internal mechanisms working silently in the background. Two of the most critical components in this architecture are ****Write-Ahead Log (WAL) files**** and ****replication slots****. While they serve essential purposes, they also have the potential to create serious issues if neglected.

Let’s break down why these two components demand your attention.

## **🔁 WAL Files: Durable by Design, Risky if Ignored**

PostgreSQL uses Write-Ahead Logging (WAL) to guarantee data durability. Every transaction — whether it’s a row insert, update, or delete — is first recorded in a WAL file before being applied to the main database. This ensures that, in the event of a system crash or power failure, the database can be brought back to a consistent state by replaying these logs.

However, under certain conditions, WAL files can accumulate quickly:

* ****Heavy transactional workloads**** generate large volumes of WAL data.
* ****Long-running backups**** prevent WAL files from being removed, since PostgreSQL needs to ensure recoverability throughout the backup duration.
* ****Replication lag**** delays WAL recycling because the primary server must retain WALs until replicas confirm receipt.

Over time, these WAL files can consume substantial disk space — sometimes reaching gigabytes or more in active environments. If not pruned or archived efficiently, they can lead to disk exhaustion and performance degradation.

## **🔄 Replication Slots: Essential for Replication, Hazardous if Abandoned**

Replication slots are PostgreSQL’s way of ensuring that data changes are reliably passed from the primary server to replicas. When a replication slot is active, PostgreSQL retains WAL files until the connected replica has safely received and processed them. This guarantees ****zero data loss**** in streaming replication.

But there’s a catch.

If a replication slot is ****left unused****, or if a replica becomes ****permanently disconnected****, PostgreSQL will continue to retain all corresponding WAL files — indefinitely. The database has no way of knowing whether the replica will reconnect later or not. As a result, disk usage climbs steadily, and WAL recycling is effectively blocked.

In high-availability or cloud environments, this scenario is especially common. A replica may be decommissioned or migrated, but its slot remains registered, causing the primary to hoard WAL files unnecessarily.

## **🧼 Why Cleanup Matters**

Failing to monitor and manage WAL files and replication slots can lead to:

* Unpredictable disk usage spikes
* Slower performance due to I/O bottlenecks
* System outages when the disk fills up
* Difficult recovery situations under pressure

This is why ****regular cleanup and monitoring**** are not optional — they’re essential for database hygiene and long-term operational stability. By understanding how these components work and being proactive in managing them, you can avoid crises and keep your PostgreSQL environment lean, healthy, and performant.

## **🧹 Cleaning Up WAL Files in PostgreSQL**

PostgreSQL’s transaction reliability is largely owed to a foundational mechanism known as ****Write-Ahead Logging**** (WAL). While this system is essential for preserving data integrity, it can also become a silent disk hog if left unmanaged. In this section, we’ll walk through what WAL files are, where they live, and how to examine the settings that govern their behavior — setting the stage for a safe and effective cleanup.

## **📘 1.1 What Are WAL Files?**

To understand PostgreSQL’s resilience, you need to understand ****WAL files****. Write-Ahead Logs are PostgreSQL’s way of ensuring that no committed transaction is ever lost — even if the server crashes.

### **🔍 How WAL Works**

Instead of writing data changes directly to the main data files, PostgreSQL first records ****every change**** in a WAL file. These logs capture the ****intent**** of each transaction — whether it’s an INSERT, UPDATE, DELETE, or even a DDL command like CREATE TABLE. Once logged, the database proceeds to apply the changes to actual data blocks in the background.

This approach provides several benefits:

* ****Crash Recovery:**** If PostgreSQL crashes mid-operation, it can replay WAL logs on restart to restore the database to a consistent state.
* ****Replication:**** Streaming replicas receive and apply WAL changes to stay in sync with the primary.
* ****Point-in-Time Recovery:**** With full WAL archiving, you can restore your database to any specific point in time — useful for recovering from accidental data changes.

### **🗂️ Where Are WAL Files Stored?**

By default, WAL files are stored in the pg\_wal directory, which lives inside your PostgreSQL data directory:

/data\_directory/pg\_wal/

To confirm the exact location of your data directory, you can run the following SQL command in psql:

SHOW data\_directory;

This command will return the full filesystem path where PostgreSQL is storing its internal data, including WAL files. Knowing this is important for any manual inspection or cleanup efforts.

💡 **Tip:** If your disk is filling up and you find large files inside pg\_wal, it’s a sign that WAL retention is being held back — possibly due to replication lag or archiving issues.

## **⚙️ 1.2 Check Current WAL Settings**

Before you take any cleanup actions, it’s essential to understand ****how your PostgreSQL server is currently configured to manage WALs****. A few core settings dictate how WALs are generated, how long they’re kept, and whether they’re archived for backups.

Let’s examine two critical settings: wal\_level and archive\_command.

### **🔹 WAL Level**

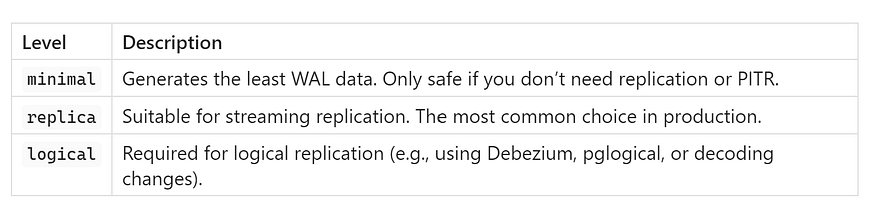
The wal\_level setting controls ****how much information**** PostgreSQL writes into WAL files. This has a direct impact on how large and frequent the WAL logs will be — and also determines what kind of replication or recovery features your database can support.

To check your current WAL level, run:

SHOW wal\_level;

The result will be one of the following:

Press enter or click to view image in full size



Choosing the wrong level can either cause excessive WAL generation or block replication capabilities. For example, setting wal\_level = logical without actually using logical replication creates unnecessary I/O and disk usage.

🛠️ **Pro Tip:** Only use logical if you need advanced replication or change data capture (CDC). Otherwise, stick with replica for typical HA setups.

### **🔹 Archive Command**

When you enable WAL archiving, PostgreSQL uses the archive\_command setting to ****copy completed WAL segments to a safe location**** — often to external storage or a backup server.

Check your current archiving configuration using:

SHOW archive\_command;

The output will be one of two things:

* An actual shell command (e.g., cp %p /mnt/wal\_archive/%f) — which tells PostgreSQL how to copy WAL files once they’re done being written.
* An empty string — which means WAL archiving is ****disabled****.

If WAL archiving is turned on but your archive destination is full or misconfigured, PostgreSQL ****won’t delete old WAL files**** — leading to disk buildup in the pg\_wal directory. This is a common cause of unexpected storage issues.

⚠️ **Watch Out:** If archive\_command is set but failing silently (e.g., the target directory is unavailable), your disk usage will grow without warning. Always verify that the archive directory is writable and monitored.

## **🧭 Summary**

To safely clean up WAL files in PostgreSQL, you must first understand how they are generated and retained. Knowing your data\_directory, wal\_level, and archive\_command settings will give you the clarity needed to make safe adjustments — whether that means dropping unused replication slots, tuning retention settings, or diagnosing an archiving issue.

## **🛠 Managing WAL Retention and Safe Cleanup in PostgreSQL**

PostgreSQL’s Write-Ahead Logging (WAL) is essential for ensuring durability, crash recovery, and replication. However, if not properly monitored and maintained, WAL files can gradually accumulate, leading to disk bloat and performance degradation. In this section, we’ll walk through how PostgreSQL retains WAL files, how to monitor their impact on disk space, and how to clean them up safely using PostgreSQL’s built-in utilities.

## **📏 1.3 Review WAL Retention Policy**

PostgreSQL doesn’t hold onto WAL files forever — it uses an internal ****retention policy**** to determine how many WAL files to keep. This is controlled primarily by two configuration parameters:

### **🔹**min\_wal\_size**and**max\_wal\_size

* ****min\_wal\_size**** defines the ****minimum total size**** of WAL files that PostgreSQL will try to keep, even if they are no longer needed. This ensures that PostgreSQL doesn't have to frequently create new WAL files during busy periods, which can be inefficient.
* ****max\_wal\_size**** defines the ****maximum total size**** of WAL files PostgreSQL will retain before it forces a checkpoint to clean up and recycle old WAL segments.

You can view the current settings directly from SQL:

SHOW max\_wal\_size;  
SHOW min\_wal\_size;

For example, if your max\_wal\_size is set to 1GB, PostgreSQL will attempt to keep WAL files under that limit by triggering checkpoints as needed.

### **💡 Why Adjust This?**

If your pg\_wal directory is consuming too much disk space, and your workload doesn't require keeping that many WAL files, you can reduce max\_wal\_size. This prompts PostgreSQL to clean up older WAL segments more aggressively.

### **🛠 How to Change**max\_wal\_size

To modify this setting:

ALTER SYSTEM SET max\_wal\_size = '2GB';

After making this change, reload the configuration to apply it:

SELECT pg\_reload\_conf();

⚠️ **Important:** Setting max\_wal\_size too low can result in very frequent checkpoints. This might hurt write performance due to increased I/O. Always balance disk space savings with performance considerations.

## **💽 1.4 Monitor Disk Usage**

Before cleaning up, it’s critical to ****assess how much space WAL files are consuming****. On most systems, these files are stored in the pg\_wal directory inside PostgreSQL’s data directory.

Use the df -h command in the shell to check disk space usage:

df -h /var/lib/postgresql/16/main/pg\_wal

Here’s what this command does:

* df -h shows disk space usage in human-readable format.
* Replace /var/lib/postgresql/16/main/pg\_wal with the actual path to your pg\_wal directory, based on your PostgreSQL installation.

### **📊 Interpreting the Output**

You’ll get output like this:

Filesystem Size Used Avail Use% Mounted on  
/dev/sda1 100G 85G 15G 86% /

If your disk usage is high (e.g., above 80%), and the pg\_wal folder is the culprit, it’s time to investigate further and possibly take action to free up space.

🔍 **Tip:** You can also use du -sh inside the data directory to estimate how much space pg\_wal alone is consuming:

du -sh /var/lib/postgresql/16/main/pg\_wal

## **🧹 1.5 Clean Up Using**pg\_archivecleanup

If WAL archiving is enabled in your PostgreSQL server, WAL files are stored not only in pg\_wal, but also in your ****archive directory****. Over time, these archived WAL files can accumulate and consume significant storage. PostgreSQL provides a tool called ****pg\_archivecleanup**** to help you safely delete obsolete archived WAL segments.

### **🔧 What is**pg\_archivecleanup**?**

pg\_archivecleanup is a command-line utility that:

* Scans the archive directory.
* Deletes WAL files older than a specified segment.
* Ensures that WAL files still needed for recovery or replication are preserved.

This is especially useful in backup or disaster recovery pipelines where old WAL files are no longer needed but haven’t been purged.

### **🛠 Example Command**

pg\_archivecleanup -d /postgres/logs/archive\_wal 00000001000000000000001E

Let’s break this down:

* -d /postgres/logs/archive\_wal tells the tool where to look for the archived WAL files.
* 00000001000000000000001E is the ****oldest WAL segment still required****. All files ****older**** than this will be deleted.

You can find the appropriate WAL segment name to use from backup metadata or replication monitoring tools. The format of WAL files includes a timeline and log/segment number, such as 00000001000000000000001E.

⚠️ **Caution:** Never use arbitrary WAL segment names. Make sure the value you supply is safe — i.e., no longer needed for PITR (Point-in-Time Recovery) or replication. Deleting WALs prematurely can break replicas or prevent a full recovery from backups.

## **✅ Final Thoughts**

Efficient management of PostgreSQL WAL files is essential to maintaining a healthy and performant database system. In this section, we’ve covered how to:

* ****Understand and tune WAL retention policies**** using min\_wal\_size and max\_wal\_size,
* ****Monitor disk usage**** using df -h and du,
* And ****safely clean up**** archived WAL files with pg\_archivecleanup.

Together, these practices help you prevent disk overflow, improve stability, and ensure that your PostgreSQL server can recover and replicate without becoming a storage liability.

## **🔁 Cleaning Up Replication Slots in PostgreSQL: Free Up Disk Space the Smart Way**

PostgreSQL provides several advanced features to support replication and real-time data streaming. One such feature is the ****replication slot****, a powerful mechanism used to ensure that changes made to the database are not lost before they are received by replicas or logical decoding clients.

But with great power comes great responsibility. If replication slots are ****abandoned or forgotten****, they can cause serious disk space issues by preventing the cleanup of old WAL (Write-Ahead Log) files. Let’s break down exactly how replication slots work, how to identify when they’re no longer needed, and how to safely remove them.

## **🧠 Understanding Replication Slots**

Replication slots serve as a ****guarantee**** that PostgreSQL won’t delete WAL files that are still needed by subscribers. These subscribers could be:

* Streaming physical replicas (for high availability)
* Logical replication clients (e.g., Debezium, pglogical)
* Custom applications consuming database changes (e.g., Kafka CDC connectors)

When you create a replication slot, PostgreSQL retains WAL segments on the primary server ****until**** that slot reports the data has been successfully received and processed. This ensures reliability in data delivery — but it also means that ****WAL files can pile up quickly**** if the slot is not actively used.

If a replica goes down or a logical client stops consuming data, and the slot isn’t removed, PostgreSQL continues hoarding WAL files that would otherwise be recycled. Over time, this can lead to ****disk usage growing uncontrollably****, eventually resulting in space exhaustion and possible outages.

## **📋 2.1 View All Replication Slots**

Before making any decisions, the first step is to inspect the current state of your replication slots. PostgreSQL provides a system view called pg\_replication\_slots that gives you detailed information about each slot.

Run the following query in your PostgreSQL shell:

SELECT \* FROM pg\_replication\_slots;

This will return a list of all replication slots, both physical and logical. Key columns to observe:

* ****slot\_name****: The name assigned to the replication slot.
* ****slot\_type****: Can be physical (for streaming replication) or logical (for logical decoding).
* ****active****: Boolean value indicating whether a client is currently connected and using the slot.
* ****restart\_lsn****: Log Sequence Number indicating where WAL retention starts for this slot.

### **Example Output**

slot\_name slot\_type active restart\_lsn replica\_slot\_1 physical t 0/4501A30 logical\_slot\_2 logical f 0/25010F0

In the above output:

* replica\_slot\_1 is in use and connected (active = true)
* logical\_slot\_2 is inactive and might be a candidate for removal

## **🔎 2.2 Identify Unused Slots**

The next step is to identify ****unused or stale**** replication slots — these are the ones that are no longer in use but are still consuming resources.

### **Look for the following signs:**

1. ****active = false****

* Indicates that no client is currently connected to the slot.
* If this status has been false for a long time, the slot is likely abandoned.

2. ****Very old restart\_lsn****

* A stale or outdated LSN shows that the slot hasn’t been consuming WAL data in a while.
* The older the restart\_lsn, the more WAL files are retained to support that slot — increasing disk usage.

### **How to Determine “Very Old”?**

Compare the restart\_lsn with the current WAL insert location:

SELECT pg\_current\_wal\_lsn();

If the difference between restart\_lsn and the current WAL LSN is significant, and the slot has been inactive, it’s a strong indicator the slot is obsolete.

🔍 **Tip:** Also check your PostgreSQL monitoring system (e.g., pgAdmin, Grafana, or custom Prometheus dashboards) to track the time since the slot was last active.

## **🧹 2.3 Drop Unused Replication Slots**

Once you’ve confirmed that a replication slot is no longer needed — meaning its client (replica or logical subscriber) has been decommissioned, migrated, or shut down — you can remove the slot manually.

Use the pg\_drop\_replication\_slot() function:

SELECT pg\_drop\_replication\_slot('my\_replication\_slot');

Replace 'my\_replication\_slot' with the actual name of the slot you want to drop.

### **Example:**

SELECT pg\_drop\_replication\_slot('logical\_slot\_2');

This will free PostgreSQL from having to retain WAL segments for that slot, and ****you will immediately see disk space usage begin to drop****, assuming those WALs are not required by other slots.

## **⚠️ Proceed with Caution**

Before dropping any replication slot:

* ****Double-check**** with your team, especially DevOps or replication administrators.
* ****Ensure**** that the slot’s client is truly no longer in use.
* ****NEVER**** drop a slot that is marked active = true — this will abruptly terminate replication and may lead to data inconsistencies or loss of synchronization.

💡 **Pro tip:** Document slot usage and ownership in your organization so that it’s clear which system or team owns which slot — especially important in environments with multiple logical replication clients.

## **✅ Summary**

Replication slots are essential for reliable and fault-tolerant replication in PostgreSQL, but they come with one critical requirement: ****manual management****. If not cleaned up, stale slots can cause serious storage issues by preventing WAL cleanup.

In this section, you’ve learned how to:

* List all existing replication slots using pg\_replication\_slots
* Identify stale or abandoned slots by checking active status and restart\_lsn
* Drop unused slots safely with pg\_drop\_replication\_slot()

Regularly auditing your replication slots — especially after replica decommissions or migration events — ensures your PostgreSQL instance stays lean, efficient, and resilient.

## **🔧 Proactive Maintenance Best Practices**

As we’ve seen, PostgreSQL’s WAL files and replication slots are essential for ensuring durability, crash recovery, and high availability. But without proactive management, they can silently consume disk space and lead to performance degradation or even outages.

To avoid surprises, it’s critical to adopt a few ****best practices**** that allow you to stay ahead of potential issues.

## **📊 Regularly Monitor WAL Size and Replication Slot Usage**

Don’t wait until your disk is full to check the status of WAL files or replication slots. Instead, make it a routine part of your database health checks.

* Monitor the size of the pg\_wal directory.
* Track how many WAL files are retained over time.
* Periodically query pg\_replication\_slots to review slot activity and age of restart\_lsn.

This visibility allows you to catch unusual growth patterns early — before they impact system performance.

du -sh /path/to/pg\_wal

SELECT \* FROM pg\_replication\_slots;

## **⚙️ Automate Monitoring with**pg\_stat\_replication**and Alerts**

For larger systems or production environments, manual checks aren’t enough. Instead, integrate automated monitoring that tracks:

* Replication slot activity
* Replication lag
* WAL accumulation over time

PostgreSQL provides a powerful view called pg\_stat\_replication that shows details about each connected replica:

SELECT \* FROM pg\_stat\_replication;

You can use this data to set up alerts using tools like ****Prometheus + Grafana****, ****Zabbix****, ****Nagios****, or even custom shell scripts. Trigger alerts when:

* Replication lag exceeds a safe threshold
* WAL directory size crosses a critical limit
* A replication slot has been inactive for too long

This level of automation ensures you never miss an early warning sign.

## **🛠 Review**max\_replication\_slots**and**wal\_keep\_size**Configurations**

Configuration tuning is another critical aspect of WAL and slot management.

* ****max\_replication\_slots****: Controls the maximum number of replication slots PostgreSQL can maintain. Make sure it’s set high enough to accommodate your HA or logical replication architecture — but not so high that abandoned slots accumulate unnoticed.
* ****wal\_keep\_size****: Defines the minimum amount of WAL data (in MB or GB) to retain on the primary server for connected replicas. If this value is too high, it can contribute to disk bloat. If it’s too low, slower replicas may fail to stay in sync.

Check these with:

SHOW max\_replication\_slots;  
SHOW wal\_keep\_size;

Adjust as needed with ALTER SYSTEM and reload the config:

ALTER SYSTEM SET wal\_keep\_size = '512MB';  
SELECT pg\_reload\_conf();

## **🧹 Periodically Run**pg\_archivecleanup**for Archived WALs**

If you’re using WAL archiving (for PITR or backups), remember that PostgreSQL ****won’t delete archived WAL files automatically****. These can accumulate over time, consuming valuable storage.

Use the pg\_archivecleanup tool to remove WAL files that are no longer required by your recovery or replication strategy.

Example:

pg\_archivecleanup -d /postgres/wal\_archive 00000001000000000000005A

Be careful to only remove WAL files that are ****no longer needed**** for recovery or failover. Automating this with a scheduled cron job can prevent long-term storage problems.

## **✅ Conclusion**

Managing WAL files and replication slots in PostgreSQL isn’t just a best practice — it’s a necessity. Without proper oversight, these background processes can silently drain your disk space, impact performance, and even risk system availability.

By adopting the following simple but powerful habits:

* Monitoring WAL growth and slot activity regularly
* Automating alerts for lag or slot issues
* Tuning replication and WAL-related parameters thoughtfully
* Running safe cleanup procedures like pg\_archivecleanup

— you can ensure your PostgreSQL environment remains ****efficient****, ****stable****, and ****production-ready****.

Take the time to understand these mechanisms and maintain them proactively. Your future self — and your infrastructure team — will thank you.