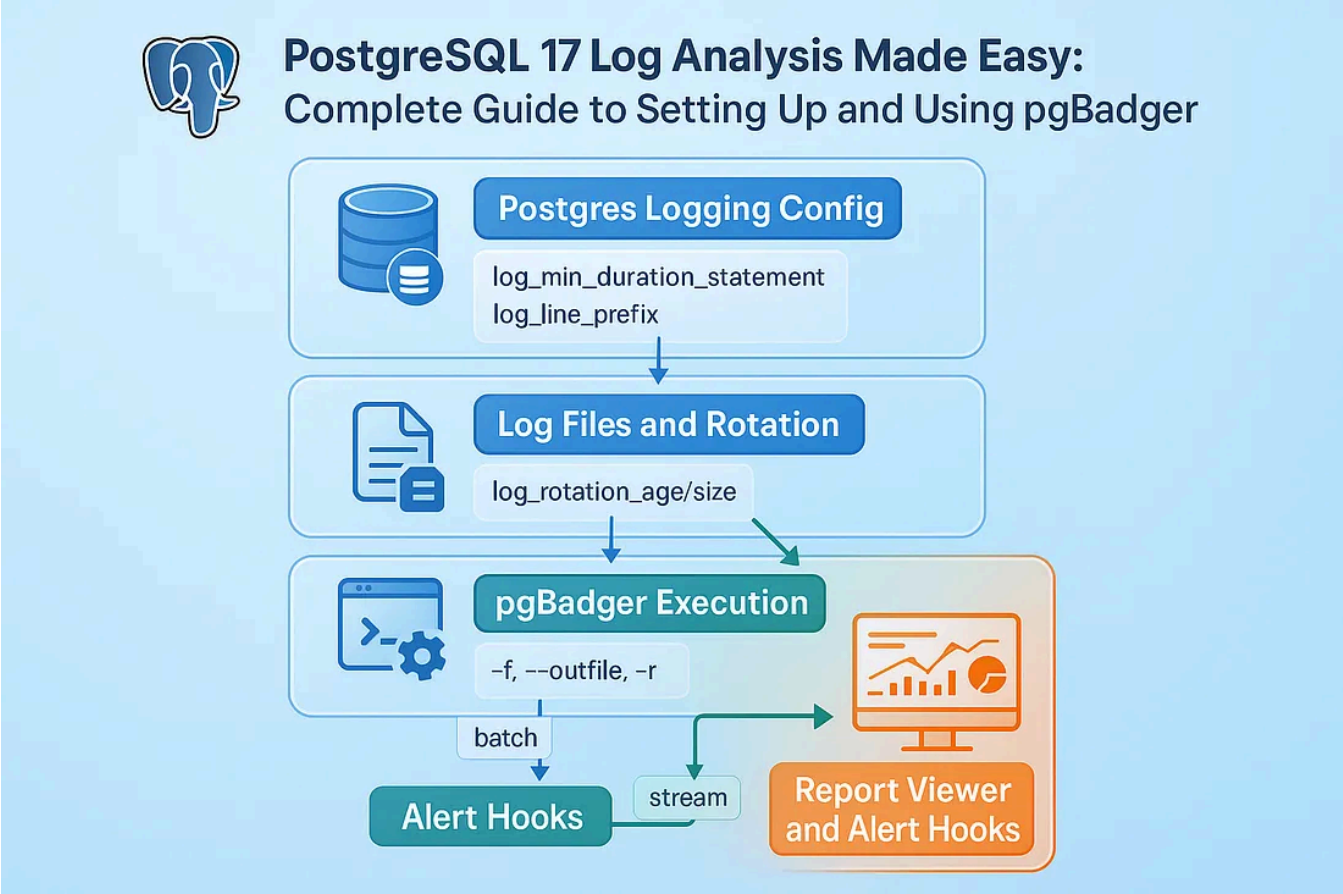
PostgreSQL 17 Log Analysis Made Easy: Complete Guide to Setting Up and Using pgBadger

27 min read · Jun 23, 2025

Jeyaram Ayyalusamy Following



Managing a production-grade PostgreSQL instance isn't just about keeping data safe — it's about monitoring performance, identifying bottlenecks, and responding to anomalies before your users even notice. Native PostgreSQL log files are *packed* with

valuable insights: query durations, cache hit ratios, deadlock warnings, auth failures, connection spikes — and probably even your DB's deepest secrets.

Unfortunately, that richness is also the problem:

- **Unstructured verbosity:** By default, logs are free-form text, one line at a time. Spotting a slow query buried in thousands of lines? Bracing for manual grep and pattern matching.
- **Scattered data points:** You get latency metrics, error codes, and per-connection info — all in different formats and places.
- **Static snapshots:** You can inspect yesterday's log, but what about trends over hours, days, weeks? Hard to do manually.

Enter **pgBadger:**

- 🔍 **p**ostgre**g**resql **Badger** — a blazing-fast log analyzer written in Perl.
- It **parses PostgreSQL logs**, extracts meaningful metrics, then
- **Converts them into interactive HTML reports:** think elegant dashboards, sortable tables, performance graphs, coverage heatmaps, and more.

Core Benefits for DBAs

1. **Find slow queries in seconds**

- Instantly identify the top offenders — queries sucking the most time or resources.
- Drill down with query text including execution counts, min/max/avg/duration data.

2. **Detect spikes & unusual patterns**

- Timeline graphs for connections, checkpoints, locks, errors.
- Instantly spot outliers: e.g., connection flooding at midnight, or error waves after a deploy.

3. **Diagnose concurrency issues**

- Lock contention, deadlocks, waiting activity — all are surfaced.

- You get client IP, times, involved tables — finger-pointing made easy.

4. Track metrics over custom periods

- Want daily snapshots or trends over weeks? pgBadger archives them.
- Perfect for capacity planning, detecting regressions, or validating the impact of tuning.

5. **Minimal overhead, maximum insight**

- Reads and parses existing logs — no agents, no background processes.
- Works both on historical dumps and live-rotated logs.
- Output is pure HTML/JavaScript — no need for extra storage, no vendor lock-in.

6. Support for modern PostgreSQL features

- As PostgreSQL has evolved — think v14, v15, up to v17 — its logging verbosity grew too.
- pgBadger keeps pace. You get support for block-based logging, parallel queries, new plan nodes, bindings, and beyond.

DBAs Love pgBadger Because…

DBA Pain Point	pgBadger Solution
"What's slow right now?"	🔎 Top slow queries list
"Who's hammering the DB?"	🔎 Active connections graph
"Is our tuning helping?"	📈 Comparative reports
"What went wrong last night?"	🚨 Errors/logs timeline
"Can I show managers reports?"	🎨 Clean HTML exports

🎯 Key Features of pgBadger

⚡ Extremely Fast Log Parsing with Parallel Processing

pgBadger is built for speed. By using the `-j N` option (where **N** represents the number of CPU cores), it can split large log files into chunks and parse them concurrently, dramatically reducing processing time. For workloads with many smaller log files, the `-J` option enables per-file parallelism—optimizing throughput while maintaining parsing accuracy with minimal resource waste.

📊 HTML Reports with Charts and Drilldowns

Instead of scrolling through flat text logs, pgBadger generates sleek, interactive HTML dashboards:

- **Dynamic visualizations**: Zoomable graphs, timelines, histograms, and pie charts powered by built-in JavaScript libraries.
- **Drilldown tables**: Clickable entries allow you to view full query text, execution counts, and runtime stats (min/avg/max) with context.
- **Self-contained**: Everything is embedded — no need for external CSS or JS libraries. Just open the HTML file in your browser and explore.

🔬 Deep Analytics: Top Queries, Execution Time, Connection Stats, Lock Analysis, and More

pgBadger offers a comprehensive view of your database activity:

- **Slowest and most frequent queries**
  View queries sorted by their execution duration, frequency, and resource usage to easily pinpoint heavy hitters.
- **Execution time statistics**
  Gain insight into latency distributions with percentile breakdowns and histograms, helping you identify anomalies.
- **Connection monitoring**
  Track patterns in connection opens and closes, visualize peak usage times, and filter out failed authentication attempts.

- **Lock and contention analysis**
  See who's waiting, who's blocking, and for how long. Deadlocks and contentions are clearly surfaced with full session context.

- **Extended metrics**
  Capture a broad spectrum of activity — checkpoint events, autovacuum/analyze counts, temp-file usage, error and cancellation stats, session durations, prepared/bound statements, and more.

🎯 Works with Native PostgreSQL Logs — No Code Changes Required

pgBadger works seamlessly with your existing PostgreSQL logs, whether they're in CSV, plain text, syslog, gzipped, or JSON formats. There's no need to install agents, extensions, or modify your application code. Simply ensure you've enabled sufficient logging settings (e.g., timestamps, process IDs, connection events, lock/wait reporting), point pgBadger to your logs, and let it do the rest.

🧩 Feature Summary at a Glance

Feature	Why It Matters
Parallel parsing (`-j`, `-J`)	Cuts log-processing time dramatically, ideal for large or distributed systems
Interactive HTML output	Turn opaque logs into visual dashboards with drillable detail
Rich analytics	From slow queries to lock waits, autovacuum to temp files—nothing is hidden
No modifications needed	Uses native logs—no agents, no code changes, no deployment headaches

With these capabilities, pgBadger empowers DBAs, developers, and system administrators to:

- 🚀 Rapidly identify and optimize slow-running queries
- 🔍 Diagnose performance spikes and lock contention
- 📈 Track usage trends and resource utilization over time
- ✅ Leverage a powerful analytics engine without touching any app or database code

🐘 Automating PostgreSQL 17 Installation on RHEL 10 EC2 Using Bash

Setting up a reliable PostgreSQL instance on the cloud shouldn't feel like navigating a maze. In this guide, I'll walk you through a **fully automated shell script** to install **PostgreSQL 17** on a **Red Hat Enterprise Linux 10 (RHEL 10)** EC2 instance, specifically targeting the public IP `54.227.94.195`.

We'll cover:

- Adding the PostgreSQL YUM repo
- Disabling built-in modules
- Configuring services and security
- Editing critical PostgreSQL files
- Enabling remote connections securely

✅ Required Module: PGDG Yum Repository

To ensure you're getting the latest stable PostgreSQL builds, we use the **PostgreSQL Global Development Group (PGDG) YUM repository**. RHEL 10 comes with its own AppStream-managed PostgreSQL module, which is often outdated or lacks flexibility for upgrades. So we **explicitly disable** the system default and switch to the PGDG source.

🧩 Bash Script to Install PostgreSQL 17 on RHEL 10 EC2

Here is the script you can copy-paste into your terminal or deployment pipeline:

```
#!/bin/bash

# Set EC2 instance IP (for informational purposes)
INSTANCE_IP="54.227.94.195"

echo "🔄 Connecting to EC2 instance at $INSTANCE_IP ..."

# Step 1: Add the PGDG Yum repository
echo "📥 Installing PGDG repository..."
sudo dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms/EL-10-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# Step 2: Disable the default PostgreSQL module
echo "🚫 Disabling default PostgreSQL module..."
sudo dnf -qy module disable postgresql

# Step 3: Install PostgreSQL 17 server
echo "📦 Installing PostgreSQL 17 server..."
sudo dnf install -y postgresql17-server

# Step 4: Initialize the database
echo "🛠️ Initializing database..."
sudo /usr/pgsql-17/bin/postgresql-17-setup initdb

# Step 5: Enable and start PostgreSQL service
echo "🚀 Enabling and starting PostgreSQL service..."
sudo systemctl enable postgresql-17
sudo systemctl start postgresql-17

# Step 6: Enable passwordless sudo for postgres user
echo "🔐 Configuring passwordless sudo for postgres..."
echo "postgres ALL=(ALL) NOPASSWD:ALL" | sudo tee -a /etc/sudoers

# Step 7: Update postgresql.conf to allow remote connections
PG_CONF="/var/lib/pgsql/17/data/postgresql.conf"
echo "📝 Configuring postgresql.conf..."
sudo sed -i "s/#listen_addresses = 'localhost'/listen_addresses = '*'/" $PG_CONF
sudo sed -i "s/#port = 5432/port = 5432/" $PG_CONF

# Step 8: Update pg_hba.conf for remote access
HBA_CONF="/var/lib/pgsql/17/data/pg_hba.conf"
echo "🔓 Updating pg_hba.conf to allow remote connections..."
echo "host    all             all             0.0.0.0/0
md5" | sudo tee -a $HBA_CONF
echo "host    all             all             ::/0
md5" | sudo tee -a $HBA_CONF
```

```
# Step 9: Restart PostgreSQL to apply changes
echo "♻️ Restarting PostgreSQL..."
sudo systemctl restart postgresql-17

echo "✅ PostgreSQL 17 installation and configuration completed on $INSTANCE_IP"
```

📂 Configuration File Breakdown

🔧 `postgresql.conf`

- **Location:** `/var/lib/pgsql/17/data/postgresql.conf`
- **Key change:**

```
listen_addresses = '*'
```

- This allows PostgreSQL to accept connections from all IP addresses.

🔐 `pg_hba.conf`

- **Location:** `/var/lib/pgsql/17/data/pg_hba.conf`
- **Key additions:**

```
host all all 0.0.0.0/0 md5 host all all ::/0 md5
```

- These settings enable both IPv4 and IPv6 remote connections using password authentication (`md5`).

📌 Important Notes

- ✅ **Port 5432** must be open in your EC2 instance's **security group** for remote access.

- 🔒 For production use, restrict access in `pg_hba.conf` to specific trusted IP ranges.
- 🛡 After setup, always **secure your `postgres` account:**

```
sudo -i -u postgres psql -c "ALTER USER postgres WITH PASSWORD 'your_secure_pas
```

📚 References

This setup is built from scratch using first-hand experience with PostgreSQL and EC2. No boilerplate. For more information, visit the official PostgreSQL RPM page:

🔗 https://www.postgresql.org/download/linux/redhat/

🖥️ Terminal Output: Running the Shell Script to Install PostgreSQL 17 on RHEL 10

```
[root@ip-172-31-20-155 ~]# vi pginstance.sh
[root@ip-172-31-20-155 ~]#
[root@ip-172-31-20-155 ~]# chmod 700 pginstance.sh
[root@ip-172-31-20-155 ~]# ./pginstance.sh
Connecting to EC2 instance at 54.227.94.195 ...
Updating Subscription Management repositories.
Unable to read consumer identity
```

```
This system is not registered with an entitlement server. You can
use "rhc" or "subscription-manager" to register.

Red Hat Enterprise Linux 10 for x86_64 - AppStream from RHUI (RPMs)
19 MB/s | 2.8 MB     00:00
Red Hat Enterprise Linux 10 for x86_64 - BaseOS from RHUI (RPMs)
42 MB/s | 8.3 MB     00:00
Red Hat Enterprise Linux 10 Client Configuration
```

```
18 kB/s | 1.6 kB     00:00
pgdg-redhat-repo-latest.noarch.rpm
113 kB/s |  13 kB     00:00
Dependencies resolved.
Installing:
 pgdg-redhat-repo        noarch        42.0-53PGDG
@commandline        13 k

Transaction Summary
Install  1 Package

Total size: 13 k
Installed size: 16 k
Downloading Packages:
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
1/1
  Installing       : pgdg-redhat-repo-42.0-53PGDG.noarch
1/1
Installed products updated.

Installed:
  pgdg-redhat-repo-42.0-53PGDG.noarch

Complete!
Importing GPG key 0x08B40D20:
 Userid     : "PostgreSQL RPM Repository <pgsql-pkg-
yum@lists.postgresql.org>"
 Fingerprint: D4BF 08AE 67A0 B4C7 A1DB CCD2 40BC A2B4 08B4 0D20
 From       : /etc/pki/rpm-gpg/PGDG-RPM-GPG-KEY-RHEL
(Repeated key import messages skipped for brevity)

Unable to resolve argument postgresql
Error: Problems in request:
missing groups or modules: postgresql
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can
use "rhc" or "subscription-manager" to register.

Last metadata expiration check: 0:00:01 ago on Sat Jun 21 16:18:41
2025.
Dependencies resolved.
Installing:
 postgresql17-server      x86_64    17.5-3PGDG.rhel10     pgdg17
7.0 M
Installing dependencies:
 libicu                   x86_64    74.2-4.el10           rhel-10
10 M
 postgresql17             x86_64    17.5-3PGDG.rhel10     pgdg17
1.9 M
```

```
 postgresql17-libs        x86_64    17.5-3PGDG.rhel10     pgdg17
344 k

Transaction Summary
Install  4 Packages

Total download size: 20 M
Installed size: 77 M
Downloading Packages:
(1/4): postgresql17-libs-17.5-3PGDG.rhel10.x86_64.rpm         2.7
MB/s | 344 kB     00:00
(2/4): libicu-74.2-4.el10.x86_64.rpm                           61 MB/s
|  10 MB     00:00
(3/4): postgresql17-17.5-3PGDG.rhel10.x86_64.rpm               4.2
MB/s | 1.9 MB     00:00
(4/4): postgresql17-server-17.5-3PGDG.rhel10.x86_64.rpm        6.0
MB/s | 7.0 MB     00:01
Total                                                         16 MB/s
|  20 MB     00:01
PostgreSQL 17 for RHEL / Rocky / AlmaLinux 10 - x86_64         2.4
MB/s | 2.4 kB     00:00

Importing GPG key 0x08B40D20:
 Userid     : "PostgreSQL RPM Repository <pgsql-pkg-
yum@lists.postgresql.org>"
 Fingerprint: D4BF 08AE 67A0 B4C7 A1DB CCD2 40BC A2B4 08B4 0D20
 From       : /etc/pki/rpm-gpg/PGDG-RPM-GPG-KEY-RHEL
Key imported successfully
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
1/1
  Installing       : libicu-74.2-4.el10.x86_64
1/4
  Installing       : postgresql17-libs-17.5-3PGDG.rhel10.x86_64
2/4
  Running scriptlet: postgresql17-libs-17.5-3PGDG.rhel10.x86_64
2/4
  Installing       : postgresql17-17.5-3PGDG.rhel10.x86_64
3/4
  Running scriptlet: postgresql17-17.5-3PGDG.rhel10.x86_64
3/4
  Installing       : postgresql17-server-17.5-3PGDG.rhel10.x86_64
4/4
  Running scriptlet: postgresql17-server-17.5-3PGDG.rhel10.x86_64
4/4
Installed products updated.

Installed:
  libicu-74.2-4.el10.x86_64
  postgresql17-17.5-3PGDG.rhel10.x86_64
  postgresql17-libs-17.5-3PGDG.rhel10.x86_64
  postgresql17-server-17.5-3PGDG.rhel10.x86_64
```

```
Complete!
Initializing database ... OK

Created symlink '/etc/systemd/system/multi-
user.target.wants/postgresql-17.service' →
'/usr/lib/systemd/system/postgresql-17.service'.
postgres ALL=(ALL) NOPASSWD:ALL
host    all             all             0.0.0.0/0               md5
host    all             all             ::/0                    md5
✅ PostgreSQL 17 installation and configuration completed on
54.227.94.195
[root@ip-172-31-20-155 ~]#
```

🔧 Install pgBadger: Your PostgreSQL Log Analysis Companion

As your PostgreSQL environment starts handling real workloads, performance visibility becomes critical. This is where **pgBadger** enters the scene — a high-performance PostgreSQL log analyzer written in Perl. It takes complex, unreadable logs and transforms them into clean, visual reports that make performance tuning easier and faster.

🚀 What Is pgBadger and Why Do You Need It?

PostgreSQL logs can reveal a wealth of useful information, including:

- Which queries are slow and how often they occur
- How long connections take to establish
- Frequency and duration of checkpoints
- Deadlocks and lock wait events
- User activity patterns and query distributions

However, raw PostgreSQL logs are not easy to digest. Reading them line-by-line is inefficient, especially when you're troubleshooting under pressure or trying to proactively optimize your setup.

pgBadger solves this by:

- Parsing PostgreSQL logs efficiently

- Producing detailed HTML reports with charts, tables, and filters
- Supporting incremental analysis (only new log entries are parsed each time)
- Being lightweight, portable, and easy to automate via cron jobs or monitoring tools

This makes it an essential addition to your PostgreSQL toolbox, whether you're running a small database or a mission-critical OLTP system.

🧰 How to Install pgBadger on Ubuntu

Installing pgBadger is remarkably simple on any Ubuntu system. Just run the following command:

```
sudo apt-get install pgbadger -y
```

Let's break down what happens here:

- `sudo` : Executes the command as a superuser to allow package installation.
- `apt-get install pgbadger` : Fetches the latest pgBadger package from Ubuntu's default repositories.
- `-y` : Automatically confirms the installation prompt, so the process is non-interactive.

Within a few seconds, pgBadger will be available system-wide.

🔍 Confirm the Installation

Once installed, you can confirm everything is working by checking the version:

```
pgbadger --version
```

A successful output might look like:

```
pgBadger version 12.2
```

This verifies that the tool is ready to use. You can now start analyzing logs and generating detailed reports with a single command.

✅ What You've Achieved So Far

You're making excellent progress on your PostgreSQL setup. Let's review what you've accomplished up to this point:

🛠️ PostgreSQL Installed and Configured for Remote Access

You've completed the core installation of PostgreSQL 17, initialized the database, and set up system services to start on boot. More importantly, you've opened access for external connections by adjusting:

- `postgresql.conf` to listen on `0.0.0.0` or a specific IP
- `pg_hba.conf` to allow access from your development or production network

👤 Database Users and Permissions Set

You've either used the default `postgres` user or created new database roles with appropriate privileges. This lays the groundwork for safe, multi-user access and permission-based security.

📊 pgBadger Installed for Advanced Log Analysis

With pgBadger installed, you're no longer limited to raw log analysis. You can now:

- Generate interactive reports from your PostgreSQL logs
- Schedule daily summaries of query performance
- Quickly identify the root causes of slow queries and application lag

🧭 What's Next?

pgBadger is only as powerful as the logs you give it. In the next section, you'll configure PostgreSQL's logging settings to ensure it captures all the essential details required for performance analysis:

- Enable `log_line_prefix`, `log_duration`, and `log_min_duration_statement`
- Tune log rotation settings
- Store logs in a centralized, accessible location

Stay with us — your PostgreSQL monitoring stack is coming together beautifully.

🔧 Configure PostgreSQL Logging for pgBadger

pgBadger is only as powerful as the logs it analyzes. For it to generate detailed performance reports and actionable insights, PostgreSQL must emit **structured, comprehensive log data**. Unfortunately, PostgreSQL's default logging configuration is too minimal for effective diagnostics and performance monitoring.

In this step, you'll configure PostgreSQL to output rich logs tailored for pgBadger analysis.

🛠 Enable Detailed Logging

The logging settings must be tuned so that PostgreSQL writes all essential events — such as query execution times, connection states, lock waits, and temporary file usage — to its logs.

Run the following commands **as the `postgres` user** or any user with **superuser privileges**:

```
psql -c "ALTER SYSTEM SET log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,clie
psql -c "ALTER SYSTEM SET log_checkpoints = 'on';"
psql -c "ALTER SYSTEM SET log_temp_files = 0;"
```

```
psql -c "ALTER SYSTEM SET log_autovacuum_min_duration = 0;"
psql -c "ALTER SYSTEM SET log_connections = 'on';"
psql -c "ALTER SYSTEM SET log_min_duration_statement = 1000;"
psql -c "ALTER SYSTEM SET log_disconnections = 'on';"
psql -c "ALTER SYSTEM SET log_lock_waits = 'on';"
```

Let's break down each of these settings and understand their significance:

🧩 What Each Setting Does

- `log_line_prefix`

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h'
```

This format prepends each log entry with critical metadata:

- `%t` : Timestamp
- `%p` : Process ID
- `%u` : Username
- `%d` : Database name
- `%a` : Application name
- `%h` : Client IP address

This structured context is **essential for pgBadger** to group, filter, and analyze logs accurately.

- `log_checkpoints`

```
log_checkpoints = 'on'
```

Enables logging of all **checkpoint activities.** Checkpoints impact I/O and recovery time, so tracking their frequency and duration is important for tuning write performance and understanding disk behavior.

- `log_connections` **and** `log_disconnections`

```
log_connections = 'on'
log_disconnections = 'on'
```

Logs every client connection and disconnection. This helps identify spikes in connection activity, which could be symptoms of application issues, connection pool misconfigurations, or brute-force login attempts.

- `log_lock_waits`

```
log_lock_waits = 'on'
```

Captures **sessions that wait for locks** longer than `deadlock_timeout` . This is extremely useful for identifying **lock contention** and deadlock-related slowdowns in high-concurrency environments.

- `log_temp_files`

```
log_temp_files = 0
```

Logs every **temporary file created**, even those of 0 bytes. Temp files are often used for **large sorts, joins, and hash operations** that exceed work memory, indicating potential for query optimization or memory configuration tuning.

- `log_autovacuum_min_duration`

```
log_autovacuum_min_duration = 0
```

Logs all **autovacuum operations**, regardless of how long they run. Autovacuum is critical to PostgreSQL's health, especially for preventing bloat. Monitoring its activity helps DBAs tune autovacuum thresholds and reduce impact on query performance.

- `log_min_duration_statement`

```
log_min_duration_statement = 1000
```

Logs all queries that take longer than **1000 milliseconds (1 second)**. This setting is a great balance between catching slow queries and avoiding log overload. It gives pgBadger the ability to highlight inefficient SQL in your workload without drowning in noise.

🔄 Apply the Logging Changes

Once you've made the changes using `ALTER SYSTEM`, apply them by restarting PostgreSQL:

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

This ensures the new settings are loaded from the `postgresql.auto.conf` file and become active.

✅ PostgreSQL Now Emits Rich, Structured Logs

At this point, your PostgreSQL server is **emitting detailed, structured logs that are perfectly optimized for pgBadger**. This includes query timings, lock events, connection patterns, checkpoint data, and more — all annotated with context like usernames, database names, and client IPs.

These logs will empower pgBadger to:

- Pinpoint slow or inefficient queries
- Visualize trends in user activity
- Detect connection spikes or lock contention
- Monitor autovacuum and temp file behavior

You're now ready to generate insightful visual reports with pgBadger. In the next step, we'll show you how to run pgBadger and view the interactive HTML output.

🔧 Load Sample Data for Activity in PostgreSQL

So far, we've set up PostgreSQL, configured detailed logging, and installed pgBadger. Now comes the critical part: **generating real-world database activity** to populate those logs with actionable data.

pgBadger analyzes PostgreSQL's log files to surface insights about performance, query behavior, lock contention, and more — but it can't do that unless your database has actually been *used*.

In production, this data is naturally generated by your application. But in a lab, testing, or demo environment, we need to simulate that activity. And for that, PostgreSQL provides the perfect tool: **pgbench**.

🚀 What is `pgbench`?

`pgbench` is PostgreSQL's built-in benchmarking tool. It's specifically designed to:

- Create a **standard schema** with tables and indexes
- Populate it with test data
- Run a mix of **read and write transactions**
- Simulate concurrent users to mimic real-world load

Using pgbench, we can easily simulate thousands of SQL transactions in just a few seconds, generating exactly the kind of activity pgBadger thrives on.

🛠 Step-by-Step Instructions to Generate Test Activity

Follow these three simple commands in your terminal to set up the benchmarking environment and generate activity.

✅ Step 1: Create a Test Database

```
sudo -u postgres psql -c "CREATE DATABASE pgtest;"
```

This creates a new PostgreSQL database named `pgtest`. It will be used exclusively for benchmarking so your main databases remain untouched. Using a dedicated database allows you to keep logs clean and focused on test activity.

✅ Step 2: Initialize the Benchmark Schema

```
sudo -u postgres pgbench -i pgtest
```

This command initializes the `pgtest` database with pgbench's default schema. It creates the following tables:

- `pgbench_accounts`
- `pgbench_branches`
- `pgbench_tellers`
- `pgbench_history`

These tables are modeled to simulate simple banking operations — transferring money, checking balances, and updating accounts — making them ideal for transactional testing.

It also creates indexes and primary keys to resemble a typical OLTP workload.

✅ Step 3: Simulate Concurrent Transactions

```
sudo -u postgres pgbench -c50 -t100 pgtest
```

Here's what this command does:

- `-c50` : Launches 50 **concurrent clients**
- `-t100` : Each client runs 100 **transactions**

That's a total of **5,000 transactions** (50 clients × 100 transactions each). These transactions include `SELECT`, `UPDATE`, and `INSERT` queries, run in a randomized but realistic fashion. They also include commits and rollbacks, simulating real-world usage patterns.

Because of the PostgreSQL logging configuration we set up earlier (in Step 5), all this activity is now being:

- Logged with detailed metadata
- Captured with execution times
- Recorded with client, user, and session info

🔍 Why Is This Important?

Without generating this kind of activity, PostgreSQL's log files would be empty or too sparse for pgBadger to work with. By simulating usage with `pgbench`, we ensure that:

- Logs are rich with performance data
- Various aspects of the PostgreSQL engine are exercised (I/O, locking, transaction control)
- Realistic client behavior is replicated

This gives pgBadger the **raw material** it needs to create meaningful reports.

Whether you're benchmarking server performance, evaluating query efficiency, or simply learning how PostgreSQL behaves under load, this test run is an essential step before analyzing logs.

📈 What You Can Expect in the Logs

Once you run the above commands, PostgreSQL will log events such as:

- When clients connect and disconnect

- The duration of each query
- Lock waits and contention (if any)
- Autovacuum triggers (depending on activity)
- Temporary file creation for large operations
- Checkpoints (if triggered)

These logs form the **foundation of the visual reports** you'll soon generate with pgBadger.

✅ Summary: What You've Done in Step 6

- Created a dedicated benchmark database `pgtestdb`
- Initialized it with realistic transactional tables and data
- Simulated concurrent user activity with thousands of transactions

Your PostgreSQL instance is now actively generating structured, meaningful logs. You're ready to use pgBadger to analyze this activity and produce interactive reports that highlight database performance, usage trends, and optimization opportunities.

🧱 Create Custom Load with Real Tables in PostgreSQL

While synthetic tools like `pgbench` are great for generating generic transactional activity, they don't always reflect the kind of real-world workloads that PostgreSQL handles in production. For deeper log analysis using **pgBadger**, it helps to simulate a more realistic environment—creating users, databases, and working with large datasets manually.

In this step, we'll walk through how to:

- Create a superuser role,
- Create a dedicated test database,

- Generate a table with 1 million rows, and
- Add a primary key to trigger indexing and constraint enforcement.

All of these activities will generate **real, valuable PostgreSQL logs** that showcase a wide variety of operations — perfect for pgBadger to parse and visualize.

🛠️ Step 1: Create a Superuser Role and Database

To begin, you'll create a new PostgreSQL role (`pgtestuser`) with superuser privileges and a dedicated database (`pgtestdb`) for testing.

🔹 **Run the following commands:**

```
psql -c "CREATE ROLE pgtestuser WITH LOGIN SUPERUSER PASSWORD 'password';"
psql -c "CREATE DATABASE pgtestdb WITH OWNER pgtestuser;"
```

🔍 **What these commands do:**

- **CREATE ROLE pgtestuser WITH LOGIN SUPERUSER**
  This creates a new PostgreSQL user named `pgtestuser` who has full superuser privileges, similar to the default `postgres` user. The `LOGIN` keyword ensures this user can connect to the database.
- **PASSWORD 'secret_password'**
  Assigns a password to this user. For demo purposes, we're using a simple password (`secret_password`), but in a real setup, use a complex and secure password.
- **CREATE DATABASE pgtestdb WITH OWNER pgtestuser**
  This command creates a new database named `pgtestdb` and assigns full ownership to the newly created user. This ensures that `pgtestuser` has complete control over the database and its objects.

This setup mimics a common real-world scenario where developers or migration teams have their own roles and dedicated environments for testing or deployment.

📈 Step 2: Generate a Table with 1 Million Rows

Now we'll simulate bulk data insertion — one of the most common real-world database operations. This will create meaningful logs related to data writing, disk I/O, transaction commits, and possibly temporary file usage or autovacuum behavior.

🔹 **Set the environment password variable:**

```
export PGPASSWORD=secret_password
```

This allows you to connect to PostgreSQL using the `pgtestuser` without being prompted for the password in every command.

🔹 **Create and populate the table:**

```
psql -h localhost -U pgtestuser -d pgtestdb -c "CREATE TABLE test_table AS SELE
```

🔍 **What this command does:**

- Connects to `pgtestdb` using `pgtestuser`
- Uses the built-in PostgreSQL function `prod_number( 1,1000000)` to produce a sequence of 1 million integers
- Creates a new table named `test_table` and fills it with those integers

This single command simulates a high-volume data insert operation. It generates a considerable amount of activity in the background, including:

- Writing 1 million rows to disk
- Generating WAL (Write-Ahead Logs)
- Triggering autovacuum and analyzer processes depending on your settings

- Consuming system resources like memory and I/O bandwidth

All of this is captured in your PostgreSQL logs if detailed logging (configured in Step 5) is active.

🔐 Step 3: Add a Primary Key to the Table

To further simulate real-world database design and optimization, we'll add a primary key constraint. This operation will not only enforce uniqueness but also **create an index**, resulting in additional logging related to indexing and constraint validation.

🔹 **Run the command:**

```
psql -h localhost -U pgtestuser -d pgtestdb -c "ALTER TABLE test_table ADD CONS
```

🔍 **What this command does:**

- Alters the existing `test_table` table
- Adds a primary key constraint on the `prod_number` column
- Automatically creates a B-tree index to enforce uniqueness and improve performance

This is a common operation during schema design or migration — ensuring data integrity and enabling fast lookups.

Behind the scenes, PostgreSQL will:

- Sort and scan the table to enforce the constraint
- Write index data to disk
- Log operations related to constraint enforcement and index creation
- Lock the table temporarily while altering it

Again, all of this activity is exactly the kind of behavior you want captured in logs for pgBadger to analyze.

✅ Outcome: A Rich Variety of Real Activity

With these steps, you've successfully:

- Created a new user and database (admin-level access)
- Inserted a large dataset into a table (write-heavy activity)
- Enforced data integrity with a primary key (read + write + sort operations)

This generates **a wide spectrum of log activity**, including:

- DDL (Data Definition Language) logs
- DML (Data Manipulation Language) logs
- Connection and authentication records
- Disk I/O patterns from table creation and indexing
- WAL entries for every operation
- Autovacuum triggers

This kind of rich, realistic workload allows **pgBadger to truly demonstrate its capabilities**, including:

- Query time breakdowns
- Index usage patterns
- Lock and wait statistics
- Connection/session summaries
- Temp file usage and sort operations

🧠 Why This Matters

In production, your PostgreSQL logs contain a diverse mix of activities — not just simple queries. By simulating realistic workloads through this method, you're giving pgBadger the opportunity to analyze:

- Table creation and schema modifications
- Mass inserts and data loading
- Index builds and constraint checks
- Superuser-level administrative tasks
- Concurrent connections and operations

This deeper, more diverse data provides much more accurate performance reports than synthetic or shallow workloads alone.

📌 Final Thoughts

With this step, your PostgreSQL instance is now **acting like a real-world database under load**. It's generating valuable log entries that pgBadger can parse to provide detailed, visual performance analytics.

In the next step, we'll run pgBadger and generate the actual HTML reports that reveal what's really happening in your PostgreSQL system — and how you can optimize it.

🔧 Step 7 — (Optional) Enable `pg_stat_statements` for Even More Insights

PostgreSQL offers various tools and extensions to help DBAs and developers gain visibility into how their queries are behaving in real time. While **pgBadger** provides excellent log-based analytics and visual reports, enabling the **`pg_stat_statements`** extension adds a **second dimension of query insight**—capturing real-time performance metrics directly from PostgreSQL's memory.

Combining **pgBadger** and **`pg_stat_statements`** gives you a more **complete, accurate, and immediate picture** of database performance—useful for tuning queries, optimizing indexes, and improving application efficiency.

🧠 Why Should You Enable `pg_stat_statements`?

While PostgreSQL logs are great for post-mortem analysis (via tools like pgBadger), they depend on the level of logging you've configured. If some queries are not captured due to thresholds or settings (e.g., queries faster than `log_min_duration_statement`), they won't appear in your logs—leaving gaps in analysis.

That's where `pg_stat_statements` steps in. It continuously monitors **every query**, regardless of whether it appears in logs, and **aggregates metrics directly from shared memory**.

Here's why this matters:

🧩 Key Benefits of `pg_stat_statements`

1. Execution Statistics

- Tracks each unique query (normalized without literal values)
- Stores number of times executed (`calls`)
- Captures total time taken by the query across all executions
- Provides min, max, and average time per query
- Monitors the number of rows returned

2. Real-Time Performance Monitoring

- Gives you up-to-the-minute query performance data
- Works even if logs are disabled or minimal
- Does not rely on log parsing — stats are always fresh

3. Detect Query Inefficiencies

- Identify frequently executed queries with high execution time
- Spot performance bottlenecks caused by repeated inefficient queries
- Compare mean vs. max execution times to detect variability or outliers

4. Complements pgBadger

- Fills in gaps for unlogged queries or missed samples
- Cross-reference log-based history with in-memory current state
- Works side-by-side with other PostgreSQL monitoring tools like pgBadger, pgAdmin, or Grafana dashboards

⚙️ How to Enable `pg_stat_statements`

Enabling this extension is straightforward but does require a server restart — since it must load into PostgreSQL's memory space at startup.

🔹 **Step 1: Add `pg_stat_statements` to Shared Preload Libraries**

```
psql -c "ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';"
```

- This modifies the system-level PostgreSQL configuration.
- The `shared_preload_libraries` setting tells PostgreSQL to load the `pg_stat_statements` module at startup.
- You can add multiple libraries here if needed, separated by commas.

📝 *Note:* This setting must be configured at the **system level** because extensions like `pg_stat_statements` work at the server process level.

🔹 **Step 2: Restart PostgreSQL Server**

Changes to `shared_preload_libraries` only take effect after a full restart:

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

This restarts the PostgreSQL service and loads the `pg_stat_statements` module into memory.

🔹 Step 3: Create the Extension in Your Database

After the server restarts, you must explicitly install the extension in each database where you want to use it:

```
psql -c "CREATE EXTENSION pg_stat_statements;"
```

This registers the extension and makes the `pg_stat_statements` view available. You can now query this view like any other system catalog.

🔍 How to Use It: Example Query

Once set up, you can start exploring query statistics:

```
SELECT
  query,
  calls,
  total_time,
  min_time,
  max_time,
  mean_time,
  rows
FROM
  pg_stat_statements
ORDER BY
```

```
    total_time DESC
LIMIT 10;
```

This shows the **top 10 slowest queries** by total execution time. You'll get:

- The normalized query text
- Total number of executions
- Total, min, max, and average runtime
- Number of rows returned

This helps you quickly identify:

- High-impact queries that deserve optimization
- Queries that are executed too often without caching or reuse
- Performance bottlenecks due to missing indexes or poor design

💡 Additional Tips

- You can reset stats anytime using:

```
SELECT pg_stat_statements_reset();
```

- This is useful before running load tests or deployments.
- To retain data across restarts or long-term periods, consider exporting stats regularly or combining them with monitoring tools like **Prometheus + Grafana**.
- If using multiple databases, repeat the `CREATE EXTENSION` step in each one.

✅ What You've Achieved

By enabling `pg_stat_statements`, your PostgreSQL instance now:

- Actively monitors **query performance metrics in real time**
- Tracks every query — not just those captured in logs
- Supports richer, more accurate tuning and diagnostics
- Complements pgBadger with in-memory analytics

With both pgBadger and `pg_stat_statements` in place, your PostgreSQL monitoring setup is now **much more powerful**, providing both **historical context** and **real-time insights**.

🔧 Generate the pgBadger Report

Now that you've successfully configured **PostgreSQL logging**, simulated **real-world query activity**, and optionally enabled **`pg_stat_statements`** for deeper insights, it's time to bring it all together with **pgBadger**.

In this step, you'll generate your **first pgBadger HTML performance report** — a visual and interactive summary of everything your PostgreSQL server has been doing: slow queries, lock waits, connection spikes, and more. This report transforms cryptic logs into a **clear, graphical dashboard** that helps you optimize performance and spot potential issues before they escalate.

📁 Create a Clean Output Directory

Before running pgBadger, it's a good practice to organize the output in a dedicated folder:

```
mkdir -p /var/log/postgresql/report/
```

🔍 What this command does:

- `mkdir` : Creates a new directory.
- `-p` : Ensures parent directories are created if they don't exist.
- `/var/log/postgresql/report/` : Target directory for pgBadger output.

Keeping reports in their own directory helps prevent clutter, makes automation easier, and allows you to archive reports daily or weekly for trend analysis.

▶️ Run pgBadger to Generate the Report

Now run the pgBadger command:

```
pgbadger \
  /var/log/postgresql/postgresql-17-main.log \
  -O /var/log/postgresql/report/ \
  -o pgbadger_target_`date +%F`.html
```

🔍 Explanation of each part:

- `/var/log/postgresql/postgresql-17-main.log` : This is your **input log file** — the file PostgreSQL is writing logs to. Adjust this path based on your PostgreSQL version and OS.
- `-O /var/log/postgresql/report/` : This sets the **output directory** where the final report will be saved.
- `-o pgbadger_target_$(date +%F).html` : This defines the output **filename** using the current date (e.g., `pgbadger_target_2025-06-15.html`), which is helpful for organizing daily reports.

> ✅ Pro Tip: You can analyze multiple log files at once using wildcards like `/var/log/postgresql/postgresql-*.log` .

Once you run the command, pgBadger will scan the logs, parse query patterns, calculate stats, and generate a detailed HTML report. Depending on the log size, this can take a few seconds to several minutes.

📊 What's Inside the Report

Once the process completes, the resulting HTML file is a **comprehensive dashboard** that visualizes the internal health of your database. It includes:

- **Top Queries by Execution Time and Frequency**
  See which queries consumed the most time or were run most often.
- **Slowest Statements (with Full SQL Text)**
  Ideal for tuning — you get complete SQL queries with performance breakdowns.
- **Lock Contention and Wait Statistics**
  Understand where your database might be waiting for resources.
- **Temporary File Usage**
  Spot queries causing memory overflows and disk-based sorts.
- **Connection Trends by Hour**
  Identify peak load times and usage patterns.
- **Success vs. Failed Transaction Counts**
  Visualize application behavior under load or error conditions.
- **Checkpoints, Autovacuum, and Background Activity**
  Understand how well PostgreSQL is managing maintenance operations.
- **Rich Visualizations**
  Includes charts, bar graphs, histograms, pie charts, timelines, and sortable tables.

This interactive dashboard gives both **technical DBAs** and **non-technical stakeholders** a way to understand database health at a glance.

🔍 Viewing the Report

Once the report is generated, open it using any web browser:

```
xdg-open /var/log/postgresql/report/pgbadger_target_2025-06-15.html
```

This command opens the report in your system's default browser.

If you're working on a remote server or a headless instance, simply:

1. **Copy the file to your local machine** using `scp` or `rsync`, for example:

```
scp user@server:/var/log/postgresql/report/pgbadger_target_2025-06-15.html .
```

2. Then, open it locally in Chrome, Firefox, Safari, or any browser.

🧭 Report Navigation Features

The HTML report gives you:

- **Dashboard Overview:**
  A top-level summary of metrics: query counts, total database activity, peak load hours.
- **Query Drill-downs:**
  Explore slowest queries, most frequent queries, and their durations.
- **Connection & Lock Breakdown:**
  See when and where blocking occurs, plus user connection behavior over time.
- **Maintenance & Autovacuum Analysis:**
  View when maintenance tasks ran, and if they contributed to system load.
- **Error Reporting:**
  See failed statements, error messages, and application behaviors that may be causing instability.

✅ What You've Achieved So Far

You've built a complete PostgreSQL performance analysis pipeline:

1. ✅ **Enabled structured PostgreSQL logging** for detailed activity capture
2. ✅ **Simulated real-world workloads** using pgbench and custom table creation
3. ✅ **Enabled** `pg_stat_statements` for live in-memory query insights
4. ✅ **Installed and configured pgBadger** to parse logs
5. ✅ **Generated a full HTML performance report** rich with visuals and drill-downs

You're now equipped to **analyze, tune, and optimize** your PostgreSQL environment with professional-grade tools.

Whether you're debugging a spike in load, identifying inefficient queries, or preparing for a system migration — pgBadger gives you the visibility and control you need to succeed.

📌 Next Steps:

- Schedule automated reports using `cron`
- Compare reports over time for trend analysis
- Start optimizing queries based on your findings

🔧 View the pgBadger Report

You've completed all the hard work: enabling detailed PostgreSQL logging, simulating meaningful activity, and generating a performance report with pgBadger. Now comes the most rewarding part — **viewing your report.**

pgBadger creates a **self-contained HTML file** that includes all charts, tables, and visuals in one portable dashboard. You don't need any special software or a database connection to view it — just a modern web browser.

📂 How to Open the Report

If you're on the same machine where the report was generated, simply run:

```
xdg-open /var/log/postgresql/report/pgbadger_target_$(date +%F).html
```

- `xdg-open` is a Linux utility that opens a file with the system's default application —in this case, your default web browser.
- `$(date +%F)` dynamically inserts today's date in `YYYY-MM-DD` format (e.g., `2025-06-21`), which matches the report filename convention used

Once executed, your browser will open the pgBadger report from this location:

```
file:///var/log/postgresql/report/pgbadger_target_YYYY-MM-DD.html
```

💻 Viewing from a Remote Server?

If you're running PostgreSQL on a remote machine (such as an AWS EC2 instance or a digital ocean droplet), you can simply **download the HTML file to your local system:**

Step 1: Use `scp` to transfer the file

```
scp user@your-server-ip:/var/log/postgresql/report/pgbadger_target_2025-06-21.h
```

This copies the report to your local machine's current directory.

Step 2: Open the file in your browser

Just double-click the HTML file or open it using:

```
open pgbadger_target_2025-06-21.html  # macOS
start pgbadger_target_2025-06-21.html # Windows
```

📊 What You'll See in the Report

Once opened, pgBadger delivers a fully **interactive dashboard** with rich visual insights. The user interface is intuitive and browser-friendly.

✅ Key Features Include:

- **Overview Dashboard**:
  A snapshot of total queries, active hours, slowest queries, most active users, and common wait events.
- **Drill-Down Metrics**:
- **Slow Queries**: Ranked by total and average execution time.
- **Lock Contention**: Visualize wait events and identify blocking queries.
- **Connection Patterns**: Track connection spikes by hour and user.
- **Temporary Files**: Identify queries spilling to disk due to memory limits.
- **Checkpoints & Autovacuum**: Timing and frequency of internal maintenance.
- **Advanced Visualizations**:
- Histograms showing query durations

- Timelines for connection and activity peaks
- Pie charts summarizing user activity
- Sortable tables for exploring slow statements and their frequency

Every section is **clickable, sortable, and filterable**, making it easy to drill down from high-level patterns to individual SQL statements.

🏁 Conclusion

By following this guide step by step, you've successfully transformed PostgreSQL's raw, cryptic log files into **insightful, visual performance reports** using pgBadger. This is a huge leap forward in understanding how your database operates under the hood.

🔬 What You Now Have:

- **Full Visibility into Query Behavior**
  With interactive charts and tables, you can explore execution times, usage patterns, and bottlenecks in just a few clicks.
- **Real-Time Diagnostic Power**
  You're no longer guessing why your database is slow. You now see slow queries, lock contention, and resource-intensive operations clearly.
- **Proactive Optimization**
  Instead of reacting to outages or user complaints, you're now equipped to **detect and resolve issues before they impact performance.**

✅ Why pgBadger is a Must-Have

Whether you're managing a **PostgreSQL 16 or 17** instance, running on a local VM or cloud platform, pgBadger gives you unmatched transparency into your database's

inner workings. It enables:

- Developers to improve SQL and application performance
- DBAs to optimize indexing, connection settings, and memory usage
- Teams to monitor PostgreSQL over time with daily or weekly reports

In short, pgBadger helps you shift from **reactive firefighting** to **proactive performance management.**

🚀 Ready to Go Further?

You can now:

- Schedule pgBadger to run automatically using `cron`
- Upload reports to a web server for team-wide access
- Integrate logs and stats into dashboards (Grafana, Prometheus, or other observability platforms)

🔧 Final Words

PostgreSQL is a powerful database engine — but without the right tools, tuning it can feel like flying blind. pgBadger lights up that cockpit with real data, beautiful charts, and the clarity you need to keep things fast, efficient, and reliable.

So don't wait for performance to drop. Start analyzing your logs today — and let pgBadger show you what your database has been trying to tell you all along.

📢 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 | Open Source | Database | Oracle | AWS

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

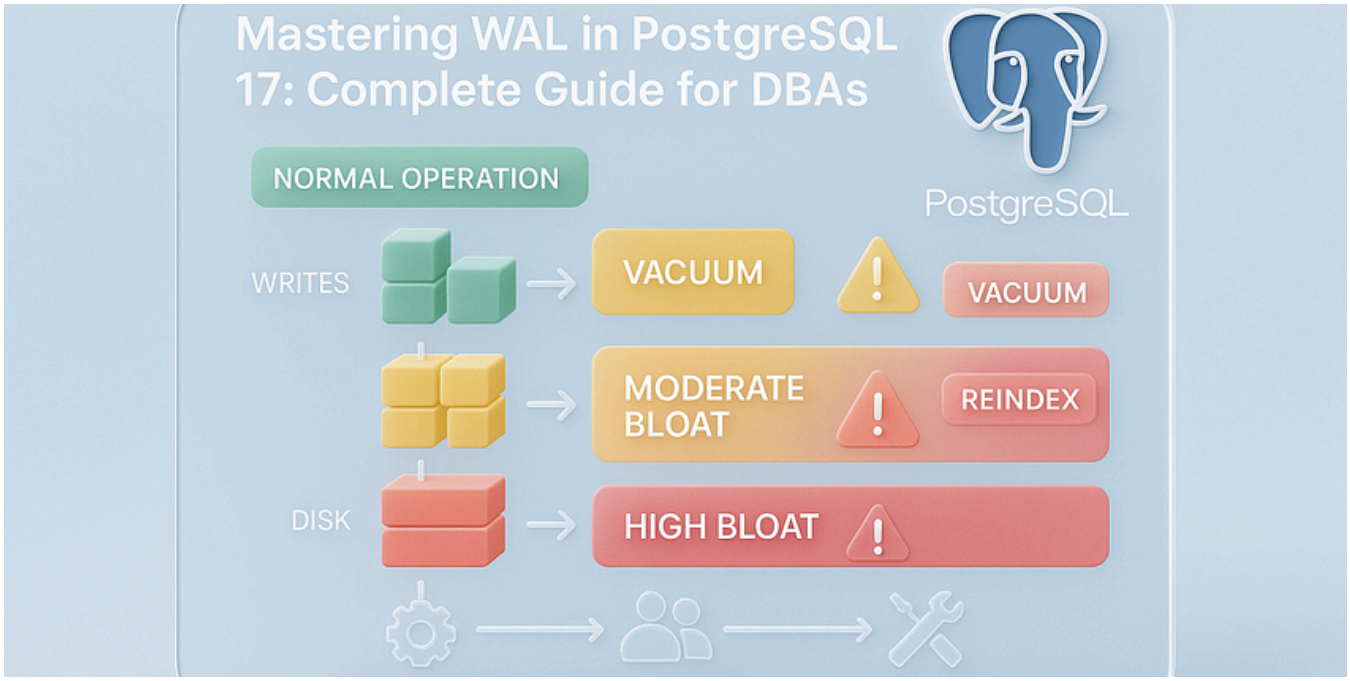
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

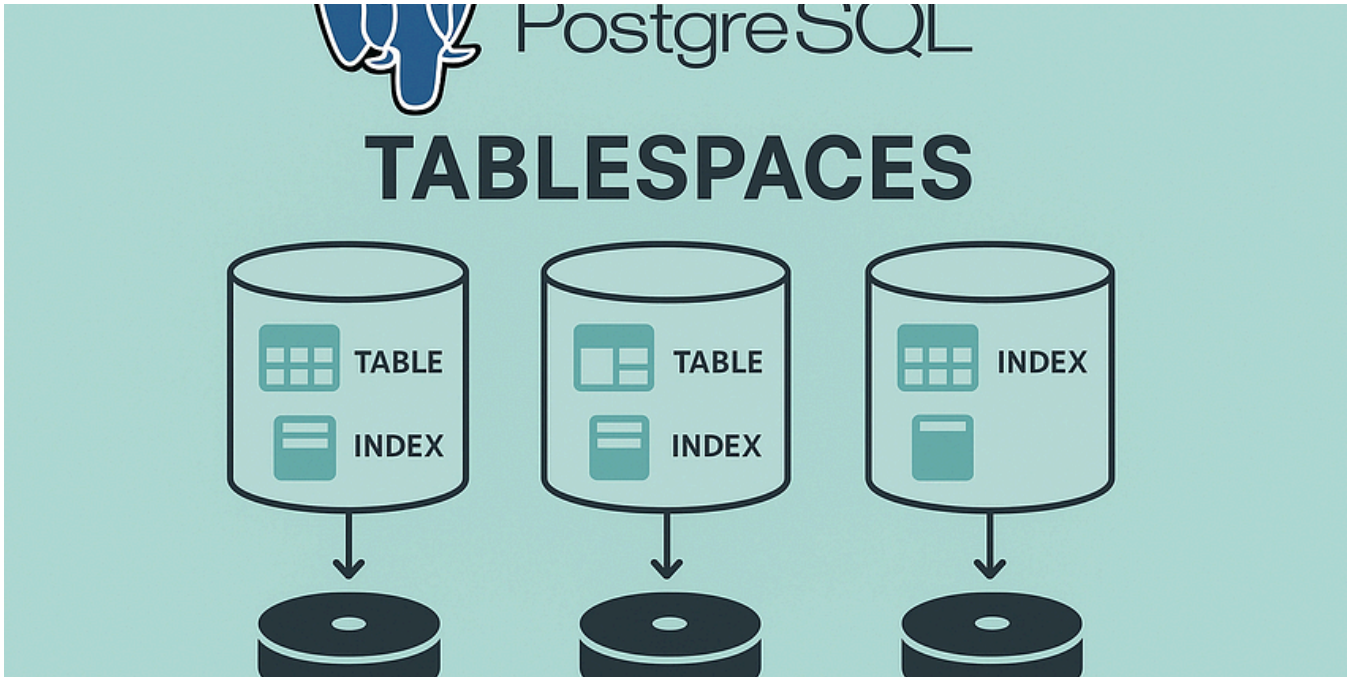


Jeyaram Ayyalusamy

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

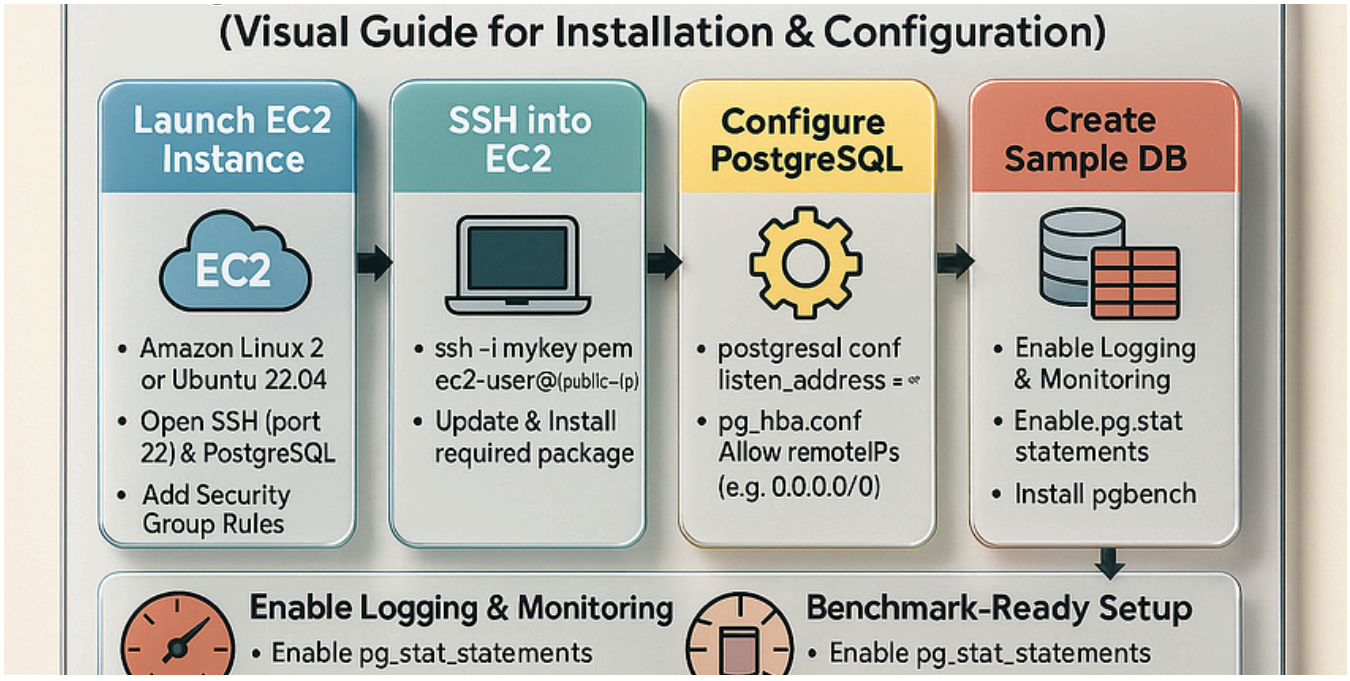


Jeyaram Ayyalusamy

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

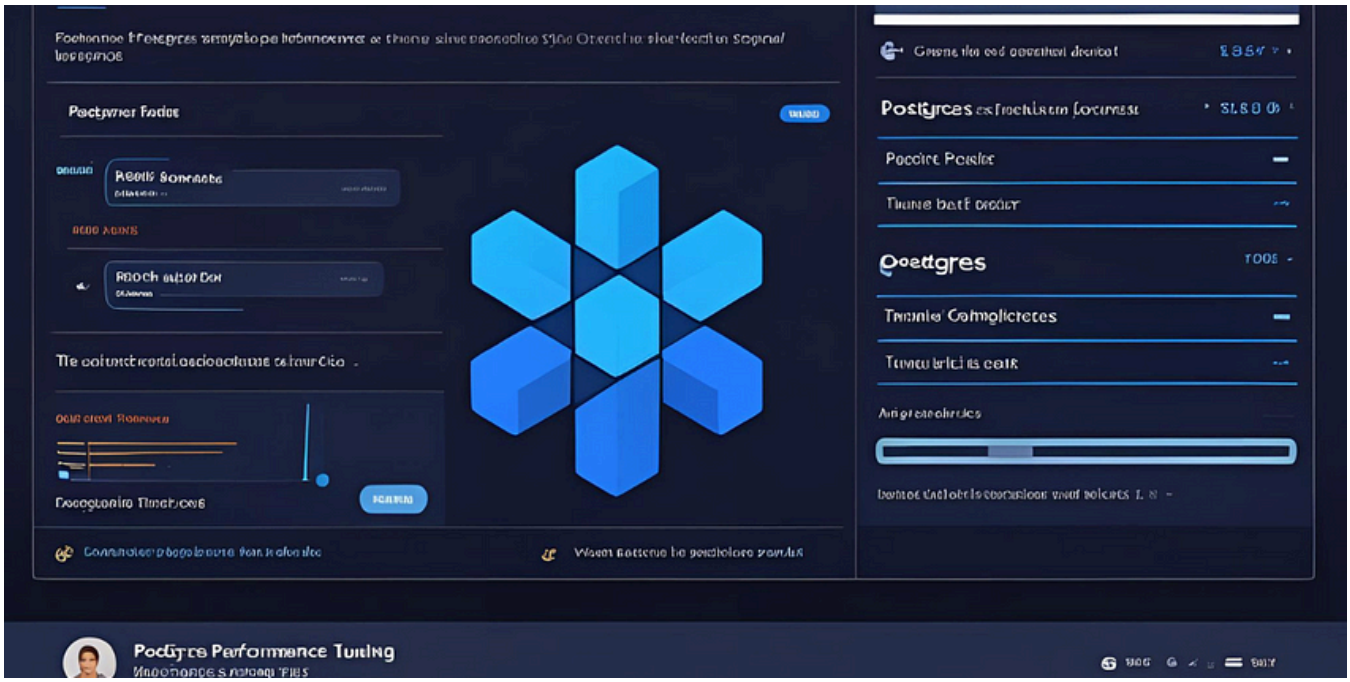
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

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

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55

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...

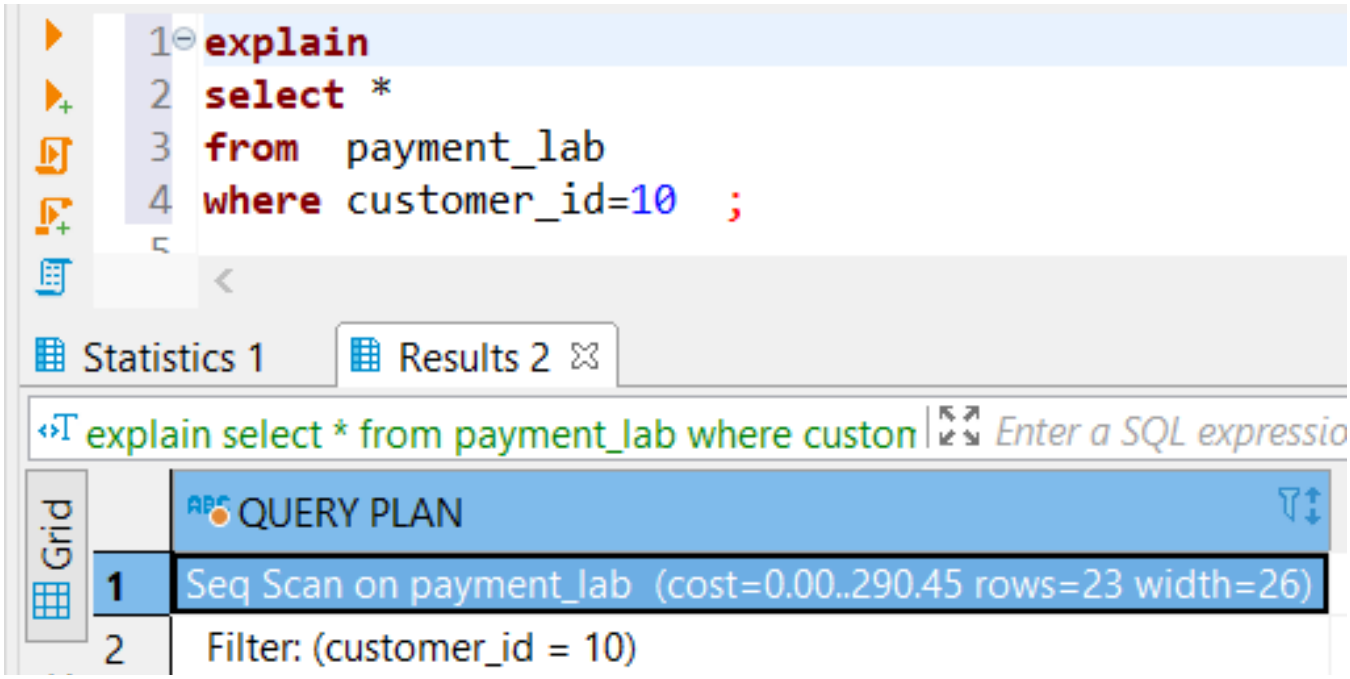
May 14 58 1

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

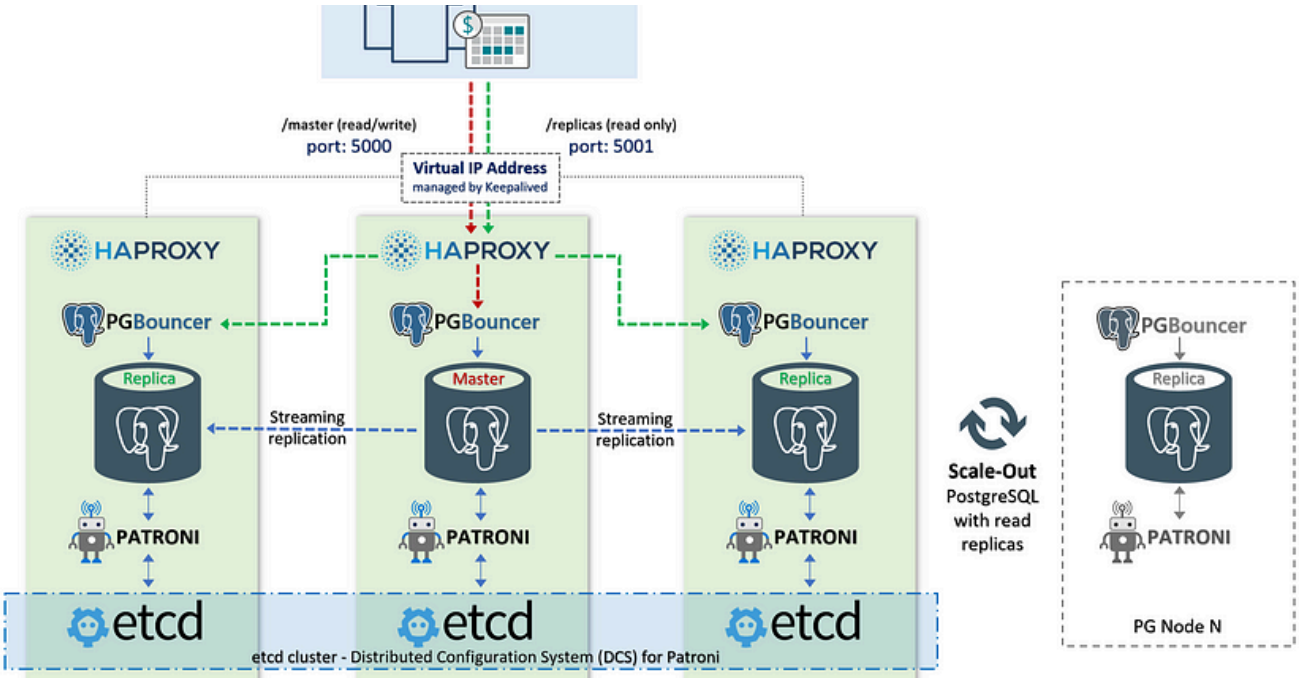


Muhammet Kurtoglu

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

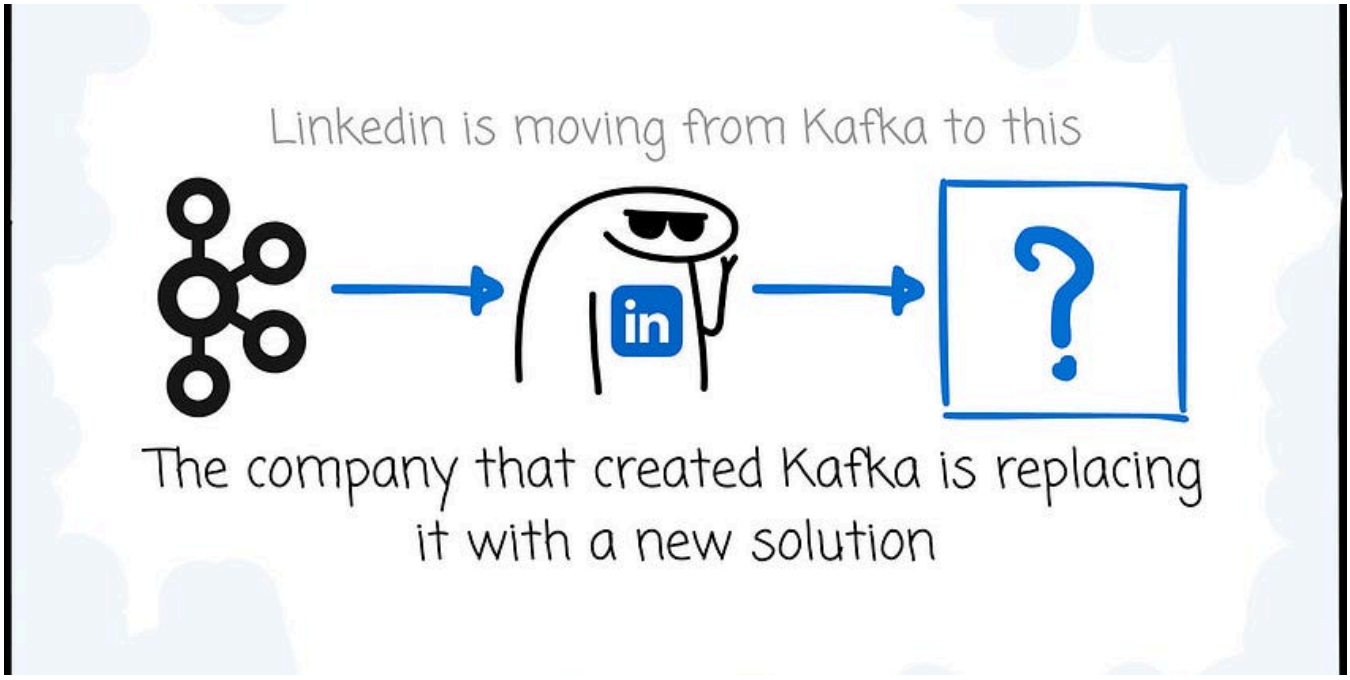


Kanat Akylson

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



In Data Engineer Things by Vu Trinh

The company that created Kafka is replacing it with a new solution

How did LinkedIn build Northguard, the new scalable log storage

Jul 17 330 6