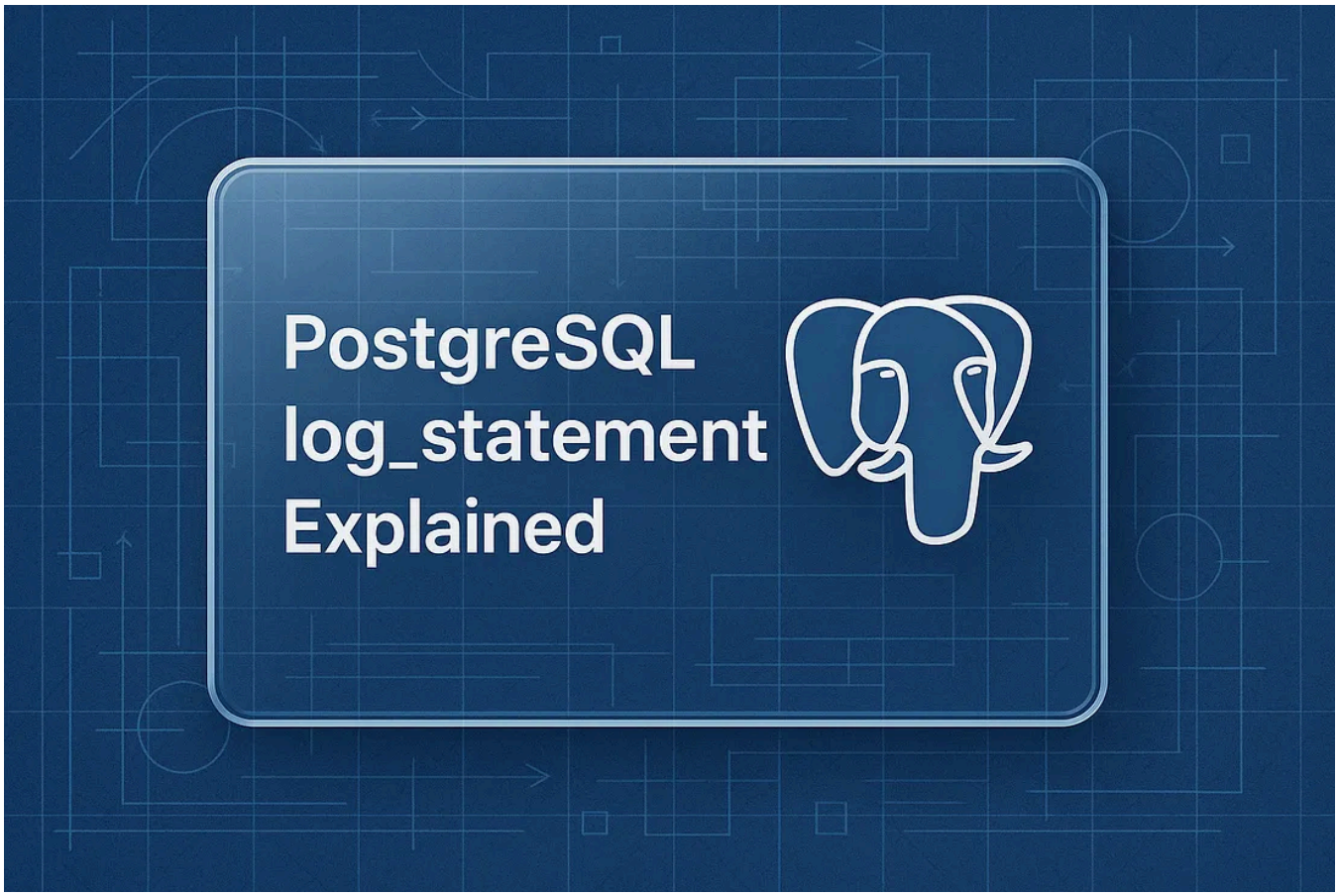
PostgreSQL `log_statement` Explained: Complete Guide With Real-Time Examples

14 min read · Jun 15, 2025

J Jeyaram Ayyalusamy Following



📘 Understanding `log_statement` in PostgreSQL: Your Key to Query Visibility

PostgreSQL is celebrated not only for its reliability and performance — but also for its **robust logging system**. Whether you're troubleshooting performance issues,

tracking user activity, or performing compliance audits, **query-level logging** is often your first and most powerful line of insight.

One of the **most critical logging parameters** in PostgreSQL — yet surprisingly underutilized — is `log_statement`.

In this article, we'll explore:

- What `log_statement` does
- How to configure it in PostgreSQL 13+
- Its different logging levels and when to use them
- Real-world examples that show how this parameter can make or break your observability strategy

Let's dive in.

🔍 What is `log_statement`?

`log_statement` is a PostgreSQL configuration parameter that **controls which SQL statements get logged** to the PostgreSQL log file.

Think of it as a filter that determines **how verbose** the query logs should be. It's incredibly helpful for:

- Monitoring application behavior
- Debugging complex interactions
- Auditing query patterns for compliance
- Detecting unexpected or malicious SQL activity

But with great power comes great responsibility — setting it too aggressively can flood your logs with noise, while setting it too narrowly may cause you to miss important insights.

⚙️ How to Configure `log_statement`

You can set `log_statement` in:

- The `postgresql.conf` file (permanent)
- Via SQL using `ALTER SYSTEM` (requires reload)
- Temporarily per session (for debugging)

Example (in `postgresql.conf`):

```
log_statement = 'all'
```

Then reload the server for changes to take effect:

```
SELECT pg_reload_conf();
```

🧰 The Four Logging Levels

PostgreSQL offers **four options** for `log_statement`, each increasing in verbosity:

1. `none` (default)

- No SQL statements are logged.
- Best for production systems with separate query logging tools.

2. `ddl`

- Logs only DDL statements such as `CREATE`, `ALTER`, `DROP`.
- Useful for tracking **schema changes** without the noise of everyday queries.

3. `mod`

- Logs all DDL plus **INSERT**, **UPDATE**, **DELETE**, and **TRUNCATE**.

- Great for tracking **data mutations** without logging every `SELECT`.

4. `all`

- Logs **everything**, including every `SELECT`.
- Ideal for **debugging** or during **security reviews**, but can generate large logs quickly.

🧠 **Tip:** Pair `log_statement` with `log_duration` or `log_min_duration_statement` to capture slow queries only.

💡 Real-World Example

Let's say you set:

```
log_statement = 'mod'
```

Then run the following:

```
SELECT * FROM customers; -- not logged
UPDATE customers SET active = false WHERE last_login < now() - interval '1 year
```

Only the `UPDATE` statement appears in your logs, because `log_statement = 'mod'` includes data-modifying queries, but **excludes read-only SELECTs.**

Now change it to:

```
log_statement = 'all'
```

Rerun the `SELECT` —now it's logged too.

This gives you full traceability of what's happening in your database at the query level.

🔐 Security & Compliance Use Cases

If you're in a **regulated environment** (e.g., healthcare, fintech), `log_statement` can play a key role in:

- **Auditing who did what**
- **Tracking schema changes**
- **Detecting unauthorized access patterns**

You can even combine it with **log_line_prefix** for detailed, timestamped logs with user info, session ID, and application name.

🚀 Final Thoughts

The `log_statement` parameter might seem like a minor setting, but it's a **powerful gateway to understanding your PostgreSQL workload.**

By setting the right logging level, you can:

- Debug faster
- Audit smarter
- Monitor more effectively

Whether you're chasing a runaway query or analyzing long-term behavior trends, `log_statement` gives you the **raw query data** you need to stay in control.

🔎 What Is `log_statement` in PostgreSQL?

When it comes to understanding what's happening inside your PostgreSQL database, logging is your best friend. One of the most useful — but often overlooked — logging features is the `log_statement` parameter.

💡 So, what is it?

The `log_statement` parameter in PostgreSQL **controls which SQL statements are written to the server log.** It acts as a filter to determine how much query-related information should be recorded.

This setting is especially valuable for:

- ✅ **Debugging:** Want to trace what a client or app is doing behind the scenes? This logs the actual SQL.
- ✅ **Auditing:** Need to track who changed what and when? Enable detailed logs.
- ✅ **Compliance:** In regulated industries (e.g., finance, healthcare), logging query history is a requirement.

With `log_statement`, you decide **how much detail** gets logged—from nothing at all to every single SQL command.

⚙ Default Behavior of `log_statement`

Out of the box, PostgreSQL is tuned for **performance**, not verbosity. That's why the default setting is:

```
log_statement = 'none'
```

Here's what this means:

- ✅ **No queries are logged** to the PostgreSQL log file
- ✅ **Better performance**, since there's minimal I/O overhead from logging
- 🚫 **Not ideal for troubleshooting**, since you can't see what queries were run
- 🚫 **Useless for audit trails**, as there's no visibility into user actions

This default makes sense for high-throughput environments where performance is critical — but for development, staging, or security-sensitive environments, you'll likely want to **increase the logging level**.

📌 When Should You Change It?

If you're:

- Debugging application behavior
- Monitoring data changes
- Investigating performance issues
- Auditing database usage

Then it's time to consider a more verbose setting like `'mod'` or `'all'`.

We'll cover the available options (`none`, `ddl`, `mod`, `all`) and how to apply them in the next section.

🔧 `log_statement` Options in PostgreSQL (With Use Cases)

PostgreSQL gives you granular control over how much SQL logging you want by offering multiple options for the `log_statement` parameter. Each level captures a different category of SQL statements, allowing you to strike the right balance between visibility and performance.

Let's explore each option in detail — with real-world use cases to help you choose the right setting for your environment.

1️⃣ `none` — No SQL statements are logged

```
log_statement = 'none'  -- Default setting
```

- 🧘 **Description:** Completely disables query-level logging.
- ⚙️ **Performance:** Very lightweight and optimal for high-throughput systems.
- 🚫 **Drawback:** You won't see any SQL activity in your logs, which makes debugging or auditing extremely difficult.

✅ **Use Case:**

- Ideal for **production environments** where performance is a top priority and other monitoring tools (like pgBadger or APMs) are already in place.
- Not suitable when troubleshooting or when visibility into query behavior is required.

2️⃣ `ddl` — Logs only DDL (schema) statements

```
log_statement = 'ddl'
```

- 🧱 **Description:** Logs **Data Definition Language** commands like `CREATE`, `ALTER`, `DROP`, and so on.
- 🔍 **Visibility:** Tracks structural changes without overwhelming the logs with everyday queries.

✅ **Use Case:**

- Great for **schema change auditing**, especially in teams where multiple developers work on the same database.
- Helps during **migrations or upgrades** to ensure no unintended schema modifications occur.

3️⃣ `mod` — Logs DDL + modifying DML statements

```
log_statement = 'mod'
```

- 🖍️ **Description:** Logs all DDL commands *plus* **data-modifying** operations like `INSERT`, `UPDATE`, `DELETE`, and `TRUNCATE`.
- 🔄 **Tracking Changes:** Gives you insight into any command that changes the state of your data.

✅ **Use Case:**

- Perfect for **auditing data changes** in staging or QA environments.
- Useful for **security reviews** where tracking who changed what data (and when) is essential.
- Strikes a good balance between visibility and log size for many production environments.

4️⃣ `all` — Logs everything, including SELECT

```
log_statement = 'all'
```

- 🧠 **Description:** The most verbose option — logs **every single SQL statement**, including read-only operations like `SELECT`.
- 🔎 **Full Transparency:** Allows complete visibility into query behavior, execution patterns, and user actions.

✅ **Use Case:**

- Best for **debugging** application behavior or performance issues in development environments.

- Valuable during **code testing, profiling, or training environments** where full query traceability is needed.

⚠️ **Warning:** This setting can generate **very large log files**, especially in read-heavy workloads. Use with caution in production environments.

📘 Summary Table

Option	Logs What?	Recommended Use
`none`	Nothing	High-performance production with external monitoring
`ddl`	Schema changes only	Auditing migrations, low-noise logging
`mod`	DDL + INSERT/UPDATE/DELETE	Tracking data changes, QA and audit
`all`	Every SQL statement	Development, debugging, full traceability

✅ Final Thoughts

Choosing the right `log_statement` level is all about **your environment's needs.** Use `none` when performance matters most, `mod` for balanced auditing, and `all` when you want to see everything during development or debugging.

PostgreSQL gives you the flexibility — **you just need to tune it wisely.**

💻 How `log_statement` Works in PostgreSQL

Every query executed in PostgreSQL passes through its internal **logging pipeline.** However, **only queries that match the `log_statement` level** get written to the PostgreSQL server logs.

This filtering mechanism helps database administrators and developers fine-tune how much visibility they need — without overwhelming the system with unnecessary log volume.

⚙️ How to Set `log_statement`

There are two main ways to configure the `log_statement` parameter:

1. **Via `postgresql.conf` file (permanent setting)**
   Located typically at:

```
/var/lib/pgsql/{version}/data/postgresql.conf
```

1. You'll need to edit the file and restart or reload PostgreSQL.
2. **Dynamically with SQL (no restart needed)**
   Use the following SQL command for immediate effect:

```
ALTER SYSTEM SET log_statement = 'mod';  -- Or 'ddl', 'all', etc.
SELECT pg_reload_conf();                 -- Apply the change
```

📁 Where Are PostgreSQL Logs Stored?

By default, PostgreSQL writes its log files to:

```
/var/lib/pgsql/{version}/data/log/postgresql.log
```

You can monitor logs in real time using:

```
[postgres@ip-172-31-20-155 ~]$ cd /var/lib/pgsql/17/data/log/
[postgres@ip-172-31-20-155 log]$
[postgres@ip-172-31-20-155 log]$ ls -ltr
total 8
-rw-------. 1 postgres postgres 2519 Jun 21 16:23 postgresql-Sat.log
-rw-------. 1 postgres postgres    0 Jun 22 00:00 postgresql-Sun.log
-rw-------. 1 postgres postgres  817 Jun 23 23:30 postgresql-Mon.log
-rw-------. 1 postgres postgres    0 Jun 24 00:00 postgresql-Tue.log
[postgres@ip-172-31-20-155 log]$

[postgres@ip-172-31-20-155 log]$ cat postgresql-Mon.log
2025-06-23 23:28:57.501 UTC [22503] ERROR:  invalid input syntax for type inet:
2025-06-23 23:28:57.501 UTC [22503] STATEMENT:  SELECT COUNT(*) AS client_conns
        FROM pg_stat_activity
        WHERE client_addr = 'your_ip_here';
2025-06-23 23:29:37.317 UTC [22503] ERROR:  column "query_time" does not exist
2025-06-23 23:29:37.317 UTC [22503] HINT:  Perhaps you meant to reference the c
2025-06-23 23:29:37.317 UTC [22503] STATEMENT:  SELECT
          pid, usename, query_time, query
        FROM pg_stat_activity
        WHERE state = 'active'
          AND (now() - query_start) > interval '5 minutes';
2025-06-23 23:30:58.439 UTC [22503] ERROR:  column "pid" does not exist at char
2025-06-23 23:30:58.439 UTC [22503] STATEMENT:  SELECT pg_cancel_backend(pid);
[postgres@ip-172-31-20-155 log]$
```

This is incredibly useful during development, testing, or incident investigations.

🔬 Hands-On Examples: How Different `log_statement` Levels Behave

Now let's see how PostgreSQL behaves under different `log_statement` settings. These examples will help you **visualize exactly what does and doesn't get logged.**

📌 Example 1: `log_statement = 'none'` (Default)

Check the current setting:

```
postgres=# SHOW log_statement;
 log_statement
---------------
 none
```

Run some SQL commands:

```
CREATE TABLE course (course_no integer, name text, price numeric);
INSERT INTO course (course_no, name, price) VALUES (1, 'postgres', 150);
SELECT * FROM course;
```

✅ **Result:**
None of these statements will appear in the logs. This is expected because `log_statement = 'none'` logs **nothing**.

Check the logs to confirm:

```
tail -30f /var/lib/pgsql/17/data/log/postgresql-Tue.log
```

You won't see any entries related to the above queries.

📌 Example 2: `log_statement = 'ddl'` (Schema Changes Only)

Now let's enable logging for DDL statements (schema changes):

```
ALTER SYSTEM SET log_statement = 'ddl';
SELECT pg_reload_conf();  -- Reload without restart
```

```
postgres=# ALTER SYSTEM SET log_statement = 'ddl';
ALTER SYSTEM
```

```
postgres=#
postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)

postgres=#
```

Run the same set of queries:

```
CREATE TABLE course (course_no integer, name text, price numeric);
INSERT INTO course (course_no, name, price) VALUES (1, 'databricks', 2000);
SELECT * FROM course;
```

```
postgres=# CREATE TABLE course (course_no integer, name text, price numeric);
CREATE TABLE
postgres=#
postgres=#
postgres=# INSERT INTO course (course_no, name, price) VALUES (1, 'databricks',
INSERT 0 1
postgres=#
postgres=# SELECT * FROM course;
 course_no |   name     | price
-----------+------------+-------
         1 | databricks |  2000
(1 row)

postgres=#
```

✅ **Result:**

Only the `CREATE TABLE` command will be logged. The `INSERT` and `SELECT` statements will not.

Confirm by checking the logs:

```
tail -10f /var/lib/pgsql/17/data/log/postgresql-Sat.log
```

You should see a log entry similar to:

```
[postgres@ip-172-31-20-155 log]$ tail -30f /var/lib/pgsql/17/data/log/postgresc
2025-06-24 00:39:22.566 UTC [22935] ERROR:  syntax error at or near "]" at char
2025-06-24 00:39:22.566 UTC [22935] STATEMENT:  ]
        ALTER SYSTEM SET log_statement = 'ddl';
2025-06-24 00:39:33.477 UTC [1823] LOG:  received SIGHUP, reloading configurati
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_vacuum_scale_fac
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_analyze_scale_fa
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_max_workers" car
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_naptime" changed
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "log_statement" changed to "
2025-06-24 00:39:33.478 UTC [1823] LOG:  configuration file "/var/lib/pgsql/17/
2025-06-24 00:40:16.029 UTC [22935] LOG:  statement: CREATE TABLE course (cours
```

But no entries for the `INSERT` or `SELECT`.

🧠 Why This Matters

- Setting the right `log_statement` level gives you **precise control** over your query visibility.
- It enables **auditing and troubleshooting** without flooding your logs.
- Knowing what gets logged (and what doesn't) helps you **interpret logs more accurately** during real-world operations.

In the next part, we'll look at more examples using `mod` and `all` to see how you can log data-changing and read-only queries too.

📌 Example 3: `log_statement = 'mod'` — Logging DDL and Modifying DML

If you want to capture **data-changing activity** in your PostgreSQL logs without logging every `SELECT`, the `mod` setting is your go-to option.

Let's enable it:

```
ALTER SYSTEM SET log_statement = 'mod';
SELECT pg_reload_conf();  -- Applies the change without restart
```

Now, run the following SQL commands:

```
INSERT INTO course (course_no, name, price) VALUES (2, 'rds mysql', 5000);
DELETE FROM course WHERE course_no = 2;
```

You should see a log entry similar to:

```
postgres=# ALTER SYSTEM SET log_statement = 'mod';
ALTER SYSTEM
postgres=#
postgres=# INSERT INTO course (course_no, name, price) VALUES (2, 'rds mysql',
INSERT 0 1
postgres=#
postgres=#
postgres=# DELETE FROM course WHERE course_no = 2;
DELETE 1
postgres=#
postgres=#
```

✅ Result:

Both the `INSERT` and `DELETE` operations will be logged, because they modify data.

However, if you run a `SELECT` query like:

```
SELECT * FROM course;
```

```
postgres=# SELECT * FROM course;
course_no  |   name     | price
-----------+------------+-------
         1 | databricks |  2000
(1 row)

postgres=#
postgres=#
```

It **will not** be logged — because `mod` doesn't log read-only operations.

📝 Summary:

- ✔️ **Logs:** `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and all `DDL` (e.g., `CREATE`, `ALTER`)
- ❌ **Does not log:** `SELECT` (unless it triggers a data-modifying function)

Use this setting if you want visibility into **data mutations** without the performance hit of logging every single query.

📌 Example 4: `log_statement = 'all'` — Logging Every Query

Need full visibility into every query hitting your PostgreSQL instance? Set `log_statement` to `all`:

```
ALTER SYSTEM SET log_statement = 'all';
SELECT pg_reload_conf();  -- Reload the config
```

```
postgres=# SELECT * FROM course;
course_no  |   name     | price
-----------+------------+-------
         1 | databricks |  2000
(1 row)
```

```
postgres=#
postgres=# ALTER SYSTEM SET log_statement = 'all';
ALTER SYSTEM
postgres=#
postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)

postgres=#
```

Now, run a few sample queries:

```
INSERT INTO course (course_no, name, price) VALUES (3, 'oracle', 10000);
SELECT * FROM course;
```

```
postgres=# INSERT INTO course (course_no, name, price) VALUES (3, 'oracle', 100
INSERT 0 1
postgres=#
postgres=# SELECT * FROM course;
course_no  |   name     | price
-----------+------------+-------
         1 | databricks |  2000
         1 | oracle     |  10000
(2 rows)

postgres=#
postgres=#
```

✅ Result:

Every query — yes, even simple `SELECT` statements—will be written to the PostgreSQL logs.

Check it with:

```
tail -30f /var/lib/pgsql/17/data/log/postgresql-Tue.log
```

You'll see log entries like:

```
2025-06-24 00:39:22.566 UTC [22935] ERROR:  syntax error at or near "]" at char
2025-06-24 00:39:22.566 UTC [22935] STATEMENT:  ]
        ALTER SYSTEM SET log_statement = 'ddl';
2025-06-24 00:39:33.477 UTC [1823] LOG:  received SIGHUP, reloading configurati
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_vacuum_scale_fac
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_analyze_scale_fa
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_max_workers" car
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "autovacuum_naptime" changed
2025-06-24 00:39:33.478 UTC [1823] LOG:  parameter "log_statement" changed to "
2025-06-24 00:39:33.478 UTC [1823] LOG:  configuration file "/var/lib/pgsql/17/
2025-06-24 00:40:16.029 UTC [22935] LOG:  statement: CREATE TABLE course (cours
2025-06-24 00:43:21.637 UTC [22992] LOG:  statement: ALTER SYSTEM SET log_state
2025-06-24 00:44:29.489 UTC [1825] LOG:  checkpoint starting: time
2025-06-24 00:44:34.711 UTC [1825] LOG:  checkpoint complete: wrote 53 buffers
2025-06-24 00:44:55.863 UTC [22992] LOG:  statement: ALTER SYSTEM SET log_state
2025-06-24 00:45:01.315 UTC [1823] LOG:  received SIGHUP, reloading configurati
2025-06-24 00:45:01.316 UTC [1823] LOG:  parameter "autovacuum_max_workers" car
2025-06-24 00:45:01.316 UTC [1823] LOG:  parameter "log_statement" changed to "
2025-06-24 00:45:01.316 UTC [1823] LOG:  configuration file "/var/lib/pgsql/17/
2025-06-24 00:45:19.968 UTC [22992] LOG:  statement: INSERT INTO course (course
2025-06-24 00:45:28.648 UTC [22992] LOG:  statement: SELECT * FROM course;
2025-06-24 00:49:29.417 UTC [1825] LOG:  checkpoint starting: time
2025-06-24 00:49:29.524 UTC [1825] LOG:  checkpoint complete: wrote 2 buffers (
```

⚠️ Warning:

Using `log_statement = 'all'` provides **maximum visibility**, which is great for debugging, audits, and development environments. **But in production, be careful:**

- 🚨 Can **quickly bloat log files** in read-heavy applications
- 💥 May **impact performance** due to disk I/O overhead

🧠 Best Practice:

Use `log_statement = 'all'`:

- In **development**, for deep debugging
- In **test environments**, for auditing or query pattern analysis
- With **short-term time limits**, to avoid excessive log generation

🔚 Wrapping Up This Section

With these hands-on examples, you've now seen how PostgreSQL reacts to each `log_statement` level:

Setting Logs What? Ideal For `none` Nothing High-performance prod environments `ddl` Schema changes Change tracking & auditing `mod` DDL + data-modifying DML Data mutation monitoring `all` Every SQL statement Debugging, full traceability

🧮 Quick Comparison: What Does Each `log_statement` Level Capture?

When choosing the right `log_statement` level, it helps to see a **side-by-side comparison** of what each setting actually logs.

Here's a quick reference table:

`log_statement`	Logs DDL (`CREATE` , `ALTER` , `DROP`)	Logs DML (`INSERT` , `UPDATE` , `DELETE`)	Logs `SELECT`
`none`	❌ No	❌ No	❌ No
`ddl`	✅ Yes	❌ No	❌ No
`mod`	✅ Yes	✅ Yes	❌ No
`all`	✅ Yes	✅ Yes	✅ Yes

This table gives you a clear understanding of how much query activity each level captures — so you can choose one that matches your observability needs without compromising performance.

🔐 Where Should You Use `log_statement` ?

The ideal setting for `log_statement` depends on **your environment** and **your goals.** Here's a breakdown of common environments and the recommended setting for each:

Environment Recommended Setting Why? Development `all` Full visibility into all queries for debugging and learning **UAT / QA** `mod` Capture schema and data changes, skip noisy SELECTs **Production** `none` or `ddl` Prioritize performance, optionally log schema changes for traceability **Compliance / Auditing** `mod` Ensure all DDL and DML operations are tracked for audit purposes

By aligning the `log_statement` level with the purpose of your environment, you avoid unnecessary noise while still capturing the information that matters.

📑 Conclusion: Fine-Tuning PostgreSQL Logging for Visibility and Performance

The `log_statement` parameter is one of PostgreSQL's most **powerful logging tools**. It gives you full control over how much query activity you want to observe, track, or audit—without requiring any external plugins.

Whether you're debugging, auditing, or protecting performance, `log_statement` helps you balance:

- 🔎 **Query visibility** — know what's running and when
- 📊 **Auditability** — trace who changed what and why
- ⚙️ **Performance** — avoid log bloat in high-throughput environments

🎯 Key Takeaway:

There's no one-size-fits-all setting. Use `log_statement` **strategically**, adjusting it based on your current workload, environment, and operational needs.

📢 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!

Postgre MySQL Sql Oracle AWS

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

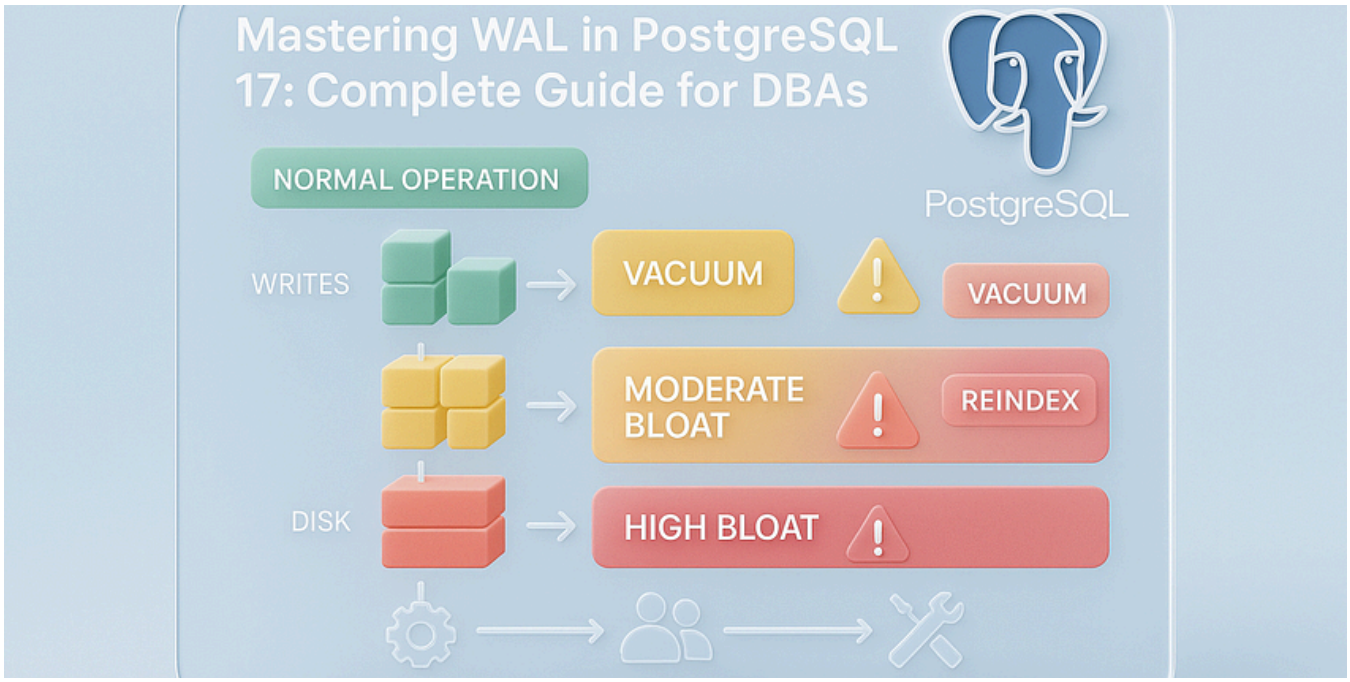
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

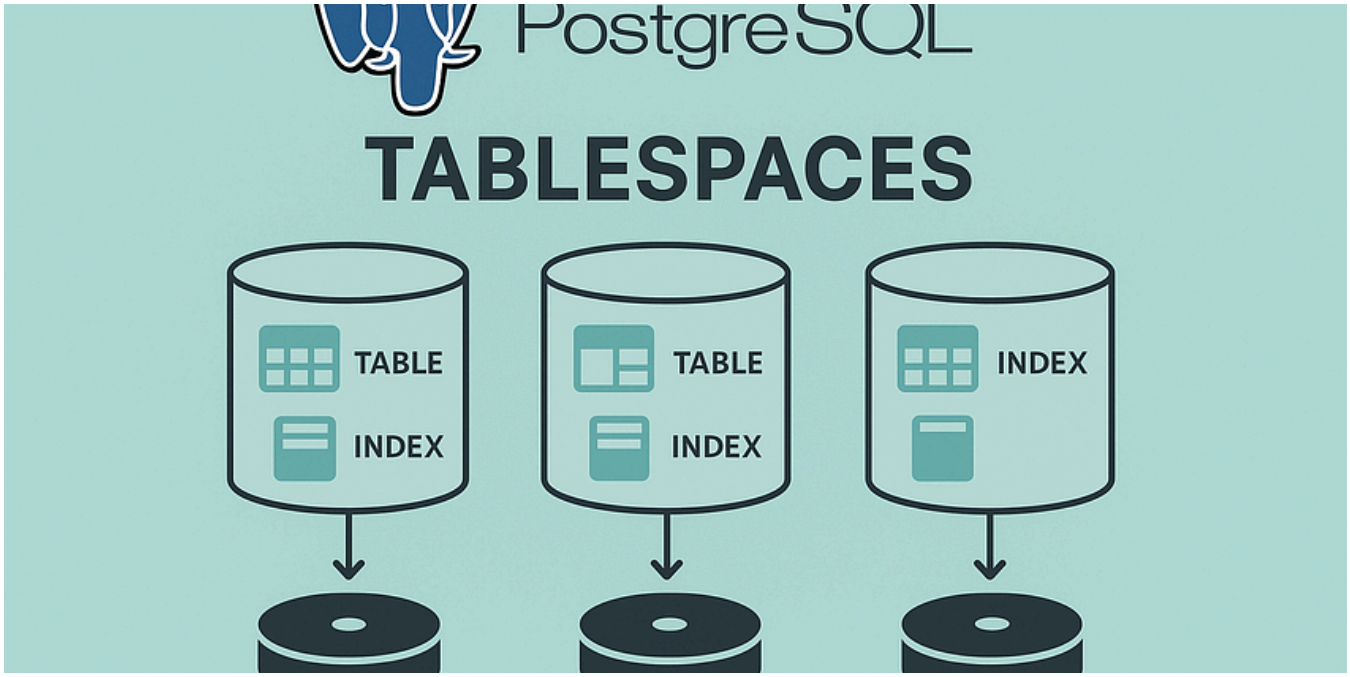


Jeyaram Ayyalusamy

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

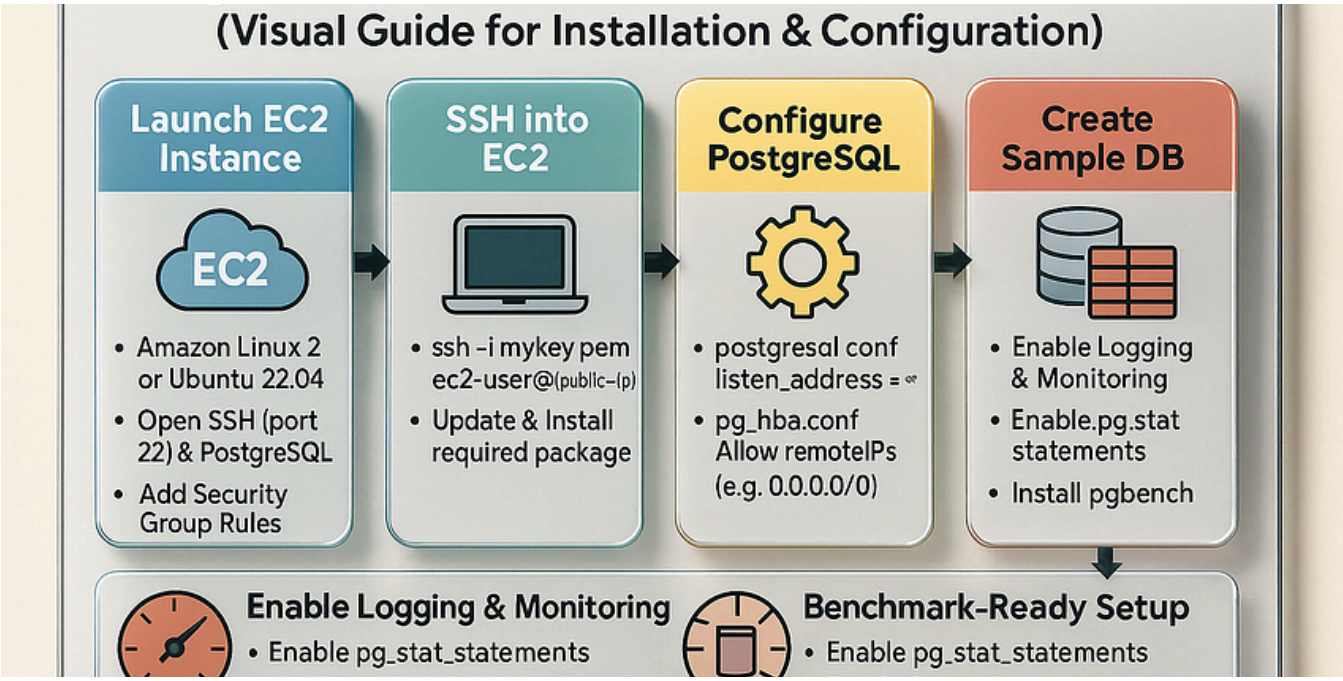


Jeyaram Ayyalusamy

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

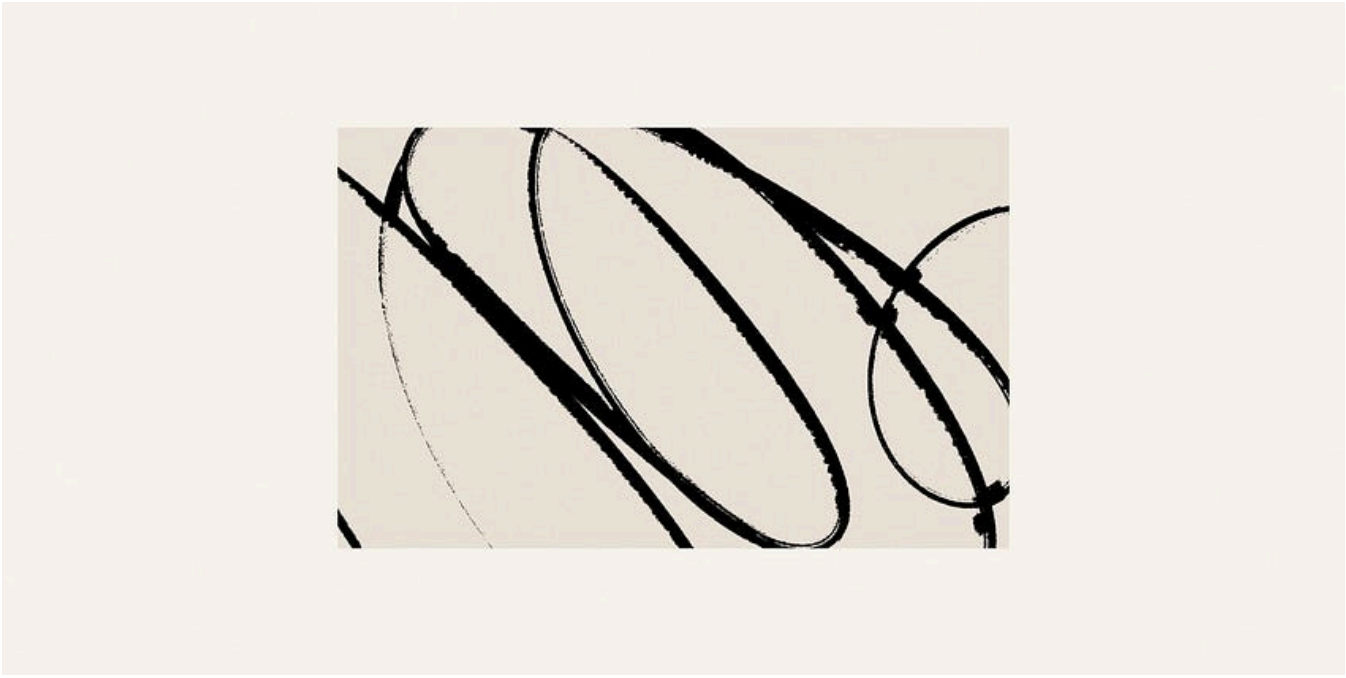
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

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

Full-Text Search in PostgreSQL: Better Than You Think

Why developers are abandoning Elasticsearch for PostgreSQL's built-in search—and how it handles 10M records without breaking a sweat

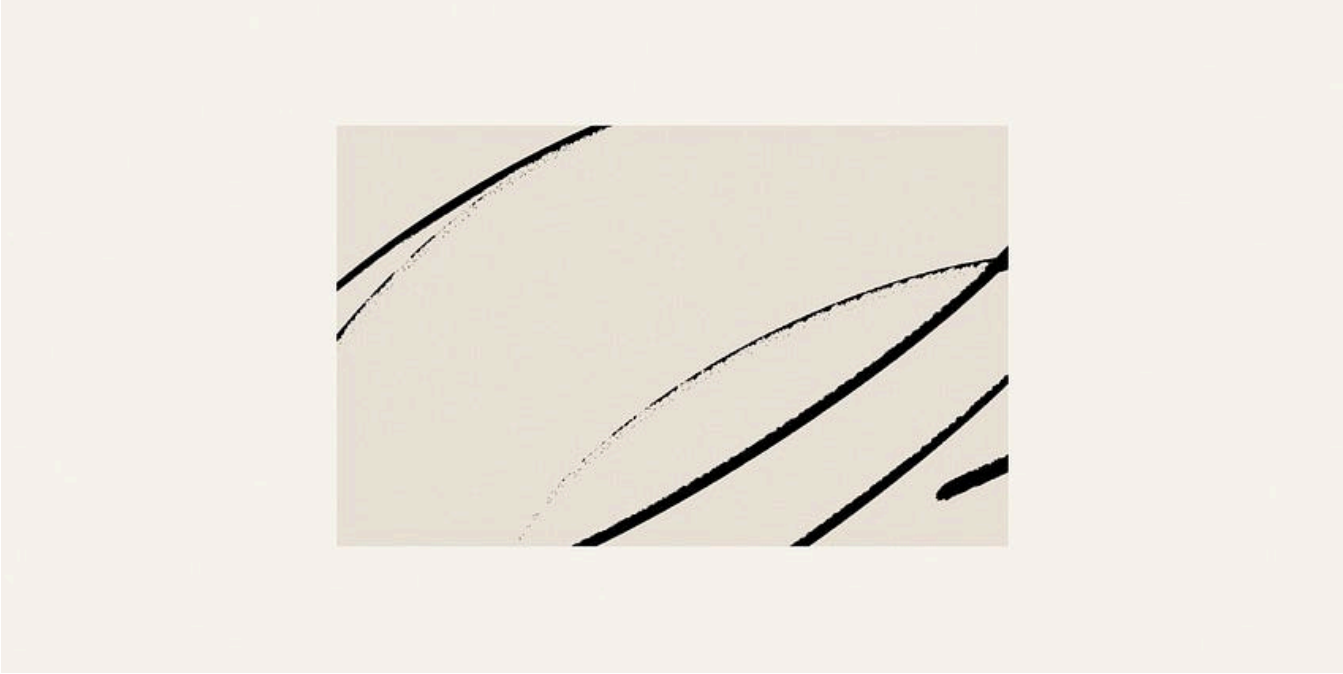
5d ago 2

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

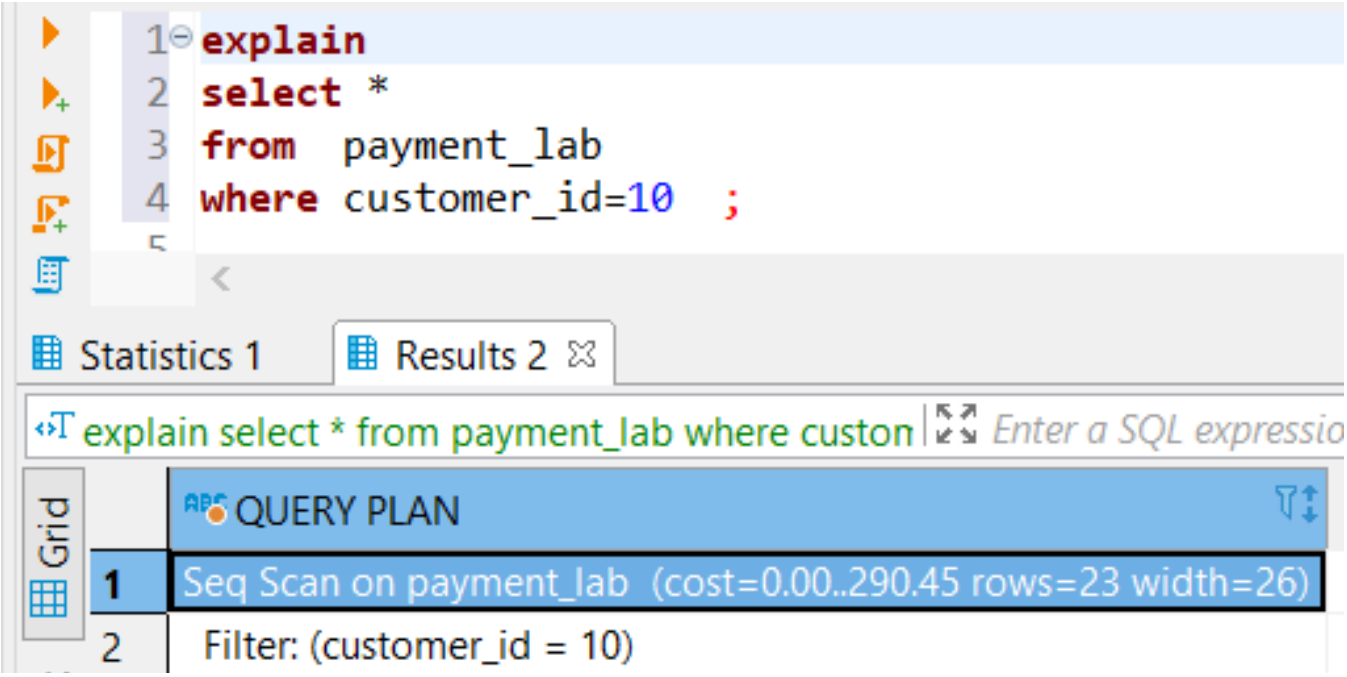


Gizem Şangür

Building a Scalable API with FastAPI and PostgreSQL: A 2025 Guide

Fast. Asynchronous. Pythonic.

Jul 17

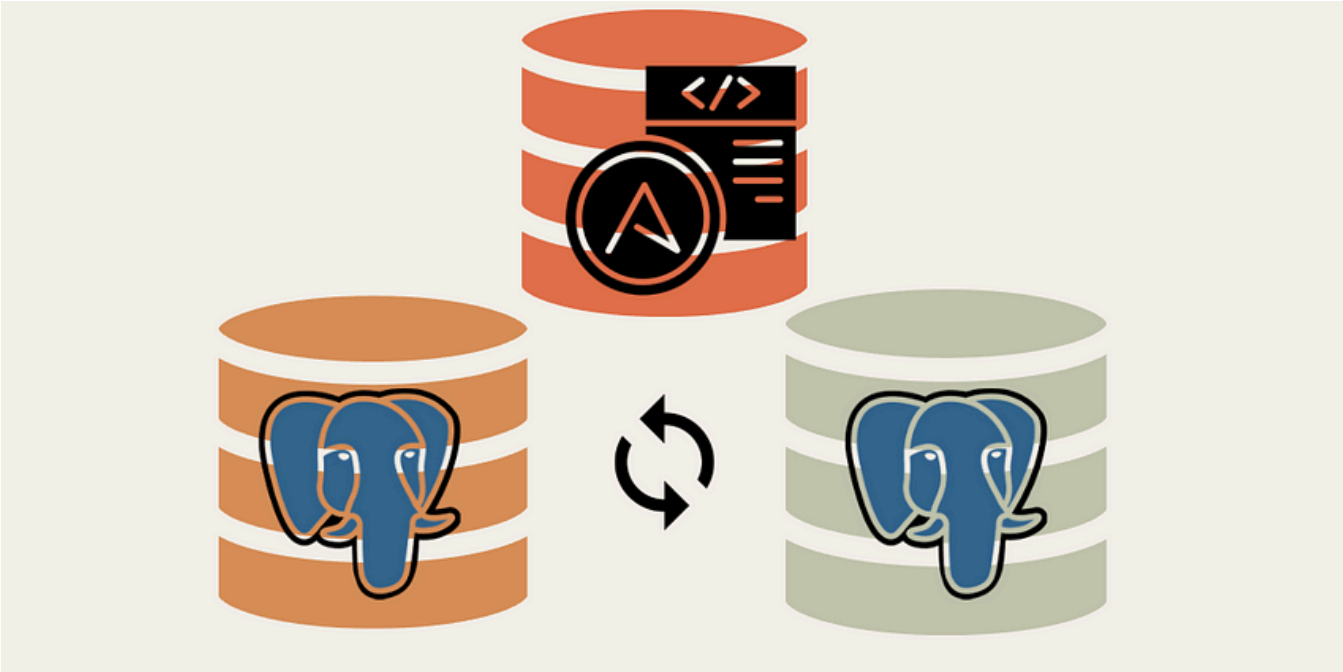


Muhammet Kurtoglu

Postgresql Query Performance Analysis

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

6d ago 10

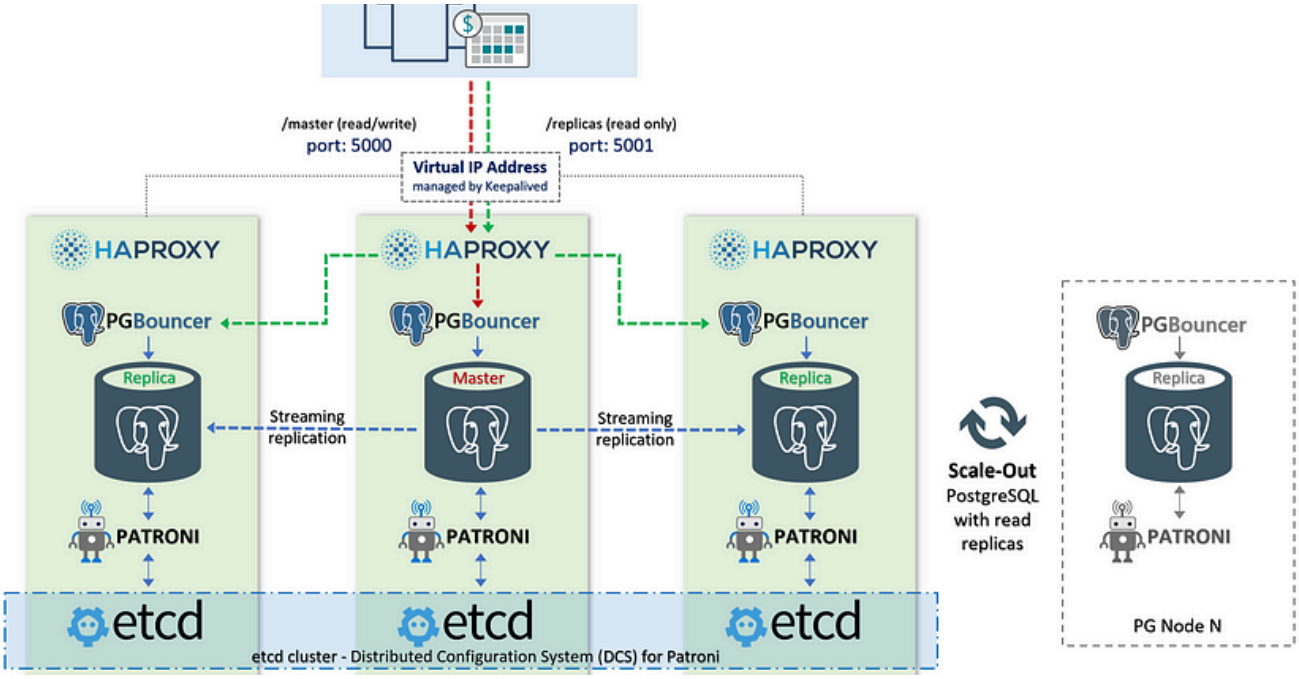


Oz

Automating PostgreSQL Streaming Replication Setup with Ansible

Setting up PostgreSQL streaming replication manually can be a tedious and error-prone task —involving installing PostgreSQL, configuring...

Jul 8 58



Kanat Akylson

How to Deploy a High-Availability PostgreSQL Cluster in 5 Minutes Using Ansible and Patroni

This tutorial shows how to spin up a production-grade HA PostgreSQL cluster in just 5 minutes using m2y ready-made GitHub repository with...

Jun 9 65 1

See more recommendations