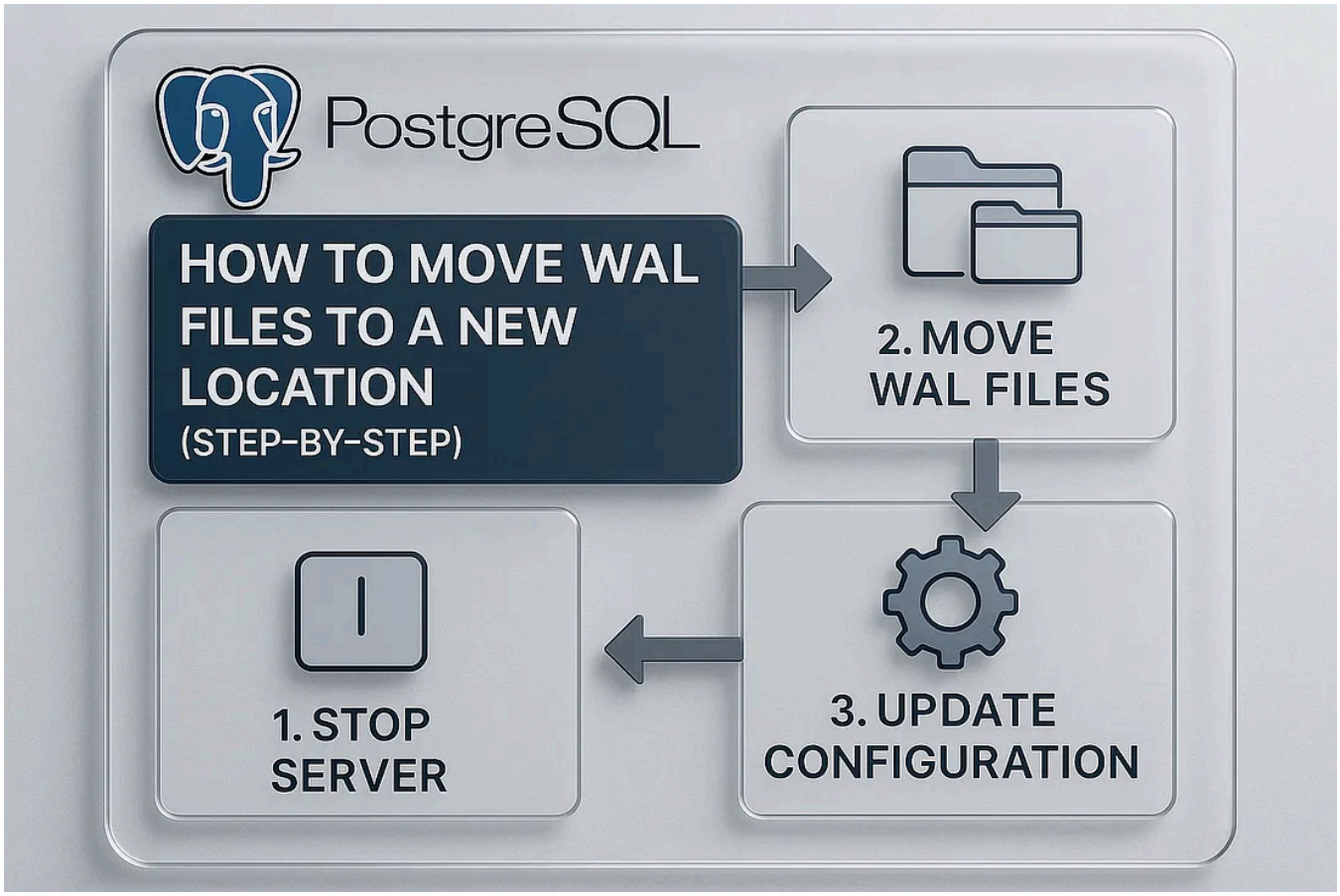
PostgreSQL 17: How to Move WAL Files to a New Location (Step-by-Step)

9 min read · 2 days ago

Jeyaram Ayyalusamy Following



In PostgreSQL 17, Write-Ahead Logging (WAL) plays a critical role in maintaining data integrity, crash recovery, and replication. But as your database grows, you may want to relocate WAL files to a different disk or mount point — for better performance, improved storage management, or hardware optimization.

This guide walks you through exactly how to **safely move WAL files (`pg_wal`)** to a new location in PostgreSQL 17 without compromising your data.

📚 Quick Overview: What Is WAL in PostgreSQL?

Write-Ahead Logging (WAL) is the foundational engine that powers PostgreSQL's reliability and resilience. Here's a closer look at how it works and why it's essential:

📝 What WAL Does

1. 📌 **Ensures Durability**
   Before PostgreSQL writes any change — like an INSERT, UPDATE, or DELETE — to the main data files, it **first records that change in the WAL**. This guarantees durability: even in the event of a crash, all intended writes are safely logged and can be replayed.

2. ⚠️ **Crash Recovery Reliability**
   When PostgreSQL detects an unexpected shutdown, it doesn't rely only on data files. Instead, it uses WAL to reconstruct any committed transaction that hadn't yet been applied to the main data files. This mechanism ensures database consistency and prevents data corruption.

🔄 Why WAL Is Always Used

PostgreSQL uses WAL by default — it's integral to every database operation:

- **Recovery after crashes:** Without WAL, any sudden crash could corrupt the database or lose data.
- **Replication deployments:** WAL is the conduit through which primary and standby servers stay in sync.
- **Point-in-Time Recovery (PITR):** Archiving WAL segments enables faithfully restoring the database to any previous moment in time.

⚙️ More Than Just Crash Recovery

While initially intended for crash recovery, WAL supports a broader set of core PostgreSQL features:

- 🔁 **Streaming Replication**
  WAL updates are continuously shipped from the primary to replica servers, ensuring real-time data synchronization.
- 🕰️ **Point-in-Time Recovery (PITR)**
  By combining a base backup with archived WAL logs, you can restore the database to any moment — seconds before a failure or error.
- 📦 **Continuous Archiving**
  Storing WAL segments in secure, durable locations forms the backbone of robust backup and disaster recovery strategies.

📁 Where WAL Files Are Stored

All WAL segments reside in your data directory under:

```
$PGDATA/pg_wal
```

The directory typically contains:

- **Active WAL files** (16 MB each by default).
- **Archive status markers** — `.ready` (waiting to be archived) and `.done` (successfully archived).

🧠 In Summary

- **WAL = durability + consistency**: Every change is first logged before being applied.
- It's **enabled by default** and underpins recovery, replication, and PITR.
- WAL segments are **stored in** `$PGDATA/pg_wal` and help reconstruct or replicate your database.

Writing this compact yet detailed overview at the beginning of your article provides essential context for readers before diving deeper into WAL archiving, configuration, or recovery.

🔧 Why Move WAL to a New Location?

Shifting the Write-Ahead Log (WAL) to a dedicated storage path brings several strategic advantages:

✅ 1. Separate WAL I/O from Data I/O

Having WAL writes on their own disk or partition prevents them from competing with regular data operations. This reduces I/O contention, ensuring both transaction logging and user queries run smoothly.

✅ 2. Improve Disk Performance & Durability

WAL use is inherently sequential — perfect for high-performance RAID arrays or SSDs. By storing WAL on durable, efficiently written storage, you minimize latency for every write transaction, boosting overall system responsiveness.

✅ 3. Expand Storage Capacity

If your main database storage is nearing capacity, offloading WAL files to a different disk, volume, or partition instantly frees up space. This gives you more room for backups, snapshots, or temporary workloads without touching the primary data directory.

✅ 4. Protect Against Filesystem Saturation

WAL can quickly consume disk space under heavy write activity. When WAL and data share the same filesystem, you risk filling it up and triggering critical failures. A separate filesystem isolates this risk — keeping your primary database stable even under extreme loads.

✅ 5. Reduce Checkpoint Stalls Under Load

During checkpoints, SQLite regularly flushes dirty pages to disk. If WAL and data writes share a drive, this process can interfere, causing significant latency. A separate WAL location decouples these operations and ensures checkpoints finish swiftly — even during peak activity.

⚠️ Prerequisites Before You Start

Before relocating WAL, make sure your environment fulfills the following conditions:

1. ✅ **PostgreSQL 17 installed and configured**
   Ensure your instance is running and WAL archiving (if enabled) is working as expected.
2. ✅ **New filesystem or disk is mounted and available**
   This could be a new EBS volume, LVM logical volume, or a secondary SSD/NVMe drive — mounted and writable by the OS.
3. ✅ **Permissions set for the `postgres` service user**
   The WAL directory must be owned by the `postgres` user and group. Use `chown postgres:postgres` and `chmod 700` or tighter to secure it.
4. ✅ **Database backup is recommended**
   Always take a base backup before system-level changes like WAL relocation. This ensures a recovery path if things go wrong during migration.

🔨 Step-by-Step WAL Relocation Process

Relocating your WAL (Write-Ahead Log) directory can improve performance and stability by decoupling logging I/O from data I/O. Here's how to do it in a detailed, safe, and step-by-step manner:

1️⃣ Locate Your Existing WAL Directory

Before you start, confirm the current location of your WAL files. This is typically the `pg_wal` folder inside your data directory.

Run:

```
echo $PGDATA/pg_wal
```

The output is usually:

```
/var/lib/pgsql/17/data/pg_wal
```

Verifying this ensures you're working with the correct paths instead of guessing based on defaults.

2️⃣ Create the New WAL Directory

Choose a new storage location dedicated to WAL — for instance, a larger disk or a faster SSD:

```
sudo mkdir -p /var/lib/pgsql/pg_wal_new
```

Then, set secure ownership:

```
sudo chown postgres:postgres /var/lib/pgsql/pg_wal_new
sudo chmod 700 /var/lib/pgsql/pg_wal_new
```

- `-p` : creates parent directories if needed.
- **Ownership (** `postgres:postgres` **):** ensures only the PostgreSQL process can write to it.
- `chmod 700` : gives full access to the postgres user and locks others out.

✅ Why These Steps Matter

- **Accuracy matters:** Misidentifying the WAL directory could cause data integrity issues if the wrong directory is moved or deleted.
- **Security & reliability:** Proper permissions ensure WAL files are not accidentally modified, deleted, or tampered with.
- **Isolation for performance:** By dedicating a filesystem or disk to WAL, you're optimizing disk I/O — leading to better overall database responsiveness and fewer checkpoint stalls.

3️⃣ Stop PostgreSQL Safely

Before moving WAL data, **shut down PostgreSQL** to prevent any active writes or corruption:

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

- The first command stops the service.
- The second confirms it's fully stopped — look for output indicating it's inactive.
- This precaution prevents inconsistencies from files being modified during transfer.

4️⃣ Copy WAL Files to the New Location

With PostgreSQL down, you can now transfer WAL files to your new directory. Use these robust methods to preserve integrity and performance:

✅ Option A: Use `rsync`

```
sudo rsync -av /var/lib/pgsql/17/data/pg_wal/ /var/lib/pgsql/pg_wal_new/
```

- `-a` ensures archive mode (preserves permissions, owners, timestamps).
- `-v` gives a summary of transferred files.
- Efficiently transfers only new or changed files if rerun.

⚠️ Option B: Use `cp`

Alternatively, use a straightforward copy:

```
cd /var/lib/pgsql/17/data/pg_wal
sudo cp -R * /var/lib/pgsql/pg_wal_new/
```

- `cp -R` : copies recursively.
- Less efficient than `rsync` , but gets the job done.

✅ Verification

After the copy finishes, double-check the migrated content:

```
ls -la /var/lib/pgsql/pg_wal_new
```

- Confirm ownership remains `postgres:postgres` .

- Ensure all WAL segment files (and any status markers) appear correctly.
- If files are missing or permissions look wrong, fix and re-copy before proceeding.

🧠 Why These Steps Matter

- **Clean shutdown** ensures pause in WAL generation, keeping files consistent.
- **Using `rsync` is best practice** for large or incremental copies—preventing unnecessary data transfer.
- **Verification builds confidence** that everything is in place before updating PostgreSQL's configuration.

5️⃣ Backup the Original WAL Directory

Before making any changes, protect against unexpected issues by renaming (rather than deleting) the current WAL directory:

```
sudo mv /var/lib/pgsql/17/data/pg_wal /var/lib/pgsql/17/data/pg_wal_bkp
```

This preserves the original files under a new name, giving you a fallback in case issues arise.

✅ Verify the Rename

Run:

```
ls -la /var/lib/pgsql/17/data/
```

Look for the backup directory, e.g.:

```
drwx------ 2 postgres postgres 4096 Jun 14 12:34 pg_wal_bkp
```

If it appears with the correct permissions, you're safe to continue.

6️⃣ Create a Symbolic Link to the New WAL Directory

Instead of editing PostgreSQL's configuration, a **symlink** seamlessly directs WAL writes to their new location:

```
cd /var/lib/pgsql/17/data/
sudo ln -s /var/lib/pgsql/pg_wal_new pg_wal
```

- This creates `pg_wal` in the data directory pointing to `/var/lib/pgsql/pg_wal_new`.
- PostgreSQL continues its normal routines, oblivious to the change.

✅ Confirm Link Creation

Use:

```
ls -lrt
```

You should see something like:

```
lrwxrwxrwx 1 postgres postgres   20 Jun 14 12:45 pg_wal -> /var/lib/pgsql/pg_wa
drwx------ 2 postgres postgres 4096 Jun 14 12:34 pg_wal_bkp
```

This confirms the symbolic link is active, and your old WAL folder remains safely backed up.

🛡️ Why These Steps Matter

Action	Purpose
Backup original directory	Safeguards against migration glitches or corrupted data during the switch
Use symlink instead of config change	Provides a non-intrusive method that keeps PostgreSQL configuration intact
Verification steps	Ensures the symlink and backup are correctly set before restarting the database

By doing this, you maintain a clean rollback path and make the relocation transparent to PostgreSQL — and future upgrades or troubleshooting become much simpler.

7️⃣ Restart PostgreSQL

Now that everything is in place, start the PostgreSQL service:

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

Then check its status:

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

- Look for lines like **“active (running)”**.
- Ensure there are no error messages regarding WAL, such as missing directories or permission issues.

✅ If the service starts cleanly, it means PostgreSQL is now writing WAL to the new location through the symlink.

8️⃣ Validate PostgreSQL Access

Confirm that the database is operational and accessible:

```
psql -h localhost -U postgres -p 5432
```

Inside `psql`, you might run simple verification commands, such as:

```
\conninfo
SELECT now();
```

This confirms that the database accepts connections and performs basic queries — indicating full functionality post-migration.

9️⃣ (Optional) Clean Up Backup Directory

After you've verified WAL files are being written to the new directory and the database is operating normally, you can **remove the old backup WAL directory**:

```
sudo rm -rf /var/lib/pgsql/17/data/pg_wal_bkp
```

⚠️ Only do this once you're absolutely certain the new WAL location is working properly. Deleting it too soon could leave you without a fallback.

📊 Key WAL Facts Every PostgreSQL DBA Should Know

1. **WAL is always enabled by default**
   PostgreSQL uses Write-Ahead Logging for every change — there's no way to turn it off. It's integral to ensuring data durability and consistency.

2. **Binary change logging**
   Instead of writing directly to data files, changes are first logged in a compact, binary WAL format. This approach provides atomicity and speeds up recovery.

3. **Log Sequence Number (LSN)**
   Every WAL record is tagged with an LSN — an internal offset that orders operations precisely. LSNs enable accurate replay during recovery and replication.

4. **Storage location**
   WAL files reside in:

```
$PGDATA/pg_wal
```

1. This dedicated directory ensures WAL is managed separately from table and index data.

2. **Performance benefits of separate disks**
   Writing WAL to a separate, fast disk (such as an SSD) significantly boosts transaction throughput. This is because it isolates sequential WAL writes from random I/O on data files, preventing contention.

3. **Dependence on accurate filesystem behavior**
   WAL's reliability depends on the underlying filesystem — calls like `fsync()` and `fdatasync()` must work correctly. If the file system misreports or fails, WAL guarantees can be compromised.

📝 Summary

- ✅ **Relocating WAL safely** can optimize I/O and improve overall database performance.

- ✅ **Symbolic links** allow you to move WAL storage without rewriting configuration settings.
- ✅ **Always stop PostgreSQL** before performing file operations to ensure data safety.
- ✅ **Validate and monitor** after relocation to prevent subtle issues.

Proper WAL storage management isn't just a performance tweak — it's a critical part of scaling reliability and recovery for large-scale PostgreSQL environments. By following these best practices, you'll ensure faster transactions, safer backups, and peace of mind.

📢 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 Open Source Oracle Sql

J

Following

Written by Jeyaram Ayyalusamy

38 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 Kernel Tuning Guide: Managing System Parameters for Optimal Performance

PostgreSQL is a highly performant database system, but to fully leverage its power—especially at scale—proper tuning at the kernel...

6d ago 3 2

Jeyaram Ayyalusamy

Mastering PostgreSQL Configuration: A Deep Dive into postgresql.conf

PostgreSQL is one of the most powerful open-source databases in the world, but much of its true potential lies hidden in its configuration...

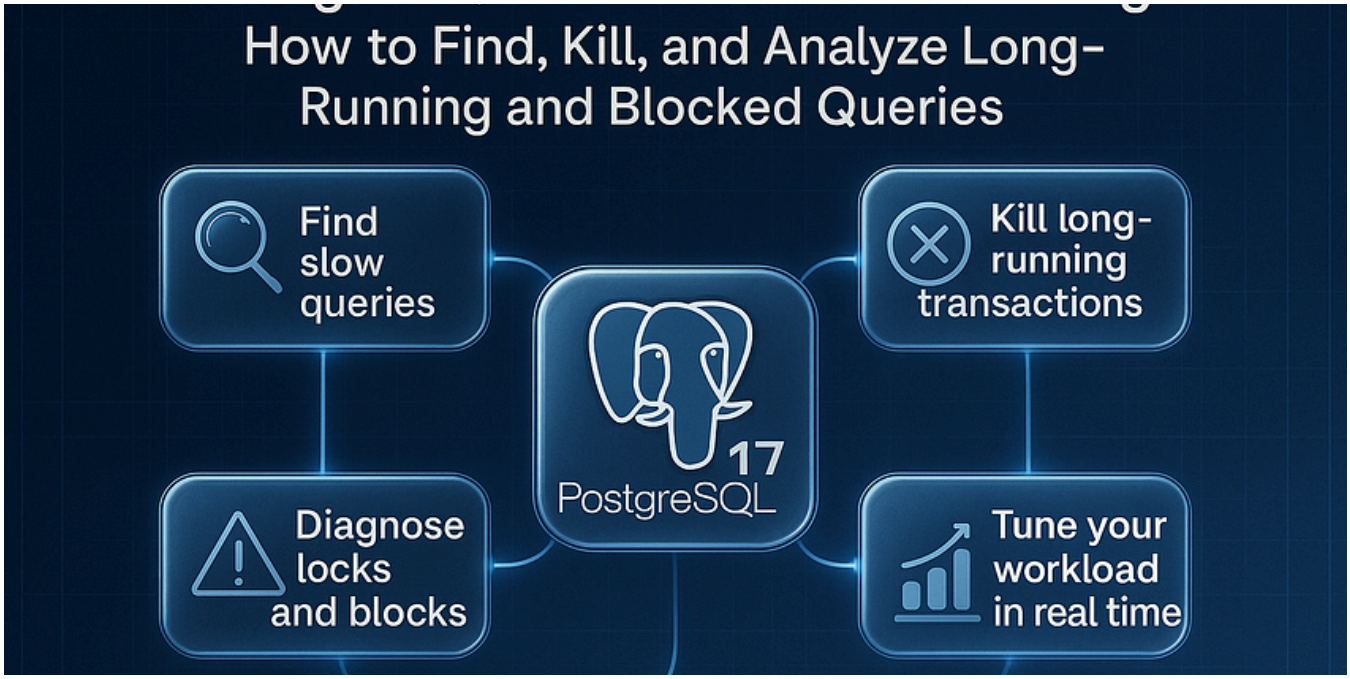
Jun 8 2

Jeyaram Ayyalusamy

PostgreSQL MVCC: The Secret Behind Its Powerful Concurrency

PostgreSQL is renowned for its performance, consistency, and concurrency. But at the core of this magic lies one of its most important —...

Jun 11 3 1



Jeyaram Ayyalusamy

PostgreSQL 17 Performance Tuning: How to Find, Kill, and Analyze Long-Running and Blocked Queries

Keeping your PostgreSQL database fast, efficient, and healthy isn't magic—it's all about proactive monitoring.

Jun 14 1

See all from Jeyaram Ayyalusamy

Recommended from Medium

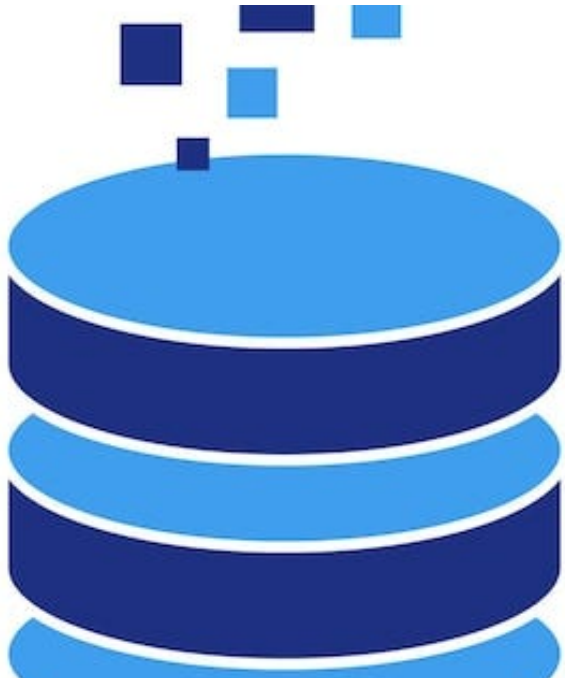
DIFFERENCE	WINNER	QUERY TYPE	CASE
~35% faster	Subquery	Aggregation Filter	1
~20% faster	Subquery	Nested Aggregation	2
±5%	Tie	Join + Filter	3
~10% faster	CTE	Complex Join Tree	4
~40% faster	CTE	Reuse in Pipeline	5

Mojtaba Azad

PostgreSQL CTEs vs. Subqueries: I Benchmarked 5 Real Cases So You Don't Have To

CTEs are readable—but are they always the right tool? I ran tests so you don't have to guess.

Jun 14 2



Priyanshu Rajput

🚀 PostgreSQL Index Internals Demystified: B-Tree, GIN, GiST, BRIN Made Ridiculously Simple

If PostgreSQL were a superhero team, indexes would be the sidekicks doing all the heavy lifting—quietly, efficiently, and without much...

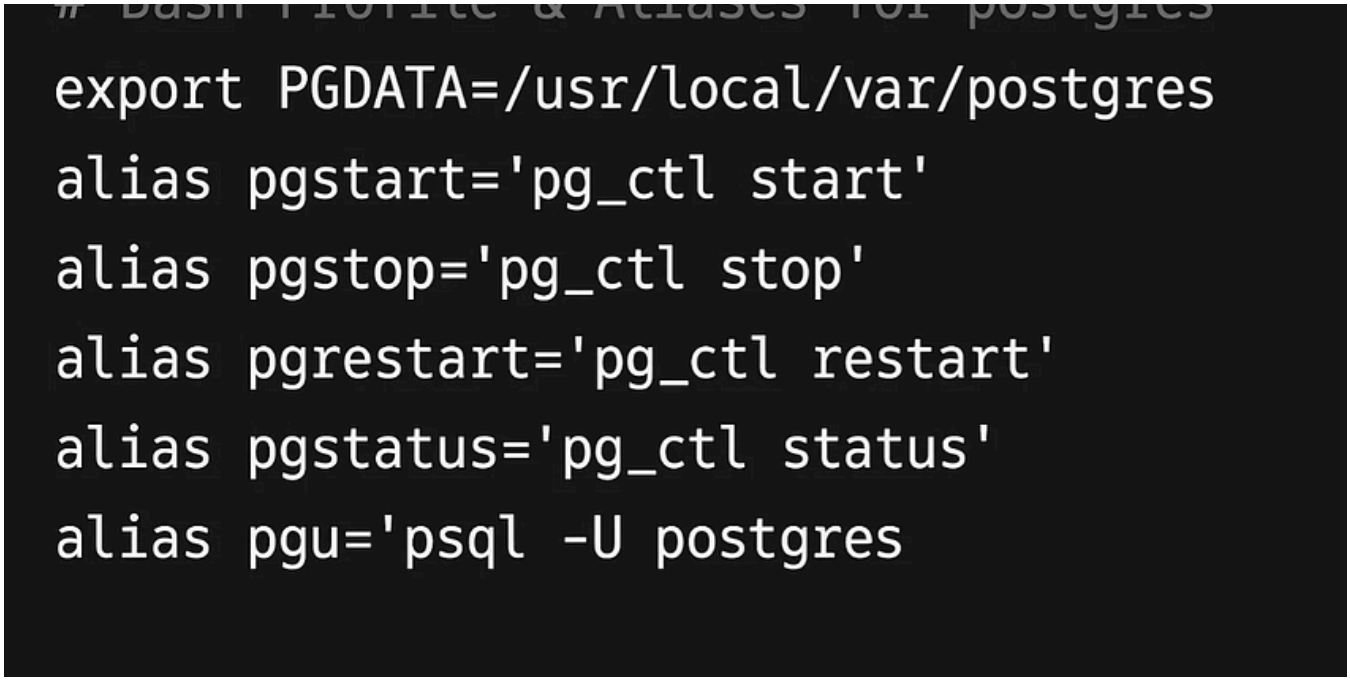
May 16 9 1

Rizqi Mulki

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

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

Apr 19 17

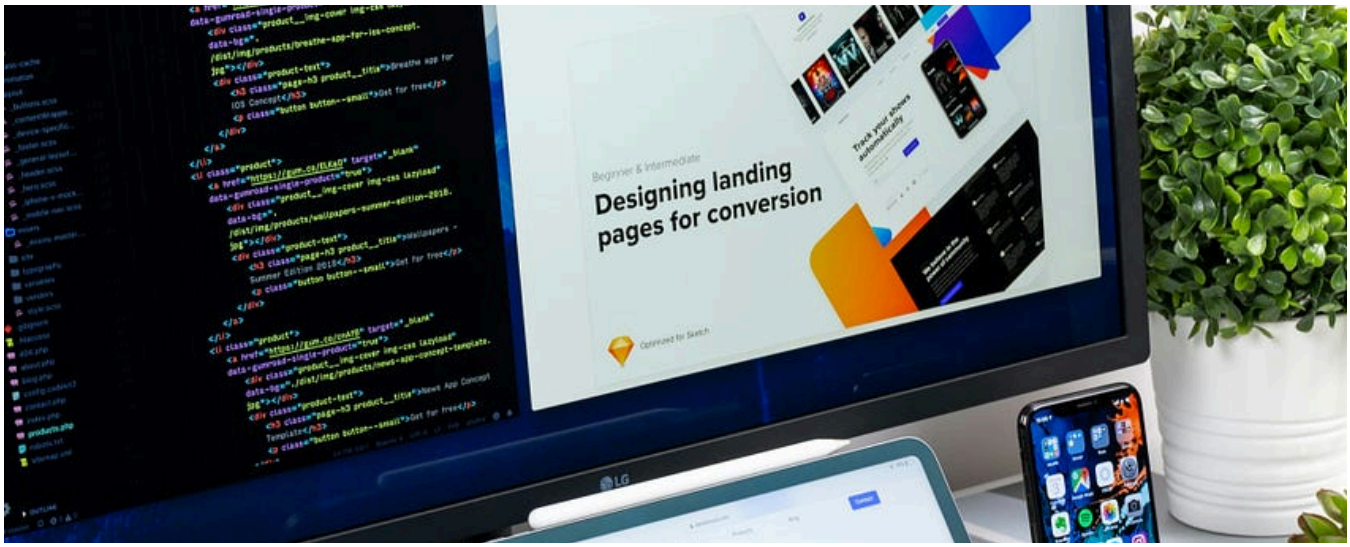


Oz

Bash Profile & Aliases for Postgres

Managing PostgreSQL on Linux? You're probably juggling between dozens of commands, logs, and scripts. With a few tweaks to your...

Jun 17 53



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

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

Jun 11 10 1

Ajaymaurya

pg_dump vs. pg_basebackup vs. pg_dumpall: Which PostgreSQL Backup Tool Should You Use?

When it comes to backing up your PostgreSQL databases, one size definitely doesn’t fit all. Depending on your setup, scale, and recovery...

Jun 13 2

See more recommendations