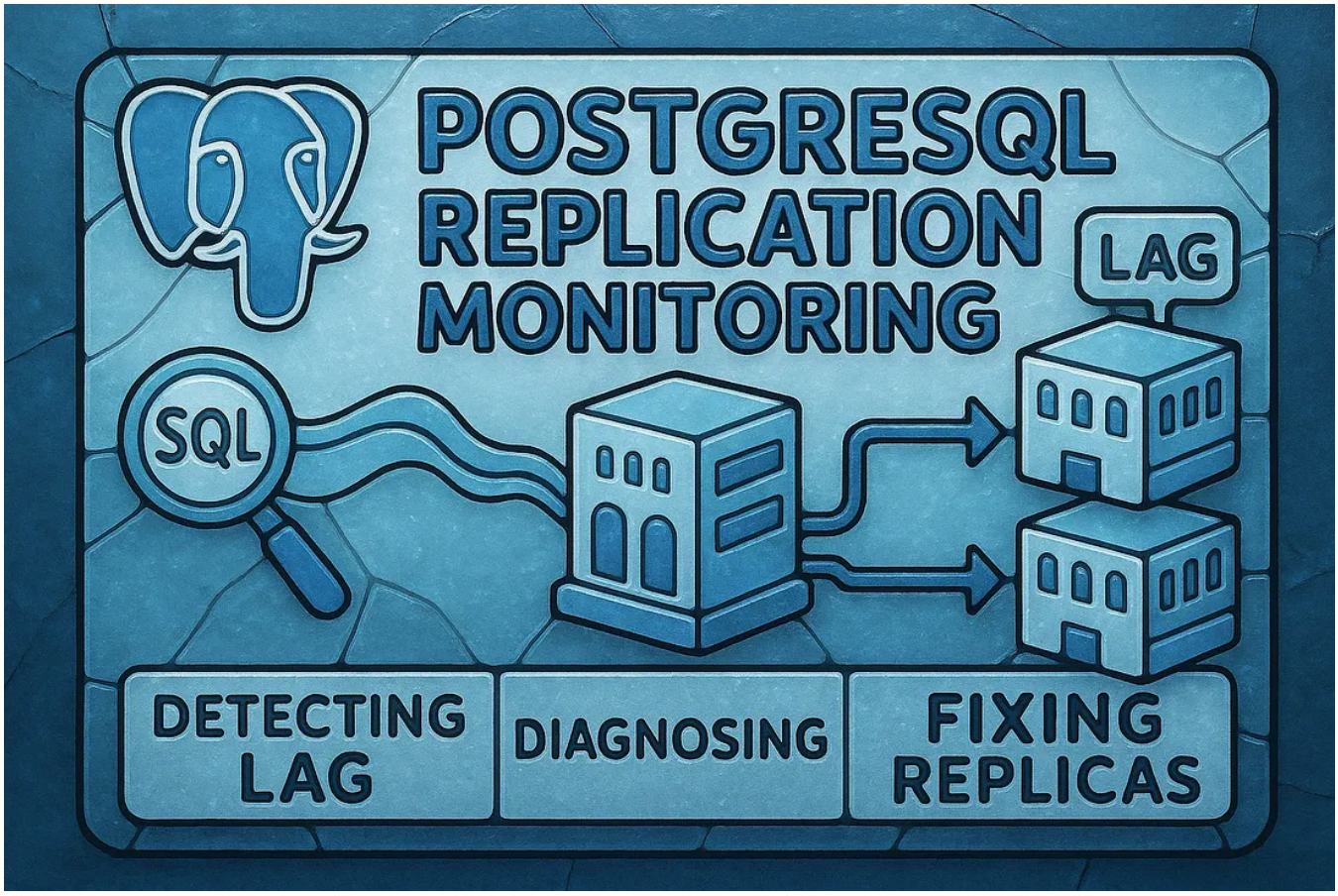
PostgreSQL Replication Monitoring: Detecting, Diagnosing & Fixing Lag and Out-of-Sync Replicas

21 min read · Jun 27, 2025

J Jeyaram Ayyalusamy Following



PostgreSQL replication is a foundational technology for building **highly available**, **fault-tolerant**, and **scalable** database systems. Whether you're running a hot standby for failover, using read replicas for query offloading, or preparing for disaster

recovery, replication ensures that your data is continuously and reliably propagated from a primary server to one or more replicas.

But here's the catch: **replication is only as strong as your ability to monitor and maintain it.** Many database administrators (DBAs) and site reliability engineers (SREs) discover too late that a replica is lagging — or worse, completely out of sync — during a production incident.

In this article, we'll walk you through the **key components of replication monitoring and troubleshooting**, including:

- ✅ **The most common causes of replication lag**
  Understand why replicas fall behind and how to proactively detect the warning signs.
- ✅ **SQL queries you can run to monitor replication in real time**
  Use PostgreSQL's built-in functions and system views to track replication progress down to the log sequence number (LSN).
- ✅ **How to fix out-of-sync replicas**
  Learn step-by-step recovery procedures to bring lagging replicas back in sync without full rebuilds — when possible.
- ✅ **Tips on WAL configuration and recovery**
  Discover how tuning WAL settings like `wal_keep_size`, `archive_timeout`, and `replication slots` can help ensure smooth, uninterrupted replication.

🔄 What Is Streaming Replication in PostgreSQL?

In modern database infrastructure, **high availability**, **disaster recovery**, and **read scalability** are no longer optional — they're essential. PostgreSQL addresses these needs through one of its most robust built-in features: **streaming replication.**

Streaming replication allows a PostgreSQL primary (or master) server to **continuously send changes** to one or more **replica (standby) servers** using a near real-time, asynchronous process. These replicas stay closely synchronized with the primary, thanks to PostgreSQL's internal mechanism called **Write-Ahead Logging (WAL).**

Let's break down what this means — and why it matters.

🔧 How Streaming Replication Works (Under the Hood)

PostgreSQL follows the **WAL-first** approach, meaning every transaction — whether it modifies a single row or a thousand — is **first written to a log file** before being flushed to disk. This ensures durability and recoverability.

Streaming replication uses these WAL files to **ship changes to replicas** over a persistent TCP connection:

1. The primary server records every data change in a WAL segment.
2. The replica establishes a streaming connection with the primary using `pg_basebackup` or other setup tools.
3. As soon as a WAL segment is written, it is **streamed to the replica in real time**.
4. The replica **replays the changes** locally, applying them in the same sequence as the primary.

This replication is typically **asynchronous**, which means there's a small delay (replication lag), but you can also configure **synchronous replication** if your use case demands strict consistency.

✅ Why Use Streaming Replication?

1. 🛡️ High Availability (HA)

If your primary PostgreSQL server crashes due to hardware failure, operating system issues, or database corruption, you can **promote a standby replica** to become the new primary. This minimizes downtime and avoids catastrophic data loss.

Streaming replication forms the **backbone of most PostgreSQL HA architectures** — especially when combined with automated failover tools like **Patroni**, **repmgr**, or **PgAutoFailover**.

2. 📈 Read Scalability

In systems with **heavy read workloads**, streaming replicas can be used to **distribute SELECT queries**. For example:

- Analytics dashboards
- Reporting engines
- BI tools

By offloading these workloads from the primary, you:

- Improve response times
- Reduce contention on write transactions
- Increase the overall throughput of your application

> 💡 Note: Replicas are **read-only** by default. You cannot write to them unless you promote them to a primary.

3. 💾 Backup Redundancy and Offloading

In many production setups, backups are taken from a replica server instead of the primary. This prevents disruption to ongoing transactional workloads and reduces the risk of performance bottlenecks.

Common backup workflows supported on replicas include:

- Physical backups (`pg_basebackup`)
- Logical exports (`pg_dump`)
- PITR (Point-in-Time Recovery) workflows using archived WALs

Having multiple replicas also allows for **geographically distributed backups**, which helps in disaster recovery scenarios like data center outages.

⚠️ Streaming Replication Is Not “Set It and Forget It”

One of the most common misconceptions among new PostgreSQL users is the belief that once replication is configured, it will take care of itself indefinitely. Unfortunately, that’s not the case.

Even though PostgreSQL’s replication is reliable and low-maintenance, **it still requires active monitoring** and occasional intervention. Here’s why:

- **Network disruptions** can break the streaming connection.
- **Disk space issues** can prevent WAL segments from being retained long enough for the replica to catch up.
- **Configuration mismatches** may cause the replica to fail during failover testing.
- **Replication lag** can silently grow if the replica’s I/O subsystem is slower than the primary.

You must keep a close eye on:

- The replication status
- Lag between the primary and standby (`pg_stat_replication`)
- WAL retention policies (`wal_keep_size` , `max_wal_size`)
- Whether your replica is replaying logs without delay (`pg_last_wal_replay_lsn`)

📌 Summary

Streaming replication in PostgreSQL is:

- A **real-time, WAL-based mechanism** for copying data from a primary to one or more replicas.
- Essential for **high availability**, **read scaling**, and **backup redundancy**.
- Easy to set up — but **not maintenance-free**.

Properly configured and monitored, streaming replication transforms PostgreSQL from a single-node database into a **distributed, fault-tolerant system** ready to handle enterprise-scale workloads.

⚠ Common Replication Issues in PostgreSQL

While PostgreSQL's **streaming replication** is known for its simplicity and reliability, real-world environments often introduce complexities that can degrade its performance or reliability. If left unchecked, these issues can cause **replication lag**, **data inconsistency**, or even **replica failure** — putting your high availability setup at serious risk.

In this section, we'll break down four of the most common replication issues DBAs encounter, why they happen, and what symptoms to look for.

1️⃣ WAL Sending Bottleneck

What happens:
The **primary server** is generating **Write-Ahead Log (WAL)** data faster than the **replica can consume** it. This means the replica starts to lag behind, unable to keep up with the flow of data being written.

Why it happens:

- A **sudden spike in write operations** (e.g., bulk inserts, batch jobs, high-volume API traffic)
- Insufficient CPU resources on the primary for WAL encoding and dispatch
- Disk or network I/O bottlenecks delaying WAL delivery to the replica

Symptoms:

- Replication lag increasing over time
- Growing number of unarchived WAL files in the primary's `pg_wal` directory

- `pg_stat_replication` shows high lag in `write_lsn` vs `sent_lsn`

> 🧠 Fix tip: Monitor WAL generation rate, increase WAL retention (`wal_keep_size`), and optimize I/O performance on the primary.

2️⃣ WAL Receiving Problems

What happens:
The **replica server** is unable to **receive or apply WAL data** at the pace it's arriving. This creates a backlog, as WAL files continue to accumulate on the primary.

Why it happens:

- **Resource constraints** on the replica (CPU saturation, high memory usage, slow disk)
- Heavy read workloads on the replica competing with WAL replay
- Unoptimized replay mechanisms (e.g., delays from full-page writes or vacuuming)

Symptoms:

- Replication appears connected, but lag steadily increases
- Replica I/O utilization is high
- Checkpoints or autovacuum processes slow down WAL application

> 🧠 Fix tip: Tune replica hardware for WAL replay performance. Monitor `pg_last_wal_replay_lsn()` to detect slow application behavior.

3️⃣ Load-Induced Replication Lag

What happens:
Both the **primary and replica** experience high system load, which creates a bottleneck at both ends of the replication pipeline.

Why it happens:

- On the **primary**, high-volume transactions generate WAL quickly.
- On the **replica**, concurrent queries or reporting workloads prevent timely WAL application.

Symptoms:

- Lag increases rapidly during peak traffic hours
- Replication catches up during off-peak times (intermittent lag pattern)
- Reduced throughput or query timeouts on the replica

> 🧠 Fix tip: Isolate reporting or long-running queries from critical replicas. Use dedicated replicas for read-only analytics.

4️⃣ Network Problems

What happens:
The **network connection** between the primary and replica becomes unstable, causing dropped packets, high latency, or full disconnections.

Why it happens:

- Physical network failures (e.g., cable disconnects, switch outages)
- Unstable VPNs or WAN connections (especially in multi-region setups)
- Misconfigured firewalls or load balancers interfering with replication traffic

Symptoms:

- Replica disconnects intermittently
- WAL segments pile up in the primary's archive or `pg_wal`
- Replication resumes only after network issues are resolved

> 🧠 Fix tip: Monitor network reliability and use dedicated replication links. Consider increasing `wal_sender_timeout` and `wal_receiver_timeout` values for unstable connections.

🧩 Summary

These common issues highlight a critical truth: **replication is only as strong as the infrastructure supporting it.** Even a perfectly configured PostgreSQL cluster can experience problems when I/O, CPU, or network conditions change unexpectedly.

By understanding the root causes behind:

- WAL sending bottlenecks
- WAL replay performance on replicas
- Load-related delays
- Network disruptions

…you can proactively design and maintain a replication setup that's **resilient, observable, and self-healing.**

🔍 Why Do Replicas Go Out of Sync in PostgreSQL?

Setting up streaming replication in PostgreSQL might seem like a one-time effort, but **keeping replicas in sync with the primary server is an ongoing responsibility.** Replication failures don't usually come with loud alarms. More often, they silently build up — and when caught too late, result in a broken or unusable standby.

Understanding the root causes of replication drift is essential to maintaining **high availability, read scaling,** and **disaster recovery readiness.** Let's explore the most common reasons replicas fall out of sync.

⚠️ 1. WAL Backlog Too Large for the Replica to Catch Up

PostgreSQL relies on **Write-Ahead Log (WAL)** files to replicate changes. These are continuously generated by the primary and streamed to replicas.

However, if a replica is **slow to apply WALs**, it accumulates a backlog. If this lag becomes too large and the WAL files needed for replay are **no longer available on the primary** (due to being recycled or deleted), the replica simply **can't catch up**.

Common triggers:

- Replica underpowered (slow disk, CPU bottlenecks)
- Long periods of replica downtime
- Spike in write activity on primary (bulk insert/update jobs)

> 🔁 If the required WAL files are missing, the replication link is broken and **cannot be resumed**. The only solution is a full **reinitialization** using a fresh base backup.

⚠️ 2. WAL Files Expired on Primary Before Replica Could Apply Them

PostgreSQL doesn't keep WAL files forever. It retains them **based on your configuration** (`wal_keep_size` , or `wal_keep_segments` in older versions) and recycles them during checkpoints.

If these settings are too conservative, WAL segments may be deleted **before a slow or recovering replica can read them**.

Real-world scenario:

- A replica is offline for 30 minutes
- The primary's WAL settings only keep 100MB of WAL files (about 6–7 files)
- In those 30 minutes, 500MB of WALs are generated and overwritten
- Replica comes back online and asks for WAL segment `X` — but it no longer exists

> 📉 Result: The replication fails, and the standby must be **rebuilt from scratch.**

⚠️ 3. Replica Down During WAL Generation Spike

Unexpected outages — even brief ones — can be catastrophic if they coincide with **high write volumes** on the primary.

Let's say the primary generates WAL at 1MB/sec during normal usage. But during a major bulk update, that rate spikes to 20MB/sec. If a replica is offline during this burst, it may miss hundreds of MBs worth of WAL segments in minutes.

Complications:

- Limited disk space on the primary for `pg_wal`
- Small `wal_keep_size` or aggressive `checkpoint_timeout`
- No active archiving to store old WALs

> 💣 When the replica reconnects, the WALs it needs are gone — replication breaks.

⚠️ 4. Misconfigured `archive_command` or `restore_command`

For clusters using **archived WALs** to support delayed replicas or PITR (Point-In-Time Recovery), the `archive_command` (on the primary) and `restore_command` (on the replica) are vital.

If either command is:

- Incorrectly written
- Fails silently
- Points to an invalid path
- Doesn't have permission to write or read

Then WAL files may **never be archived** or **fail to restore when needed.** If a replica falls back on `restore_command` to fetch a missing WAL and it doesn't work — it's game over.

> 🔧 Always test archive and restore commands manually. Add logging, alerts, and fallback handling where possible.

🔧 Key PostgreSQL Parameters That Impact Replication Health

To reduce the chances of your replicas falling out of sync, you must configure PostgreSQL's WAL and replication-related parameters with intention. Below are the most important ones — with usage guidance.

🔑 1. `wal_keep_segments` (PostgreSQL 12 and below)

```
wal_keep_segments = 32
```

- Retains a minimum number of WAL segments in the `pg_wal` directory.
- Prevents deletion of WALs needed by slow or offline replicas.

> 🧠 Deprecated in PostgreSQL 13+. Use `wal_keep_size` instead.

🔑 2. `wal_keep_size` (PostgreSQL 13+)

```
wal_keep_size = '512MB'
```

- Modern replacement for `wal_keep_segments`.
- Sets the **total WAL size** (in MB) that must be retained for replication.
- Automatically adjusts for the actual number of segments.

> 💡 Recommendation: Estimate the WAL generation rate and retention window your slowest replica needs to catch up.

🔑 3. `max_wal_size`

```
max_wal_size = '2GB'
```

- Sets the upper limit of WAL accumulation before a **checkpoint** is triggered.
- Affects how quickly old WALs are recycled.

If set too low:

- Checkpoints occur more often
- WALs are deleted quickly
- Replicas might miss segments if they lag

If set too high:

- Longer recovery times in crash scenarios
- More disk usage

> 🎯 Tune this alongside `checkpoint_timeout` and `min_wal_size` for a balance of safety and performance.

🔑 4. `archive_command`

```
archive_command = 'cp %p /var/lib/postgresql/wal_archive/%f'
```

- Specifies how PostgreSQL archives completed WAL files.

- Essential for enabling **Point-in-Time Recovery (PITR)** and for replicas that rely on WAL archiving to fetch missing segments.

> 🛠️ This command must be fail-safe. If it fails silently, your backups and replication safety net collapse.

✅ Final Recommendations

To keep your replicas healthy and prevent sudden desynchronization:

- ✅ Set `wal_keep_size` generously based on your system's write rate and expected downtime tolerance.
- ✅ Monitor `pg_stat_replication` and WAL lag regularly.
- ✅ Archive WALs if using delayed replicas, PITR, or long-distance replicas.
- ✅ Test recovery scenarios and validate your `archive_command` / `restore_command`.
- ✅ Avoid aggressive checkpointing unless needed for performance tuning.

🧩 In Summary

Replicas don't just "go out of sync." They're **pushed out** by misconfigured settings, insufficient WAL retention, or unexpected workload spikes. The good news? With the right tuning and awareness, you can build a PostgreSQL cluster that's resilient, fast, and disaster-ready.

📊 How to Monitor Replication Lag in PostgreSQL (In Depth)

In a high-availability PostgreSQL setup, **replication lag** can quietly grow into a critical issue. It occurs when a replica falls behind the primary, leading to stale data, delayed failovers, or even replication breakdown if the gap becomes unbridgeable.

Thankfully, PostgreSQL provides rich internal views and functions to monitor the health of replication streams in real time. This guide walks through the **exact SQL queries** to use on the **primary server** to measure lag, pinpoint where it's happening, and take timely action.

Let's break it down step by step. 🛠️

🖥️ All Monitoring is Done on the Primary Server

Streaming replication is a **pull-based mechanism**: the replica connects to the primary and fetches WAL (Write-Ahead Log) changes. That means the **primary knows about the state of each replica** and tracks its progress internally.

PostgreSQL stores this information in views like `pg_stat_replication` and `pg_replication_slots`. Here's how to extract meaningful insights from them.

1️⃣ View Active Replication Connections

To check which replicas are connected and their real-time status, run:

```
SELECT * FROM pg_stat_replication;
```

This system view contains one row for **each replica** currently connected via streaming replication. Key columns include:

- `pid`: The backend process ID handling the connection
- `usesysid`: User ID that initiated the replication
- `application_name`: The name assigned by the replica (often in `primary_conninfo`)
- `client_addr`: IP address of the replica
- `state`: The current replication state (e.g., `streaming`, `catchup`, `startup`)

- `sent_lsn` : Latest WAL LSN sent to the replica
- `write_lsn`, `flush_lsn`, `replay_lsn` : WAL positions acknowledged by the replica at various stages

> 🧠 Useful for: Checking how many replicas are live, confirming their streaming status, and tracking WAL positions.

2️⃣ Monitor Replication Slots

If you're using **replication slots**, you can track their health with:

```
SELECT * FROM pg_replication_slots;
```

This view lists all logical or physical replication slots. For each slot, you get:

- `slot_name` : Unique name of the replication slot
- `active` : Boolean showing whether a replica is currently using the slot
- `restart_lsn` : The oldest LSN that PostgreSQL must retain to satisfy the replica

> ⚠ Why this matters: Inactive slots prevent WAL cleanup. If a slot isn't being consumed, WAL segments can build up indefinitely, **risking disk exhaustion on the primary**.

3️⃣ Check the Current WAL Position (LSN)

To assess how far ahead the primary is compared to the replica(s), run:

```
SELECT pg_current_wal_lsn();
```

This returns the **Log Sequence Number (LSN)** of the most recent transaction written to the WAL. It's the anchor point against which replica progress is measured.

> 📌 Tip: Think of `pg_current_wal_lsn()` as the "front" of the WAL stream. The closer a replica is to this value, the less lag it has.

4️⃣ Calculate Replication Lag Metrics (By Stage)

PostgreSQL gives you the tools to measure lag **at every stage** of the replication pipeline. Here's a query to visualize sending, receiving, replaying, and total lag — for each replica:

```
SELECT
  pid,
  application_name,
  pg_wal_lsn_diff(pg_current_wal_lsn(), sent_lsn) AS sending_lag,
  pg_wal_lsn_diff(sent_lsn, flush_lsn) AS receiving_lag,
  pg_wal_lsn_diff(flush_lsn, replay_lsn) AS replaying_lag,
  pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) AS total_lag
FROM pg_stat_replication;
```

📊 What Do These Metrics Mean?

Lag Metric	What It Tells You
`sending_lag`	How far behind the replica is in terms of WAL **received**. High = slow network or walsender issues.
`receiving_lag`	How much WAL is **written but not flushed** to disk on the replica. High = slow I/O on replica.
`replaying_lag`	WAL is received and flushed but **not yet applied**. High = heavy CPU or active queries on replica.
`total_lag`	Overall delay between WAL written on primary and replayed on replica. The **true measure of lag**.

> 🧠 Real-world use: By breaking down the lag stages, you can **diagnose exactly where the bottleneck is** — whether it's network speed, replica disk performance, or application load.

🧪 Example Output

Suppose you run the query and get this output:

pid	application_name	sending_lag	receiving_lag	replaying_lag	total_lag
2789	replica01	512kB	0kB	128kB	640kB

This means:

- The primary is ahead of replica01 by 512kB (sending lag)
- The replica has flushed all received WAL (receiving lag = 0)
- It has 128kB worth of WAL ready to apply (replaying lag)
- Total delay from primary to replay: 640kB

> 📈 This kind of insight is vital for high-availability failover planning. If you promote a replica with 500MB of lag, your applications could read **stale or inconsistent data.**

✅ Final Thoughts

Replication lag is one of the most important metrics to watch in any PostgreSQL replication setup — whether you're scaling out reads, configuring failover, or preparing for PITR.

Using a combination of:

- `pg_stat_replication` for real-time connection states
- `pg_replication_slots` for WAL retention safety
- `pg_current_wal_lsn()` and `pg_wal_lsn_diff()` for precise lag measurement

You gain **full observability into replica performance.**

> 🚨 Don't wait for replication to break — add these queries into monitoring dashboards or alerting systems to catch slow or failing replicas early.

🧭 How to Monitor Replication from the Replica Server in PostgreSQL (Deep Dive)

PostgreSQL's streaming replication allows a replica server to stay in sync with the primary server by continuously receiving and replaying **Write-Ahead Log (WAL)** records. While most monitoring is performed from the **primary side**, you can gain valuable insights directly from the **replica server** itself.

This is especially important in scenarios such as:

- Verifying the health of a failover-ready standby
- Troubleshooting replication lag
- Ensuring that a replica is actively consuming WALs
- Monitoring delay in replaying transactions

Let's break this down step by step to understand how to confidently check replication status on a standby node.

✅ Step 1: Confirm Replica Status

The first step is to verify that the server is actually running in **replica (recovery) mode**.

🔍 SQL Query:

```
SELECT pg_is_in_recovery();
```

✅ What It Returns:

- `true` : The server is **in recovery mode**, i.e., it is acting as a **replica** and applying WAL records received from the primary.
- `false` : The server is a **primary** or has been **promoted** from standby (e.g., due to a failover).

🧠 Why This Matters:

This is a **foundational check**. If you mistakenly believe you're looking at a replica, but `pg_is_in_recovery()` returns `false`, then:

- The server is no longer a replica.
- It may have been **promoted** due to a failover (manual or automatic).
- Replication is no longer active.

> Always run this check before assuming replication is functioning.

✅ Step 2: Track WAL Progress (Receive vs. Replay)

PostgreSQL provides internal functions that let you track how far the replica has progressed in receiving and applying changes from the primary.

a) 🔄 `pg_last_wal_receive_lsn()`

```
SELECT pg_last_wal_receive_lsn();
```

This returns the **Log Sequence Number (LSN)** of the most recent WAL entry **received** from the primary.

- If the result is `NULL`, the replica **has not received any WAL** yet — possibly because it's disconnected.
- If a valid LSN is returned, it means the replica is **receiving WALs**, but not necessarily replaying them yet.

b) 🧪 `pg_last_wal_replay_lsn()`

```
SELECT pg_last_wal_replay_lsn();
```

This function returns the LSN of the **last WAL record that has been applied (replayed)** on the replica.

Comparing this with the `receive_lsn` shows whether the replica is:

- Replaying WALs in near real-time
- **Lagging in application** (if `replay_lsn < receive_lsn`)

> This often occurs if the replica is under **I/O stress**, has CPU bottlenecks, or is processing heavy read queries.

c) ⏱️ `pg_last_xact_replay_timestamp()`

```
SELECT pg_last_xact_replay_timestamp();
```

This function returns the **timestamp of the last transaction** replayed by the replica.

- If it returns `NULL`, then **no WAL transaction has been applied yet**, indicating:
- A very new replica
- A stalled replication stream
- A critical misconfiguration

> This timestamp is critical for calculating **time-based lag**, which we'll cover next.

✅ Step 3: Calculate Time-Based Replication Delay

While LSN comparisons are precise, they are not intuitive to humans. Time-based delay tells you **how many seconds** the replica is behind the primary in applying changes.

⏱️ SQL Query:

```
SELECT
  CASE
    WHEN pg_last_wal_receive_lsn() = pg_last_wal_replay_lsn() THEN 0
    ELSE EXTRACT(EPOCH FROM now() - pg_last_xact_replay_timestamp())
  END AS replication_log_delay;
```

🔍 What This Does:

- If the **received LSN** and **replayed LSN** are equal → the replica is caught up → returns `0` seconds lag.
- If they differ → calculates the **time difference in seconds** between now and the timestamp of the last applied transaction.

💡 Sample Output:

```
replication_log_delay
----------------------
4.83
```

This tells us that the replica is **approximately 4.83 seconds behind** the primary.

📈 Interpreting Replication Lag

Delay (seconds) Meaning Action 0 Replica is fully caught up No action needed 1–5 Acceptable in most cases Monitor regularly 5–30 Moderate lag; may indicate performance issues Check replica CPU, I/O, or network load 30+ Significant lag

Immediate investigation recommended NULL No transactions replayed Validate replication is functioning

> 🛑 In production HA environments, a lag beyond **30 seconds** is a red flag — especially if you rely on the replica for **failover readiness.**

🧪 Bonus: Real-World Use Cases

🔄 Use Case 1: Replica Health Check in a HA Cluster

Before executing a manual failover or switch-over, run all three checks on the target standby to confirm:

- It is in recovery mode (`pg_is_in_recovery() = true`)
- It is caught up in replaying WALs (`replication_log_delay = 0`)
- It has recently replayed a transaction (`pg_last_xact_replay_timestamp()` is recent)

🛡️ Use Case 2: Alerting System Integration

You can run this delay query periodically via:

- A cron job that writes to logs or sends email alerts
- A Prometheus exporter collecting `replication_log_delay` as a metric
- A custom script that triggers Slack/Teams notifications if lag > threshold

✅ Final Thoughts

Monitoring from the replica is not optional — it is **essential.** It's the only way to ensure your standby servers are:

- Actually functioning as replicas

- Receiving and applying data from the primary
- Keeping up with the primary with minimal delay

By regularly running these built-in PostgreSQL functions, you can build confidence in your replication setup and detect issues before they turn into disasters.

🔨 How to Fix an Out-of-Sync PostgreSQL Replica (The Complete Recovery Guide)

In PostgreSQL streaming replication, the **replica (standby)** constantly reads **WAL (Write-Ahead Log)** files from the primary server and replays them locally to stay in sync. But replication is only reliable **as long as the replica keeps up** with the WAL stream.

Sometimes, due to network issues, downtime, or resource constraints, the replica **falls too far behind**. If the primary has already **removed or archived the WAL files** the replica needs to catch up, replication breaks entirely.

> 📉 You may see errors like:

```
requested WAL segment has already been removed
could not receive data from WAL stream
replication terminated unexpectedly
```

When this happens, your replica is considered **out-of-sync** and must be **rebuilt from scratch**. Don't worry — the process is straightforward and robust if done correctly.

🧭 When Should You Rebuild the Replica?

You'll need to reinitialize a replica when:

- The required WAL files are **no longer available** on the primary or archive
- The replica has been **offline too long** and missed critical segments

- You want to perform a **clean reset** after troubleshooting
- There are **data integrity concerns** due to incomplete replay

🛠 Step-by-Step: Rebuilding an Out-of-Sync PostgreSQL Replica

✅ Step 1: Validate Archive and Restore Commands

Before rebuilding, it's essential to confirm that WAL archiving is configured correctly. This ensures that the replica can recover seamlessly from future delays.

🔍 On the Primary Server:

Run:

```
SHOW archive_command;
SHOW archive_mode;
```

- `archive_mode` should be `on`
- `archive_command` should show a valid command (e.g., `cp`, `rsync`, or custom script)

Example:

```
archive_mode
-------------
on
```

```
archive_command
---------------------------
cp %p /mnt/wal_archive/%f
```

This means PostgreSQL is archiving WAL segments to `/mnt/wal_archive`.

🔍 On the Replica Server:

Run:

```
SHOW restore_command;
```

- This shows how the replica **retrieves archived WALs** during recovery.
- It must point to the correct location where archived WALs live.

Example:

```
restore_command
-----------------------------
cp /mnt/wal_archive/%f %p
```

> 🧠 Valid archive/restore commands are crucial to avoid future out-of-sync scenarios.

⚠️ Step 2: Wipe the Replica's Data Directory

Before taking a new base backup, you must remove the old replica's data. This clears any corrupted, incomplete, or outdated files.

🧹 Command:

```
export PGDATA="/var/lib/pgsql/17/data"
rm -rf $PGDATA/*
```

- This assumes the PostgreSQL 17 data directory is at `/var/lib/pgsql/17/data`.

- Double-check the path before executing.
- Only run this on the **replica**, never on the primary.

> ⚠️ **Important:** Deleting the wrong data directory can result in catastrophic data loss. Always verify `PGDATA`.

📦 Step 3: Take a Fresh Base Backup Using `pg_basebackup`

Now, clone the current state of the primary server to the replica. PostgreSQL provides the `pg_basebackup` utility to do this efficiently.

📌 Command:

```
pg_basebackup -D $PGDATA -R -Xs -c fast -P -v
```

🔍 What Each Flag Does:

Flag	Description
`-D $PGDATA`	Destination directory (must match `PGDATA`)
`-R`	Automatically creates `standby.signal` and connection settings
`-Xs`	Streams WAL during backup (ensures consistent recovery)
`-c fast`	Performs a fast checkpoint to reduce primary lock time
`-P`	Shows progress
`-v`	Verbose mode (prints detailed info)

> ✅ This step prepares your replica to resume streaming from the current WAL position of the primary — with no manual configuration needed.

🔁 Step 4: Restart PostgreSQL on the Replica

Once the base backup is complete, start PostgreSQL on the replica:

```
sudo systemctl start postgresql-17
```

If you're using a different PostgreSQL version, replace `17` with the correct version number.

✅ What You Should See in the Logs:

Use `journalctl`, `tail -f`, or check PostgreSQL logs directly:

```
journalctl -u postgresql-17 -f
```

Look for messages like:

```
entering standby mode
redo starts at A/00000010
consistent recovery state reached
started streaming WAL from primary at A/00000020
```

These indicate:

- The server recognized it is a **replica**
- Recovery started successfully
- The replica is **receiving and replaying WAL** from the primary

> 💡 If the service fails to start, re-check your permissions, `PGDATA` path, or log file location.

🔍 Step 5: Validate Replication Status and Slots

If you're using **replication slots**, verify that the slot is active and functioning:

🧪 On the Primary:

```
SELECT * FROM pg_replication_slots;
```

- Look for the replica's slot (e.g., `replica_slot_1`)
- Confirm it is `active = true`

🔍 Also Check This:

```
SELECT * FROM pg_stat_replication;
```

- This view shows all active replication connections.
- Confirm your replica's `client_addr`, `state = streaming`, and `replay_lsn` is advancing.

> 📊 Lag metrics here help validate that the replica is **replaying data in near real time.**

✅ If Everything Looks Good — You're Fully Resynced!

🎉 Your PostgreSQL replica is now:

- Cleanly rebuilt using the latest base backup
- Streaming WAL changes from the primary
- Fully ready for failover, reporting, or disaster recovery use

This is a best-practice method used in production systems to **recover broken replicas** quickly and safely.

🧠 Pro Tips for Preventing Future Desyncs

- Set `wal_keep_size` large enough to tolerate expected replica lag.
- Archive WALs with a reliable `archive_command` (e.g., `rsync`, `aws s3 cp`).
- Use tools like `pgBackRest` or `barman` to manage backup + archiving.
- Monitor `pg_stat_replication` and `replication delay` via Prometheus or custom alerts.
- Automate health checks to detect broken replication before it's too late.

📦 Summary

Step	Purpose
Confirm Archive/Restore	Validate WAL streaming and PITR readiness
Delete Old Data Directory	Clean start to avoid corrupted or outdated files
Run `pg_basebackup`	Sync replica with fresh snapshot of the primary
Start Replica Service	Resume replication and monitor log recovery progress
Validate Replication Slot	Ensure replication is re-established and healthy

📈 Bonus: Real-Time Replication Health Dashboard (SQL One-Liner)

If you're managing a PostgreSQL cluster in production, **real-time replication visibility** is non-negotiable. You want a simple, reliable way to:

- Monitor replica lag
- Ensure replicas are actively streaming
- Detect delays before they cause bigger problems

Here's a **single SQL query** you can run on the **primary server** that acts like a **mini replication dashboard**:

```
SELECT
  application_name,
  state,
  pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) AS total_lag_bytes,
  EXTRACT(EPOCH FROM now() - pg_last_xact_replay_timestamp()) AS lag_seconds
FROM pg_stat_replication;
```

🔍 Breakdown of Each Column:

- **application_name**: The name of the replica process (set during `primary_conninfo`), useful for identifying which replica you're looking at.

state: Current status of replication. Most common values:

- `streaming` : Actively receiving WAL from the primary.
- `catchup` : Catching up to the current WAL position.
- `startup` : Still initializing.

total_lag_bytes: Byte-level difference between the current WAL position on the primary and the last WAL replayed by the replica.

> *Shows how much data is "in flight" — great for low-level monitoring.*

lag_seconds: Time delay (in seconds) between now and the last transaction replayed by the replica.

> *Helps you correlate network issues, I/O slowdowns, or replay bottlenecks.*

✅ Why This Is Useful

You can schedule this query to run every few seconds using:

- A cron job + psql + logging
- A Grafana/PostgreSQL datasource
- A custom Python/Node.js monitoring tool

> 🚨 Set alerts for thresholds like:
>
> ```
> lag_seconds > 30
>
> total_lag_bytes > 50MB
>
> state != 'streaming'
> ```

This simple query gives **instant insight into replication health**, without needing any external monitoring tools.

📝 Summary Cheat Sheet — PostgreSQL Replication at a Glance

Here's a compact cheat sheet of the most essential PostgreSQL commands and functions to **check, monitor, and repair replication** quickly:

🧪 Task	💻 Command	🔎 Purpose
✅ Check replication state (Primary)	`SELECT * FROM pg_stat_replication;`	View all active replica connections & states
✅ Check recovery state (Replica)	`SELECT pg_is_in_recovery();`	Returns `true` if server is a standby
📍 View current WAL LSN	`SELECT pg_current_wal_lsn();`	Shows current write-ahead log location
📊 Calculate byte-level lag	`pg_wal_lsn_diff(lsn1, lsn2)`	Compares two WAL positions (e.g., current vs replay)
⏱️ Calculate time-based lag	`pg_last_xact_replay_timestamp()`	Timestamp of last replayed transaction on replica
🧹 Rebuild a broken replica	`pg_basebackup -D ... -R -Xs`	Takes a fresh base backup from the primary

📌 Quick Tip:

If you're using tools like **Prometheus**, **Zabbix**, or **Nagios**, most of the above values can be **exported as metrics** and plotted over time. You'll get better historical insights

and alerting when thresholds are crossed.

🚀 Wrap-Up

PostgreSQL gives you all the tools you need to build a rock-solid replication architecture — but it's your responsibility to **monitor it, test it, and fix it when things go wrong.**

This one-liner SQL dashboard and cheat sheet are perfect for:

- On-call rotations
- Quick health checks
- Diagnosing slow or stale replicas
- Writing custom observability dashboards

Want to build a complete Grafana dashboard or automate replica failover with pg_auto_failover or Patroni? Let me know — happy to guide you next!

📢 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 AWS Oracle Open Source Sql

J

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

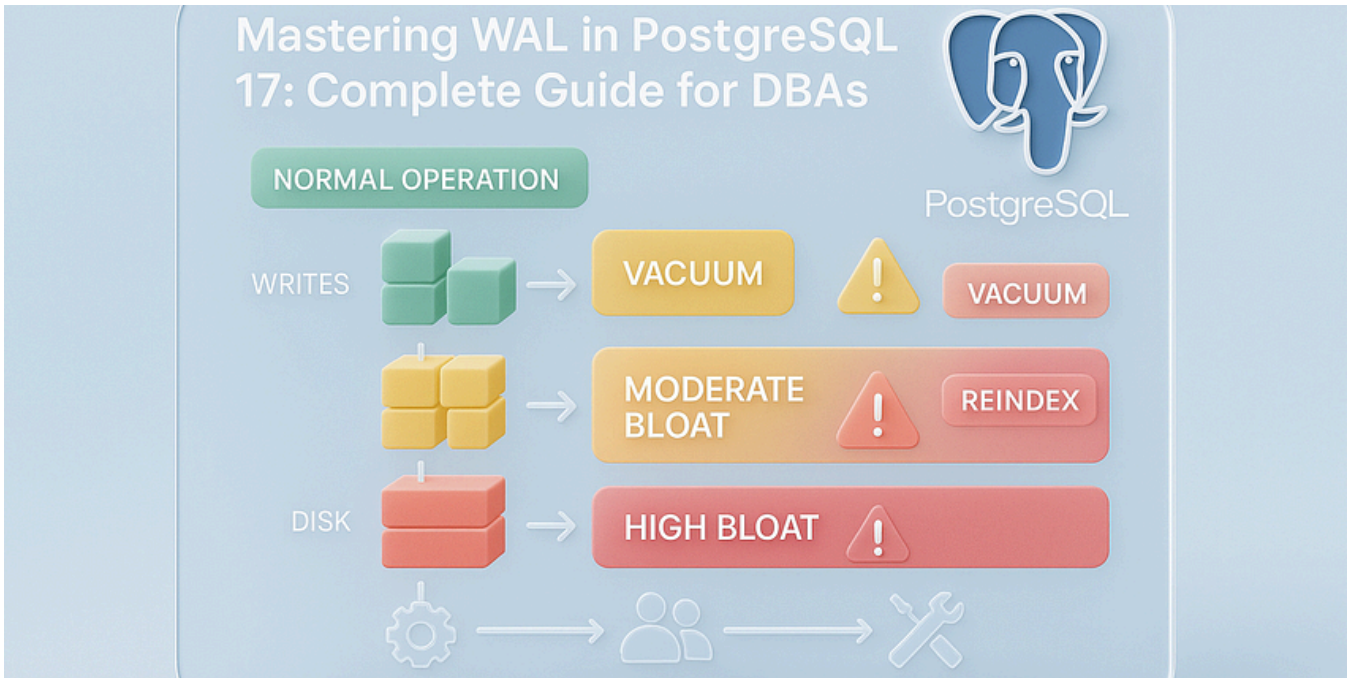
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

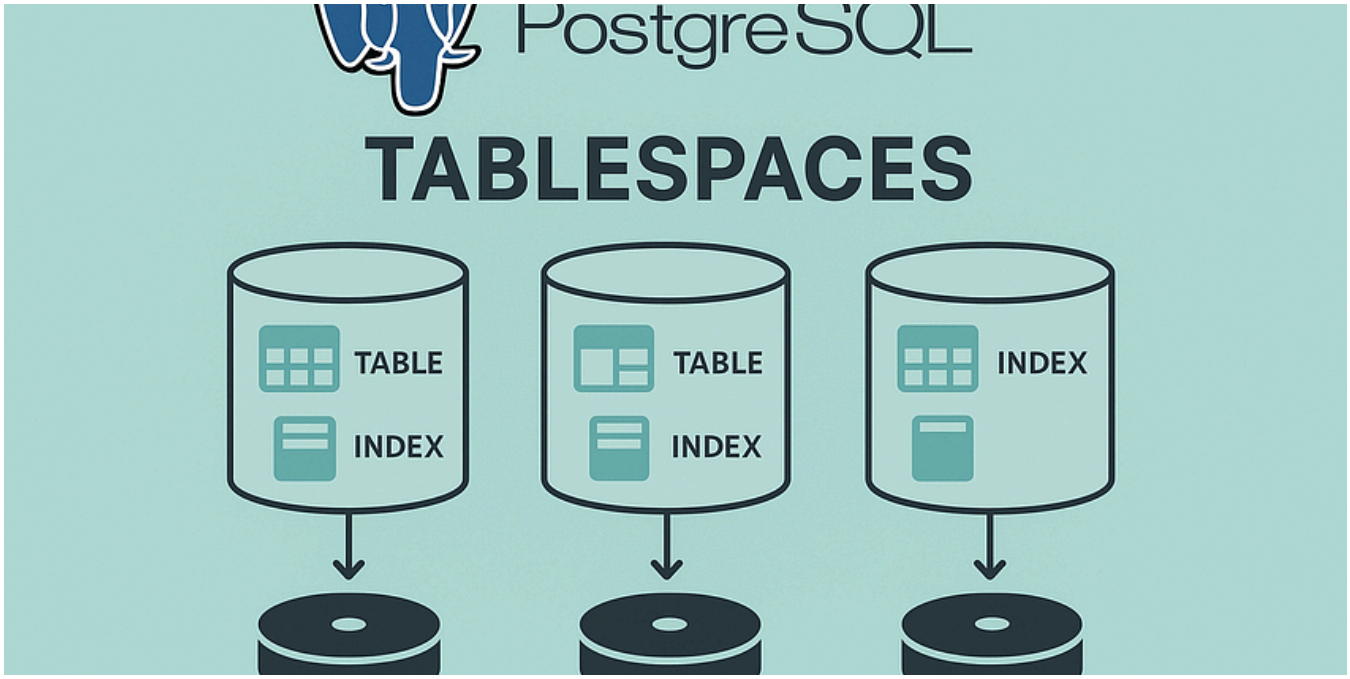


Jeyaram Ayyalusamy

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

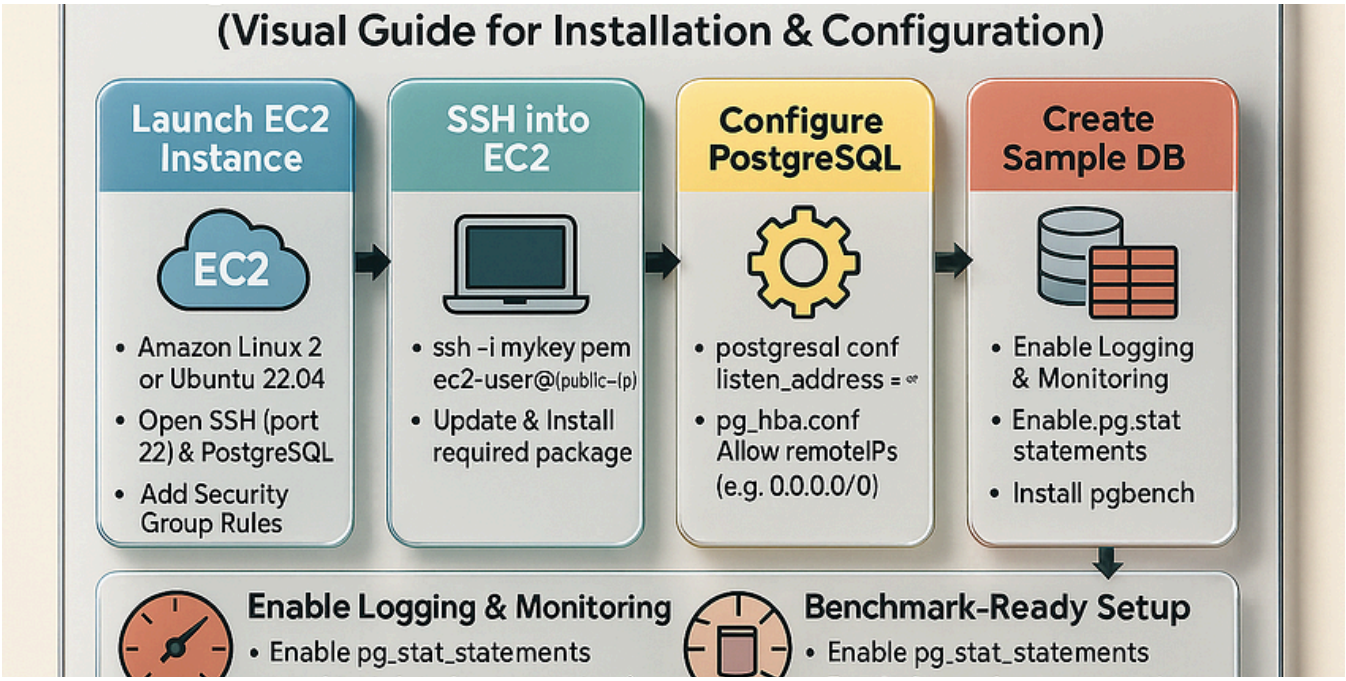


Jeyaram Ayyalusamy

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

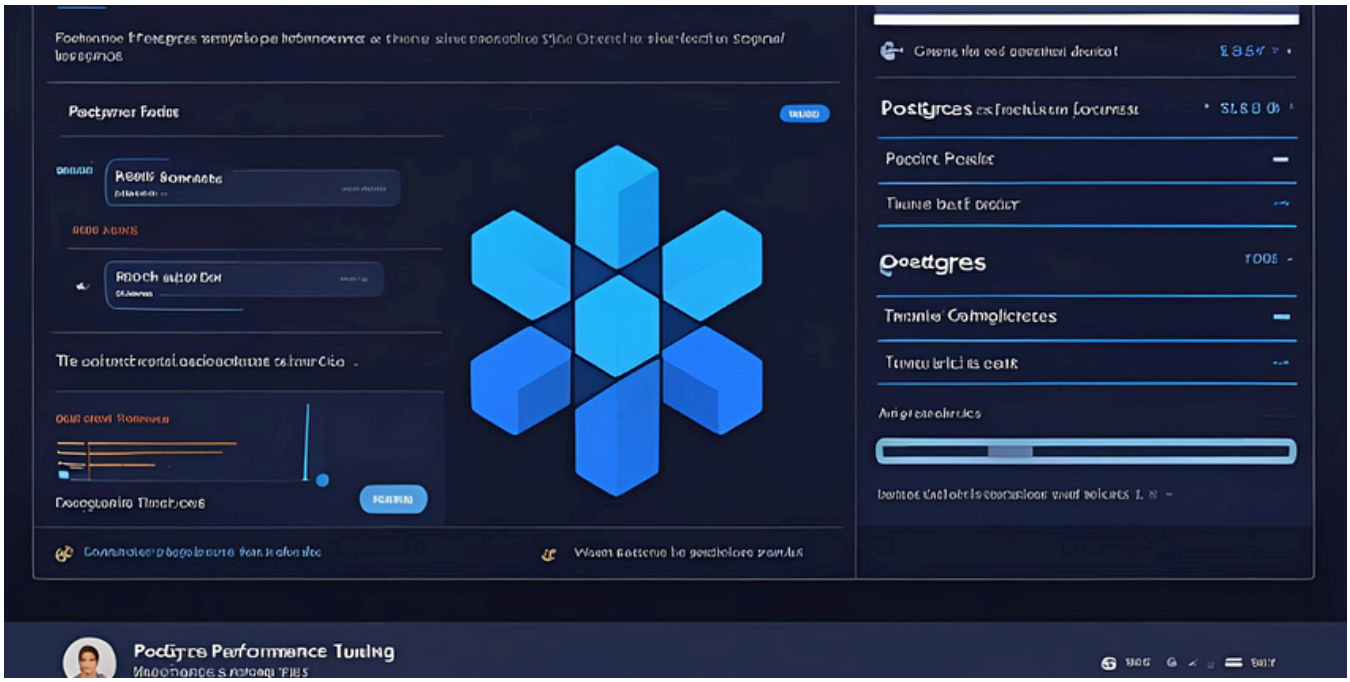
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

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

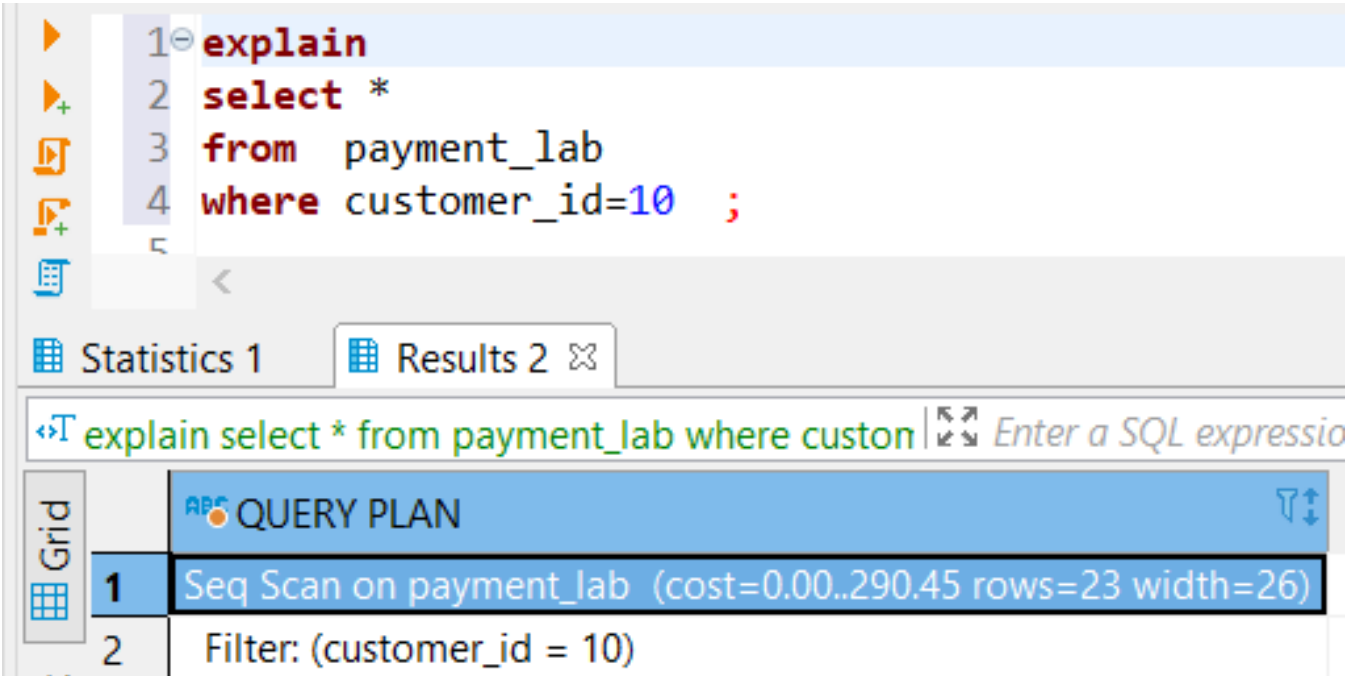
6d ago 55

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

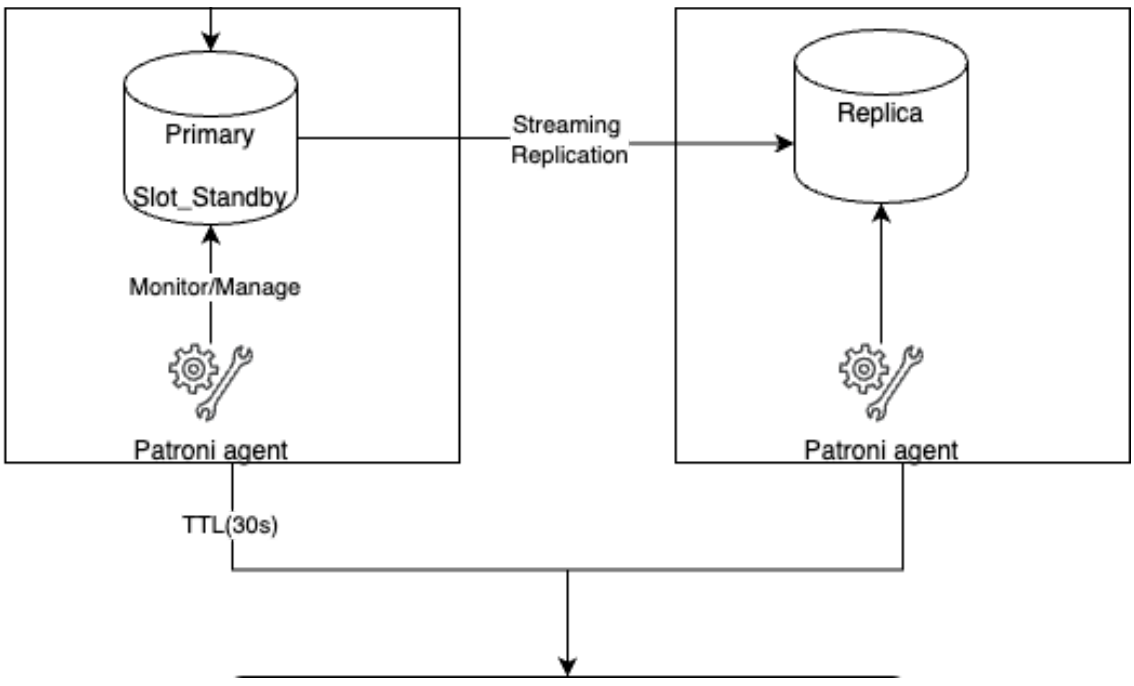


Muhammet Kurtoglu

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



PAWAN SHARMA

PostgreSQL Replication Internals & High Availability with Patroni

PostgreSQL has long been trusted for its reliability and data consistency. But building a production-grade high availability (HA) solution...

Jul 12 2

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

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

See more recommendations