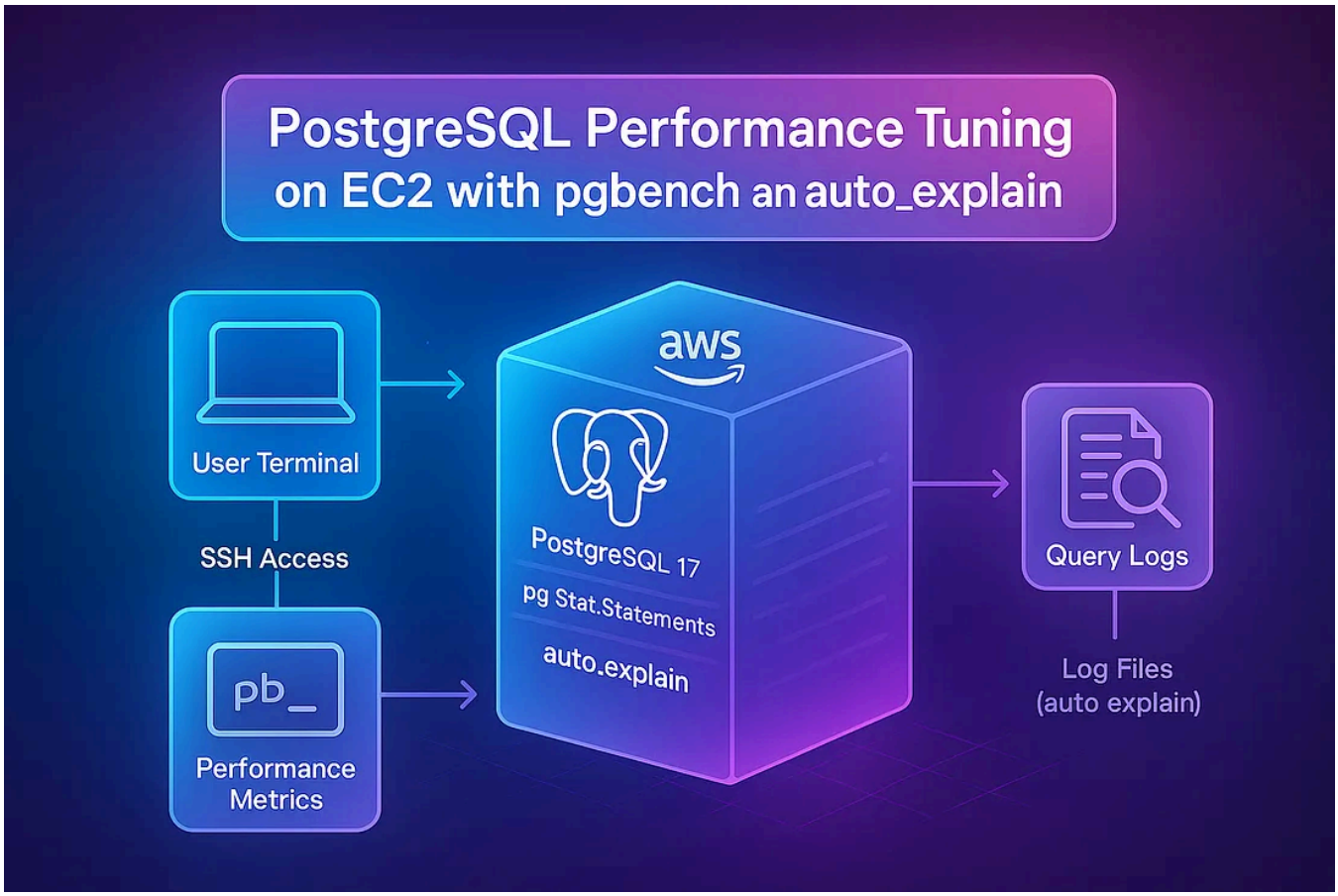
PostgreSQL Performance Tuning on EC2 with `pgbench` and `auto_explain`: A Hands-On Guide

19 min read · Jul 11, 2025

Jeyaram Ayyalusamy Following



PostgreSQL is widely regarded as one of the most powerful and reliable open-source relational database systems. Its reputation for data integrity, extensibility, and support for advanced SQL features makes it a preferred choice for developers, data engineers, and organizations around the globe.

However, it's important to understand a critical truth that often gets overlooked:

> **Installing PostgreSQL is not the final step — it's merely the foundation.**

In production environments, a default PostgreSQL setup is rarely suitable for handling high-volume workloads or delivering optimal performance. To ensure your instance is stable, scalable, and efficient, you must go **beyond installation** and invest in proper **benchmarking, monitoring, and tuning**.

🧩 The Limits of Default Configuration

When PostgreSQL is installed, it comes with a set of default configuration parameters. These defaults are conservative by design. They aim to ensure that the database can start and run safely on a wide range of systems, including low-memory development environments or local machines.

While this is beneficial for beginners or small-scale testing, it becomes a **bottleneck** in production use cases. Relying on default settings can lead to several performance and operational issues, such as:

- Suboptimal memory allocation for operations like sorting and joining.
- Low limits on maximum connections and worker processes.
- Inefficient autovacuum and background writer settings.
- Missing visibility into slow or inefficient queries.

The result? A database that may function but cannot **scale** or **respond efficiently** under real-world workloads.

🚦 Why Production-Ready PostgreSQL Requires More Than Installation

A production-grade PostgreSQL environment must be **observed**, **measured**, and **tuned** continuously. Below are the three essential areas where database administrators and engineers must focus their attention post-installation.

1. 📊 Benchmarking: Understand How PostgreSQL Performs Under Load

Before making tuning decisions, it is critical to understand how PostgreSQL behaves under stress. Benchmarking helps you:

- Simulate real-world traffic with concurrent users.
- Measure throughput (e.g., transactions per second).
- Identify the thresholds where performance starts to degrade.

Without benchmarking, optimization is guesswork. With it, you're making data-driven decisions.

2. 🛠 Monitoring: Gain Real-Time Insight into Database Behavior

Monitoring enables visibility into what PostgreSQL is doing internally. This includes:

- Query execution times and slow query patterns.
- Lock contention and deadlocks.
- Resource usage (CPU, memory, I/O).
- Log messages indicating potential issues.

Real-time monitoring tools and log inspection give you the ability to **detect anomalies early**, before they impact application performance or user experience.

3. 🔧 Tuning: Optimize PostgreSQL for Your Specific Workload

After benchmarking and monitoring, you can begin tuning PostgreSQL. This involves adjusting configuration parameters based on:

- Your system's hardware (e.g., CPU cores, available memory, IOPS).

- The nature of your workload (read-heavy, write-heavy, mixed).
- Query patterns and concurrency levels.

Examples of tuning actions include:

- Increasing `work_mem` to improve sort and join performance.
- Adjusting `shared_buffers` to cache more data in memory.
- Tuning `autovacuum` thresholds to prevent table bloat.
- Configuring `max_connections` and `parallel_workers` for better concurrency.

Proper tuning can significantly reduce query times, minimize disk I/O, and improve overall system throughput.

🧠 Conclusion: Operational Excellence Requires Continuous Effort

PostgreSQL is a powerful database engine, but power without control leads to inefficiency. If you stop at installation, you risk running a system that cannot handle real-world demands.

To fully leverage PostgreSQL in production environments, you must:

- Benchmark the database to understand its performance limits.
- Monitor query and system behavior to catch problems early.
- Tune configuration settings based on actual workload characteristics.

Each of these steps builds on the one before. Together, they transform PostgreSQL from a functional installation into a **high-performance, production-grade system** capable of scaling with your business.

🔬 PostgreSQL Performance Benchmarking with `pgbench`: A Detailed Guide for Realistic Load Testing

PostgreSQL is a powerful, open-source relational database engine trusted by developers, startups, and enterprises for its robust feature set and performance under pressure. However, performance isn't guaranteed out of the box — it must be **validated**, **monitored**, and **optimized**.

To effectively understand how PostgreSQL behaves under load, we need **repeatable benchmarks** that simulate real-world concurrency and transaction throughput. PostgreSQL provides a native tool called **`pgbench`** specifically designed for this task.

In this blog post, we'll explore:

- What `pgbench` is and why it matters
- How to execute `pgbench` tests with varying concurrency
- Why incremental benchmarking is crucial for uncovering performance bottlenecks

Each section includes **command-level examples**, **technical explanations**, and **best practices** to help you accurately measure your PostgreSQL system's behavior under different loads.

🔧 What Is `pgbench`?

`pgbench` is PostgreSQL's **built-in benchmarking utility** that simulates multiple clients running transactions against a PostgreSQL database. It is often used for:

- **Performance benchmarking**
- **Stress testing**
- **Latency and throughput measurement**
- **Tuning validation**

The tool initializes a sample schema with a standard set of tables and transaction logic, then runs a specified number of transactions using simulated concurrent clients. These transactions include `SELECT`, `INSERT`, `UPDATE`, and `DELETE` operations designed to reflect a typical OLTP (Online Transaction Processing) workload.

Why Use `pgbench`?

`pgbench` enables you to:

- Test PostgreSQL behavior under real-world-like traffic
- Simulate concurrency to find scaling limits
- Evaluate performance differences across environments or configuration changes
- Compare transaction throughput over time or under new infrastructure conditions

✅ Installing `pgbench` on Amazon EC2 (RHEL)

If you're setting up performance benchmarking with PostgreSQL on a Red Hat-based Amazon EC2 instance, you'll need to install the `pgbench` tool, which is part of the `postgresql-contrib` package.

Here's a step-by-step breakdown of the installation and validation:

🔧 Step 1: Install `postgresql-contrib`

Use the following command to install the required package, which includes `pgbench`:

```
sudo yum install -y postgresql17 postgresql17-server postgresql17-contrib
```

You may see a message like:

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

You can ignore this warning if you're using RHUI (Red Hat Update Infrastructure) on AWS.

During installation, you'll notice dependencies like `libxslt`, `uuid`, `postgresql`, and `postgresql-private-libs` are also installed.

Once prompted with:

```
Is this ok [y/N]:
```

Simply press `y` and hit Enter to proceed.

📦 Sample Output

```
[root@ip-172-31-92-46 ~]# sudo yum install -y postgresql17 postgresql17-server
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use "rhc" or

Last metadata expiration check: 0:05:22 ago on Thu Jul 10 22:42:09 2025.
Package postgresql17-17.5-3PGDG.rhel10.x86_64 is already installed.
Package postgresql17-server-17.5-3PGDG.rhel10.x86_64 is already installed.
Dependencies resolved.
================================================================================
 Package                                                 Architecture
================================================================================
Installing:
 postgresql17-contrib                                    x86_64
Installing dependencies:
 libxslt                                                 x86_64

Transaction Summary
================================================================================
Install  2 Packages

Total download size: 922 k
Installed size: 3.2 M
Downloading Packages:
```

```
(1/2): postgresql17-contrib-17.5-3PGDG.rhel10.x86_64.rpm
(2/2): libxslt-1.1.39-7.el10_0.x86_64.rpm
--------------------------------------------------------------------------------
Total
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
  Installing       : libxslt-1.1.39-7.el10_0.x86_64
  Installing       : postgresql17-contrib-17.5-3PGDG.rhel10.x86_64
  Running scriptlet: postgresql17-contrib-17.5-3PGDG.rhel10.x86_64
Installed products updated.

Installed:
  libxslt-1.1.39-7.el10_0.x86_64

Complete!
[root@ip-172-31-92-46 ~]#
```

```
Complete!
```

🔐 Step 2: Switch to the `postgres` User

After installation, switch to the PostgreSQL system user:

```
sudo su - postgres
```

✅ Step 3: Verify `pgbench` Installation

Run the following to confirm the tool is now available:

```
which pgbench
```

Expected output:

```
/usr/bin/pgbench
```

🎉 You're All Set!

You can now start using `pgbench` for initializing benchmark tables and running performance tests:

```
pgbench -i your_database_name
```

Make sure the target database exists and is accessible by the `postgres` user.

🍾 Step-by-Step: Running `pgbench` Tests on EC2

To keep benchmarking realistic, it's recommended to execute tests in an environment that closely mirrors production. In this case, we assume the tests are being conducted from an **AWS EC2 instance**, either connected locally to the PostgreSQL host or part of the same VPC/subnet. This setup minimizes network latency and emulates deployment conditions more accurately.

Each test case uses a **separate test database** to ensure clean initialization and prevent data or schema conflicts. This also enables reproducibility and side-by-side result comparison.

Let's walk through three benchmarking scenarios with increasing levels of concurrency: 250, 500, and 1000 clients. Each test runs 100 transactions per client to generate a sufficient and consistent load.

🔹 Test 1: 50 Clients, 100 Transactions

This scenario simulates a **moderately loaded environment**, such as a small-to-medium sized production workload during business hours.

✅ Step-by-Step Commands:

```
psql -c "CREATE DATABASE payments_bench_50;"
pgbench -i payments_bench_50
pgbench -c50 -t100 payments_bench_50
```

📦 Sample Output

```
[postgres@ip-172-31-92-46 ~]$ psql -c "CREATE DATABASE payments_bench_50;"
CREATE DATABASE
[postgres@ip-172-31-92-46 ~]$
```

```
[postgres@ip-172-31-92-46 ~]$ pgbench -i payments_bench_50
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
```

```
vacuuming...
creating primary keys...
done in 0.21 s (drop tables 0.00 s, create tables 0.01 s, client-side generate
[postgres@ip-172-31-92-46 ~]$
```

```
[postgres@ip-172-31-92-46 ~]$ pgbench -c50 -t100 payments_bench_50
pgbench (17.5)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 50
number of threads: 1
maximum number of tries: 1
number of transactions per client: 100
number of transactions actually processed: 5000/5000
number of failed transactions: 0 (0.000%)
latency average = 89.301 ms
initial connection time = 104.008 ms
tps = 559.905524 (without initial connection time)
[postgres@ip-172-31-92-46 ~]$
```

```
postgres=# \l+

       Name        |  Owner   | Encoding | Locale Provider | Collate |  Ctype
-------------------+----------+----------+-----------------+---------+---------
 payments_bench_50 | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
 postgres          | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
 template0         | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
                   |          |          |                 |         |
 template1         | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
                   |          |          |                 |         |
(4 rows)
```

```
postgres=#
postgres=#
```

```
postgres=# \c payments_bench_50
You are now connected to database "payments_bench_50" as user "postgres".
payments_bench_50=#
payments_bench_50=#
```

```
payments_bench_50=# \dt+
                                          List of relations
 Schema |       Name       | Type  |  Owner   | Persistence | Access method |
--------+------------------+-------+----------+-------------+---------------+--
 public | pgbench_accounts | table | postgres | permanent   | heap          | 1
 public | pgbench_branches | table | postgres | permanent   | heap          | 1
 public | pgbench_history  | table | postgres | permanent   | heap          | 4
 public | pgbench_tellers  | table | postgres | permanent   | heap          | 1
(4 rows)

payments_bench_50=#
```

🧠 What This Does:

- `CREATE DATABASE payments_bench_50;`

  Initializes a fresh database for isolated benchmarking.

- `pgbench -i payments_bench_50`

  Loads the standard `pgbench` schema and populates it with initial test data.

- `pgbench -c50 -t100 payments_bench_50`

  Executes the benchmark:

- `-c50` : Simulates **50 concurrent clients**
- `-t100` : Each client runs **100 transactions**, totaling 5,000 transactions

🎯 Objective:

This test evaluates the system's **baseline performance** under moderate concurrency. It's particularly useful for identifying:

- Average response time under moderate pressure
- Whether the system can sustain this level without performance degradation
- Potential early signs of lock contention or resource exhaustion

🔹 Test 2: 25 Clients, 100 Transactions

This test reduces concurrency to represent a **lightweight transactional load**. This scenario may reflect a staging environment or a production system during off-peak hours.

✅ Step-by-Step Commands:

```
psql -c "CREATE DATABASE payments_bench_25;"
pgbench -i payments_bench_25
pgbench -c25 -t100 payments_bench_25
```

🧠 What This Does:

- `-c25` : Launches **25 client sessions**
- `-t100` : Each client runs **100 transactions**, for a total of 2,500 transactions

🎯 Objective:

This low-concurrency test provides a **baseline metric** that future tuning or configuration changes can be compared against. It's useful for:

- Measuring performance without stress factors
- Validating system readiness before going live

- Ensuring no fundamental inefficiencies exist in a low-load scenario

🔹 Test 3: 100 Clients, 100 Transactions

This final test represents a **high-concurrency environment**, pushing the system toward its operational limits. It's designed for **stress testing** and is ideal for observing how PostgreSQL behaves under heavy load.

✅ Step-by-Step Commands:

```
psql -c "CREATE DATABASE payments_bench_1001;"
pgbench -i payments_bench_100
pgbench -c100 -t100 payments_bench_100
```

```
postgres=# \l+

        Name         |  Owner   | Encoding | Locale Provider | Collate |  Ctype
---------------------+----------+----------+-----------------+---------+--------
 payments_bench_100  | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
 payments_bench_25   | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
 payments_bench_50   | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
 postgres            | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
 template0           | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
                     |          |          |                 |         |
 template1           | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8
                     |          |          |                 |         |
(6 rows)

postgres=#
```

🧠 What This Does:

- `-c100` : Simulates **100 concurrent clients**, a high-concurrency scenario
- `-t100` : Each client performs 100 transactions (10,000 total)

🎯 Objective:

This test helps to:

- Identify resource bottlenecks (CPU, memory, I/O)
- Detect concurrency-related issues such as lock contention or deadlocks
- Evaluate the system's maximum throughput in terms of TPS (Transactions Per Second)

✅ Best Practice: Incremental Load Testing

Running tests at different concurrency levels (low → medium → high) enables a **performance curve analysis.** This helps you:

- **Compare results across scenarios** to find the inflection point where performance drops
- **Pinpoint exact thresholds** where database tuning becomes necessary
- **Tune PostgreSQL parameters** such as `work_mem`, `shared_buffers`, `max_connections`, and `autovacuum` settings

You should also monitor system metrics in parallel using tools like:

- `htop` for CPU/memory usage
- `iostat` or `vmstat` for disk and I/O monitoring
- PostgreSQL logs for query execution times and error tracking

Benchmarking is most effective when test results are correlated with system and query performance metrics.

📈 Summary

Benchmarking with `pgbench` is a practical and reliable way to simulate PostgreSQL under real-world concurrency. Using incremental concurrency tests—starting from 25 clients and increasing to 100—allows database administrators and engineers to:

- Validate system stability
- Identify and diagnose bottlenecks
- Measure baseline and peak throughput
- Guide tuning decisions for production environments

Each test scenario offers valuable insights into how PostgreSQL scales with load and helps ensure your infrastructure can meet future demand with confidence.

🔍 Enabling `auto_explain` in PostgreSQL: Unlock Deep Performance Insights

When performance issues arise in a PostgreSQL database, one of the most critical questions is:

> *Why is this query slow, and what is it actually doing under the hood?*

Tools like `pgbench` are excellent for **simulating traffic** and **measuring throughput**, but they do not provide visibility into the **execution plans** of individual slow queries. This is where PostgreSQL's `auto_explain` module becomes essential.

`auto_explain` enables PostgreSQL to automatically log **execution plans** of queries that exceed a certain duration, offering granular insight into how PostgreSQL is processing each SQL statement. This information is vital for understanding and fixing performance bottlenecks, especially in production environments.

🎯 Why Use `auto_explain`?

While traditional `EXPLAIN` or `EXPLAIN ANALYZE` commands can show query plans manually, `auto_explain` provides an **automated, always-on mechanism** for capturing execution details across all sessions and queries—without modifying application code.

With `auto_explain`, you can:

- Understand full **query behavior** without manual intervention.
- Spot inefficient **joins**, **sequential scans**, or **missing indexes**.
- Trace execution inside **stored procedures**, **triggers**, and **nested subqueries**.
- Analyze **buffer usage**, execution timing, and resource-intensive stages.

This type of visibility enables proactive query tuning and system optimization based on real-world workloads.

⚙️ Option 1: Load `auto_explain` Temporarily for a Single Session

For **ad-hoc analysis** or manual testing, `auto_explain` can be loaded for just the current session.

✅ Command:

```
psql -c "LOAD 'auto_explain';"
```

📌 Details:

- This command loads the extension into **only the active session**.
- All queries run afterward in this session will be evaluated for logging if further parameters (like `log_min_duration`) are set.
- Once the session ends, the extension is **unloaded automatically**.

📎 Use Case:

This method is ideal when:

- Testing queries inside a development or staging environment.
- Executing SQL scripts manually and capturing plans for review.
- Avoiding persistent system-level changes.

⚠️ **Note:** This approach does not persist across sessions or database restarts.

🧱 Option 2: Enable `auto_explain` Server-Wide (Recommended)

For full visibility in production or performance testing environments, it is recommended to **enable `auto_explain` globally**. This setup ensures that slow query plans are logged across all client sessions, even after system restarts.

🔧 Step 1: Enable Extension at Server Startup

To make `auto_explain` available globally, PostgreSQL must **preload the extension** using the `shared_preload_libraries` parameter.

✅ Commands:

```
vi /var/lib/pgsql/17/data/postgresql.conf
shared_preload_libraries = 'pg_stat_statements, auto_explain'

sudo systemctl restart postgresql-17
```

```
[postgres@ip-172-31-92-46 ~]$ cat /var/lib/pgsql/17/data/postgresql.conf | grep
shared_preload_libraries = 'pg_stat_statements, auto_explain'           # (char
```

```
[postgres@ip-172-31-92-46 ~]$
```

```
[root@ip-172-31-92-46 ~]# sudo systemctl restart postgresql-17
[root@ip-172-31-92-46 ~]#
```

📌 Explanation:

- `ALTER SYSTEM SET` modifies PostgreSQL's `postgresql.auto.conf` to apply the setting persistently.
- `shared_preload_libraries` is a special parameter that requires a **restart** of the PostgreSQL service to take effect.
- Including `pg_stat_statements` along with `auto_explain` allows future integration with other diagnostics tools.

🔧 Step 2: Tune `auto_explain` Logging Parameters

Once the extension is enabled, configure **what**, **when**, and **how** query plans are logged. This step is critical to avoid excessive or insufficient logging.

✅ Configuration Commands:

```
psql -c "ALTER SYSTEM SET auto_explain.log_format TO 'json';"
psql -c "ALTER SYSTEM SET auto_explain.log_min_duration TO '1000';"
psql -c "ALTER SYSTEM SET auto_explain.log_analyze TO 'on';"
psql -c "ALTER SYSTEM SET auto_explain.log_buffers TO 'on';"
psql -c "ALTER SYSTEM SET auto_explain.log_timing TO 'off';"
psql -c "ALTER SYSTEM SET auto_explain.log_triggers TO 'on';"
psql -c "ALTER SYSTEM SET auto_explain.log_verbose TO 'on';"
psql -c "ALTER SYSTEM SET auto_explain.log_nested_statements TO 'on';"
psql -c "ALTER SYSTEM SET auto_explain.sample_rate TO '1';"
```

📦 Sample Output

```
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_format
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_min_du
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_analyz
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_buffer
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_timing
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_trigge
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_verbos
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.log_nested
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "ALTER SYSTEM SET auto_explain.sample_rat
ALTER SYSTEM
[postgres@ip-172-31-92-46 ~]$
```

📜 Parameter Breakdown:

Parameter	Description
`log_format = 'json'`	Produces structured log output that is easier to parse using logging tools like ELK or CloudWatch.
`log_min_duration = 1000`	Only log queries that run longer than **1000 milliseconds** (1 second).
`log_analyze = 'on'`	Collects actual execution statistics—similar to `EXPLAIN ANALYZE`.
`log_buffers = 'on'`	Includes buffer access details (disk vs memory reads).
`log_timing = 'off'`	Disables detailed per-operation timing (reduces logging overhead).
`log_triggers = 'on'`	Captures time spent within any **triggers** fired by the query.
`log_verbose = 'on'`	Logs fully qualified object names and more granular details.
`log_nested_statements = 'on'`	Logs execution plans inside **functions**, **procedures**, and **subqueries**.
`sample_rate = '1'`	Logs **100%** of qualifying queries (set <1 to reduce log volume if needed).

⚠️ *Important*: Overly aggressive logging on high-traffic systems can lead to large log files. You can adjust `log_min_duration` or `sample_rate` based on system load.

🔧 Step 3: Reload Configuration

After applying the above settings, **reload PostgreSQL configuration** to activate changes without restarting the server.

✅ Command:

```
psql -c "SELECT pg_reload_conf();"
```

📦 Sample Output

```
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$ psql -c "SELECT pg_reload_conf();"
 pg_reload_conf
----------------
 t
(1 row)

[postgres@ip-172-31-92-46 ~]$
```

This command applies all the changes from the `ALTER SYSTEM SET` statements. Once reloaded, `auto_explain` will begin logging execution plans based on the configured duration and sampling thresholds.

📈 Real-World Impact of `auto_explain`

Once properly configured, `auto_explain` enhances PostgreSQL observability significantly. You can now:

- Identify **slow-running queries** and examine why they're slow.
- Detect **missing indexes** or **costly sequential scans**.
- Visualize query complexity using JSON logs in tools like Kibana.
- Pinpoint **recursive triggers** or **inefficient subplans** that impact response time.

For example, a query that joins multiple large tables without indexes may look fast during development but become a bottleneck under production load. With `auto_explain`, the execution plan reveals:

- Join method used (e.g., Nested Loop vs Hash Join)
- Estimated vs. actual row counts
- Time spent in each step of the plan

🧠 Conclusion

`auto_explain` is a powerful diagnostic extension in PostgreSQL that helps surface execution plans for slow queries—**automatically and transparently.** It is an essential tool for database administrators, developers, and DevOps engineers who need actionable insights into query performance.

By configuring `auto_explain`:

- Temporarily (for session-level debugging), or
- Permanently (for system-wide observability),

…you ensure that every problematic query has a traceable plan, allowing you to **optimize workloads**, **reduce latency**, and **increase overall database efficiency**.

📄 Monitoring PostgreSQL Logs in Real Time: A Critical Step in Performance Tuning

When tuning a PostgreSQL database for performance, running benchmarks and configuring logging are crucial. But without **real-time visibility into the database's internal operations**, you're flying blind. PostgreSQL's log files are a rich source of operational data, and monitoring them live allows you to **correlate performance issues directly with query activity**.

In this article, you'll learn how to:

- Stream PostgreSQL logs live during benchmarking
- Understand log output generated by `auto_explain`
- Use this data to support performance diagnostics

By the end of this walkthrough, you'll be able to observe slow queries and execution plans **as they happen**, enabling immediate feedback during testing or performance evaluation.

🔍 Live Log Monitoring with `tail`

PostgreSQL writes logs to files on disk, and those logs contain:

- Query execution durations
- Detailed execution plans (when `auto_explain` is enabled)
- Warnings, errors, and system messages

Instead of opening the log file manually every time, you can use the `tail` command to **follow the log output in real time**, providing a continuous stream of insights during benchmarking or debugging sessions.

🛠 Step-by-Step Instructions

1. Switch to the PostgreSQL User

Depending on your system configuration, you may need to become the `postgres` user to access logs.

```
sudo su - postgres
```

This ensures you have permission to read PostgreSQL's log files directly.

```
[root@ip-172-31-92-46 ~]# sudo su - postgres
Last login: Fri Jul 11 00:41:57 UTC 2025 on pts/2
[postgres@ip-172-31-92-46 ~]$
[postgres@ip-172-31-92-46 ~]$
```

2. Tail the PostgreSQL Log File

Run the following command to start watching the log file:

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

- `-10` : Outputs the **last 10 lines** of the log file initially.
- `-f` : Keeps the process open and continuously streams new entries as they are written to the log file.

```
[postgres@ip-172-31-92-46 log]$ tail -10f /var/lib/pgsql/17/data/log/postgresql
                "Temp Read Blocks": 0,
                "Temp Written Blocks": 0
              }
            ]
          },
          "Query Identifier": 7377268743118387427,
          "Triggers": [
          ]
        }
2025-07-11 01:06:55.105 UTC [4248] LOG:  checkpoint starting: time
```

This setup effectively turns your terminal into a **live monitoring window** for all PostgreSQL activities recorded in the logs.

> 💡 Adjust the log path if you're using a different PostgreSQL version or distribution.

🧪 Example: Query Plan Logging Output

If you've previously configured `auto_explain`, slow queries (i.e., those exceeding the configured `log_min_duration`) will generate entries like the following:

```
[postgres@ip-172-31-92-46 log]$ tail -66f /var/lib/pgsql/17/data/log/postgresql
        }
2025-07-11 01:06:50.062 UTC [4712] LOG:  duration: 1328.619 ms  plan:
        {
          "Query Text": "UPDATE pgbench_tellers SET tbalance = tbalance + -533
          "Plan": {
            "Node Type": "ModifyTable",
            "Operation": "Update",
            "Parallel Aware": false,
            "Async Capable": false,
            "Relation Name": "pgbench_tellers",
            "Schema": "public",
            "Alias": "pgbench_tellers",
            "Startup Cost": 0.14,
            "Total Cost": 8.17,
            "Plan Rows": 0,
            "Plan Width": 0,
            "Actual Rows": 0,
            "Actual Loops": 1,
            "Shared Hit Blocks": 140,
            "Shared Read Blocks": 0,
            "Shared Dirtied Blocks": 0,
            "Shared Written Blocks": 0,
            "Local Hit Blocks": 0,
            "Local Read Blocks": 0,
            "Local Dirtied Blocks": 0,
            "Local Written Blocks": 0,
            "Temp Read Blocks": 0,
            "Temp Written Blocks": 0,
            "Plans": [
              {
                "Node Type": "Index Scan",
                "Parent Relationship": "Outer",
                "Parallel Aware": false,
                "Async Capable": false,
                "Scan Direction": "Forward",
                "Index Name": "pgbench_tellers_pkey",
                "Relation Name": "pgbench_tellers",
                "Schema": "public",
                "Alias": "pgbench_tellers",
                "Startup Cost": 0.14,
                "Total Cost": 8.17,
                "Plan Rows": 1,
                "Plan Width": 10,
                "Actual Rows": 1,
                "Actual Loops": 1,
                "Output": ["(tbalance + '-533'::integer)", "ctid"],
                "Index Cond": "(pgbench_tellers.tid = 5)",
                "Rows Removed by Index Recheck": 0,
                "Shared Hit Blocks": 5,
                "Shared Read Blocks": 0,
```

```
                    "Shared Dirtied Blocks": 0,
                    "Shared Written Blocks": 0,
                    "Local Hit Blocks": 0,
                    "Local Read Blocks": 0,
                    "Local Dirtied Blocks": 0,
                    "Local Written Blocks": 0,
                    "Temp Read Blocks": 0,
                    "Temp Written Blocks": 0
                  }
                ]
              },
              "Query Identifier": 7377268743118387427,
              "Triggers": [
              ]
            }
2025-07-11 01:06:55.105 UTC [4248] LOG:  checkpoint starting: time
```

🔍 How to Interpret This

- **`duration: 1123.54 ms`** : The query took approximately 1.1 seconds to execute.
- **`Seq Scan`** : A sequential scan was used, which may indicate that no suitable index was found.
- **`Filter`** : The query applied a condition to filter rows—here, `abalance < 0`.

This level of insight is extremely valuable. Without `auto_explain`, you would only know that a query was slow. With `auto_explain` and live log monitoring, you can see **why** it was slow—whether due to missing indexes, poor join strategies, or inefficient filter conditions.

📌 When to Use Real-Time Log Monitoring

Real-time log monitoring is particularly effective when:

- ✅ Running **pgbench** or other benchmarking tools and needing immediate feedback.
- ✅ Evaluating the impact of recent **indexing or configuration changes**.

- ✅ Troubleshooting production incidents where query performance is degrading.
- ✅ Analyzing the behavior of **stored procedures, triggers, or nested queries.**

During benchmarking, it's common to generate thousands of transactions. Some of these will be fast, but others may degrade due to I/O contention, lock waits, or inefficient execution plans. **Tailing the logs helps catch these moments as they occur.**

🚀 Final Thoughts

With PostgreSQL 17 deployed on an EC2 instance, and equipped with live log inspection capabilities, you now have a **powerful diagnostic workflow** that closely rivals enterprise observability tools — without requiring any third-party services.

Let's recap the core achievements of this setup:

✅ What You've Accomplished:

1. **Simulated transactional load** using `pgbench`, with customized database instances to prevent conflicts and isolate results.
2. **Configured** `auto_explain` to automatically log execution plans for slow queries, capturing deep insight into performance characteristics.
3. **Monitored logs in real time** using `tail`, correlating benchmark results with execution plans to pinpoint bottlenecks.

This end-to-end workflow bridges the gap between **synthetic performance testing** and **real-world database introspection.**

🎯 Conclusion

PostgreSQL performance tuning should not be based on assumptions. Instead, it should rely on:

- **Quantitative metrics** from `pgbench`
- **Execution transparency** from `auto_explain`
- **Live operational context** from real-time log monitoring

Together, these tools offer a comprehensive view of how PostgreSQL behaves under load. They allow you to **measure precisely**, **observe deeply**, and **optimize confidently**.

Whether you're preparing for scale, diagnosing production performance issues, or fine-tuning query execution, real-time PostgreSQL log monitoring is a critical part of your toolkit.

📢 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 | Data | AWS | Oracle

J

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

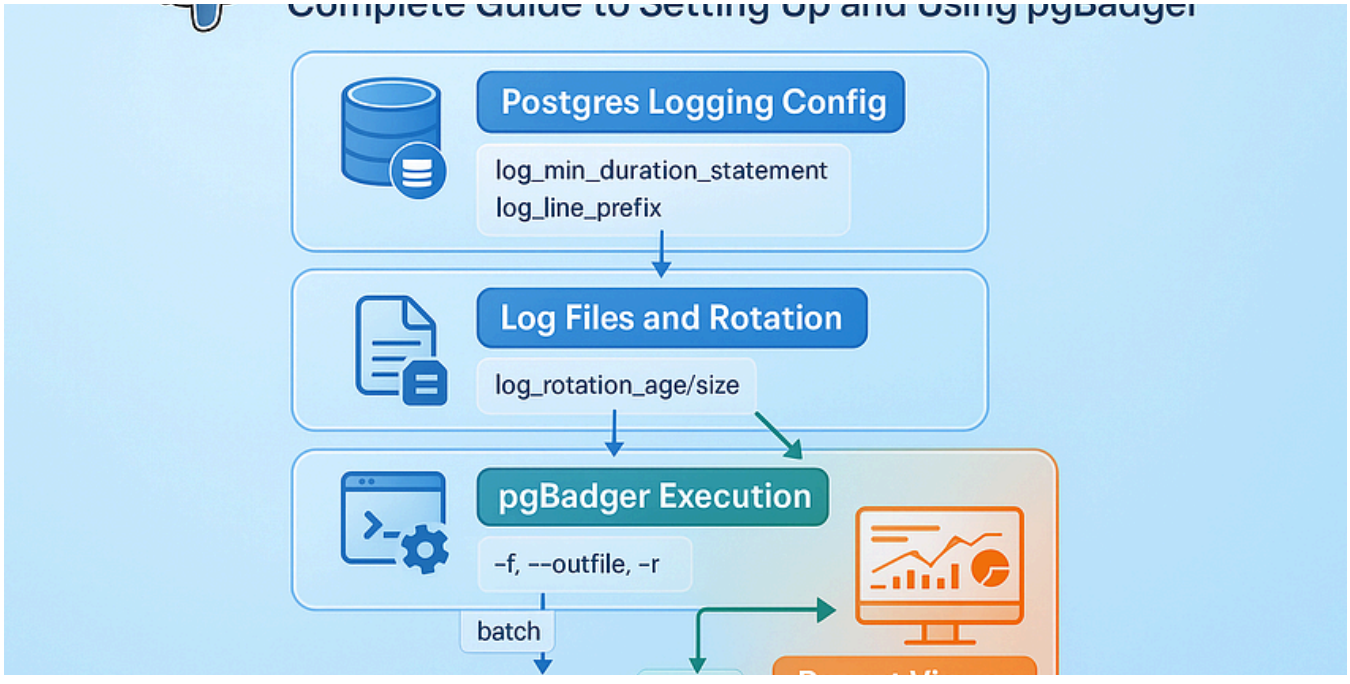
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

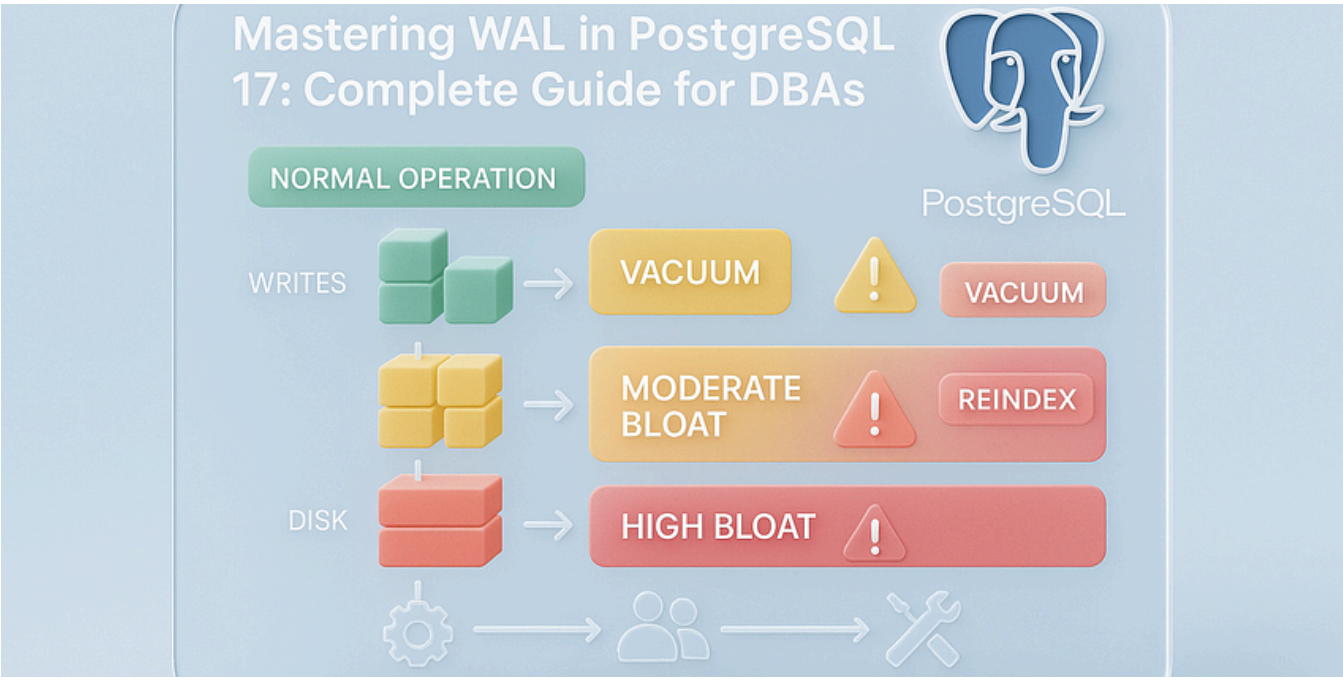


Jeyaram Ayyalusamy

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

Managing a production-grade PostgreSQL instance isn't just about keeping data safe—it's about monitoring performance, identifying...

Jun 23 52

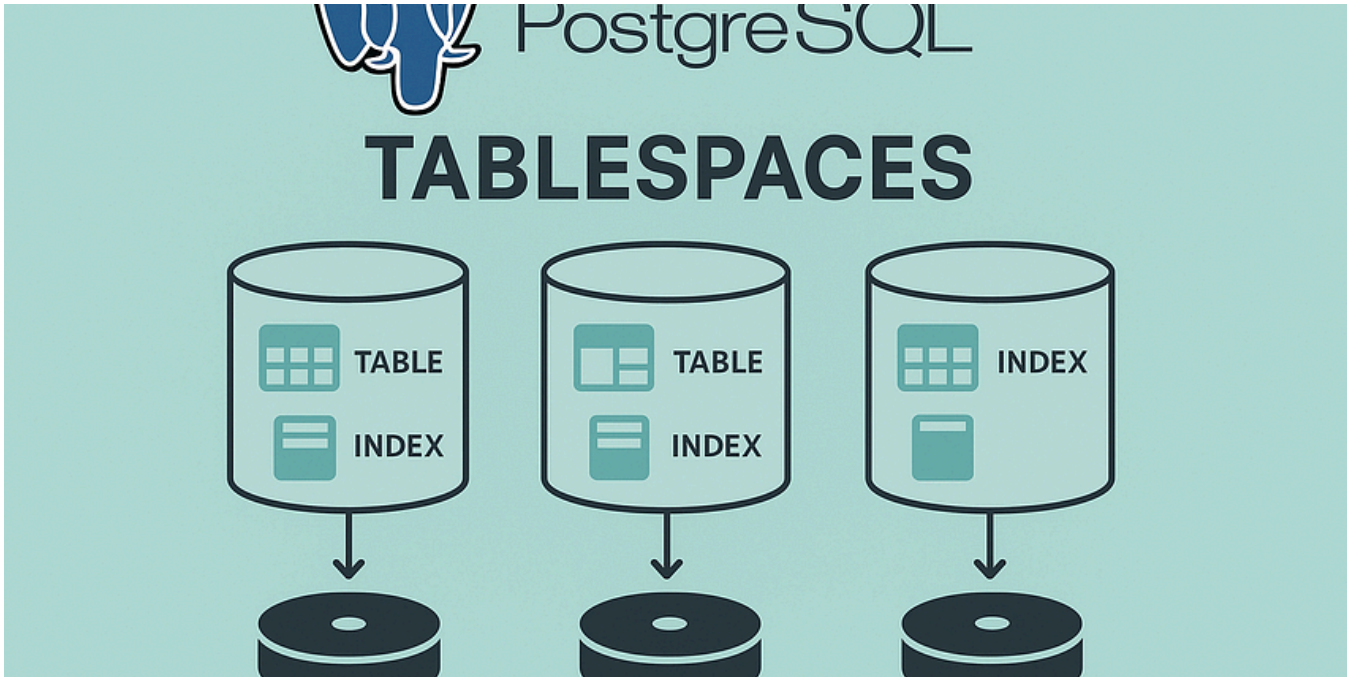


Jeyaram Ayyalusamy

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

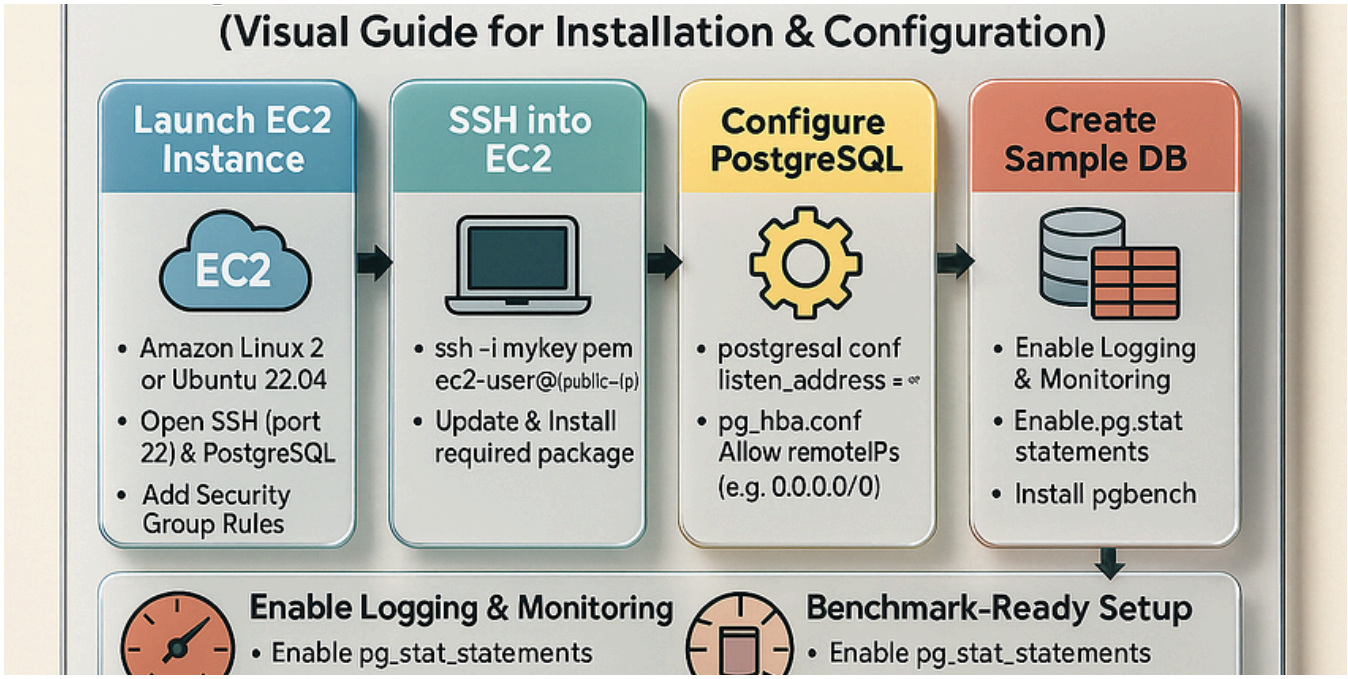


Jeyaram Ayyalusamy

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

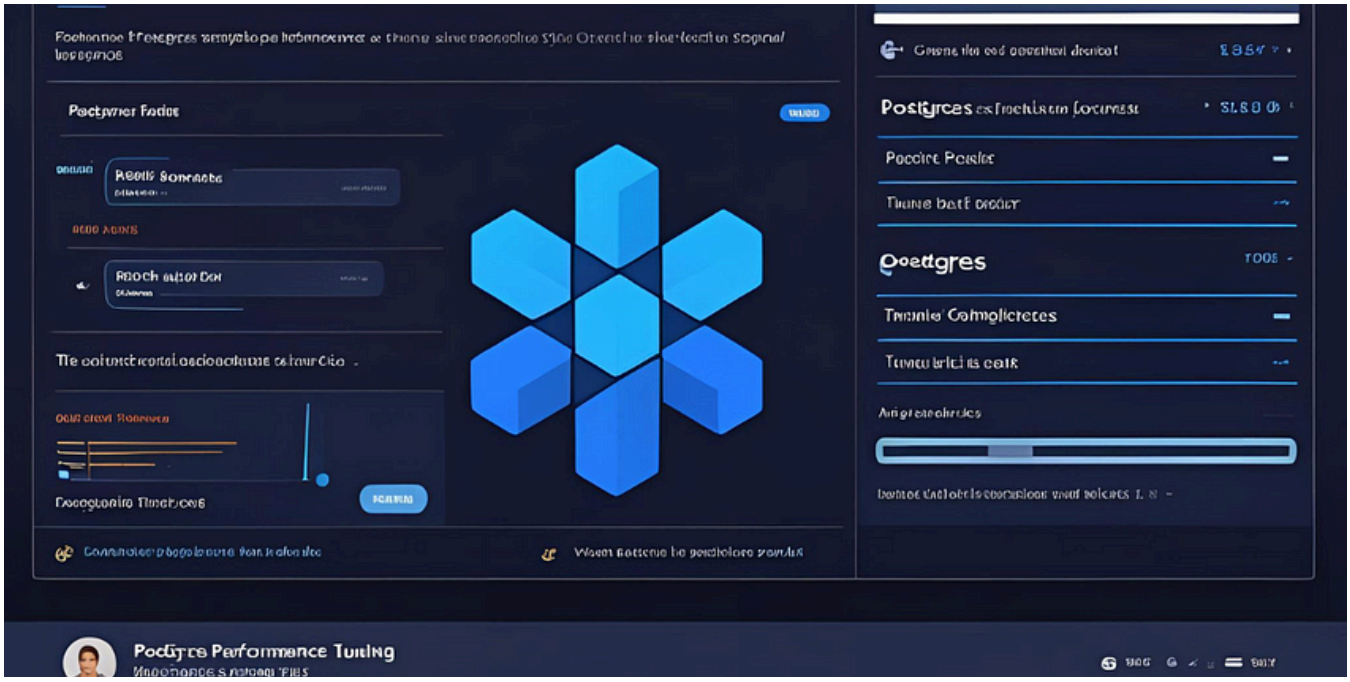
PostgreSQL 17 on AWS EC2—Full Installation & Configuration Walkthrough

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

See all from Jeyaram Ayyalusamy

Recommended from Medium

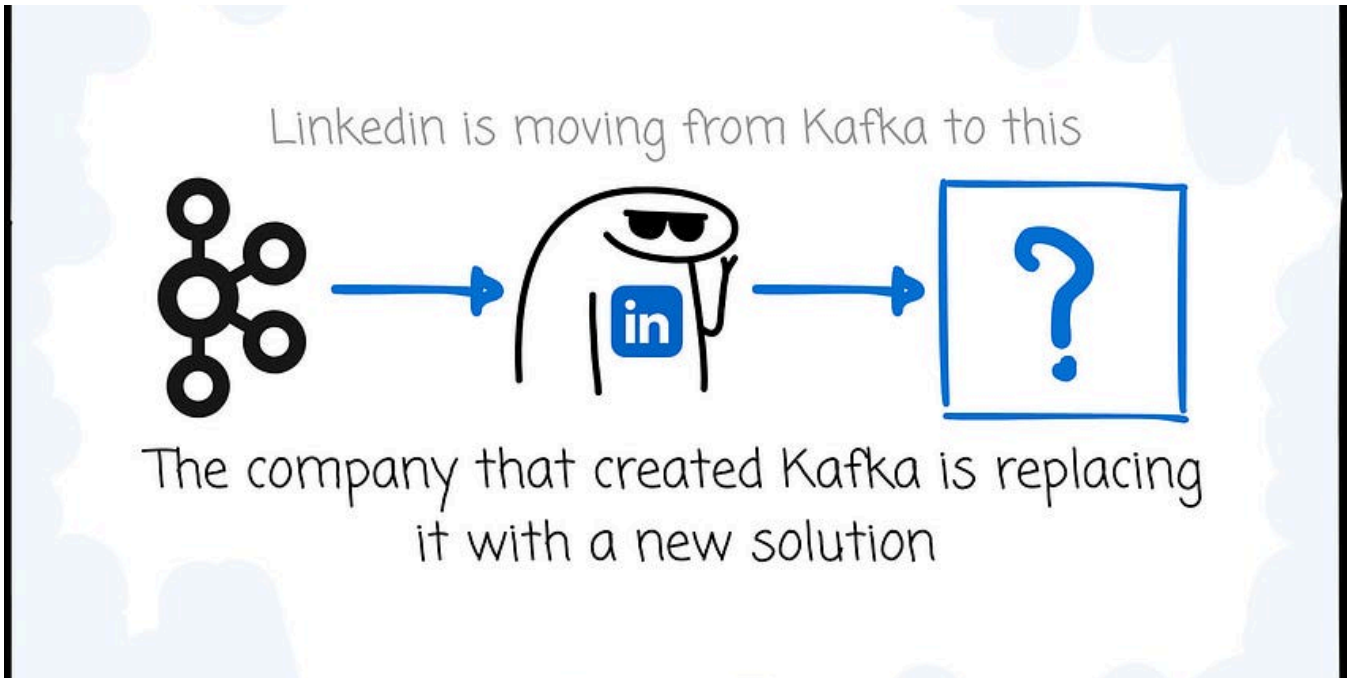


Rizqi Mulki

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55

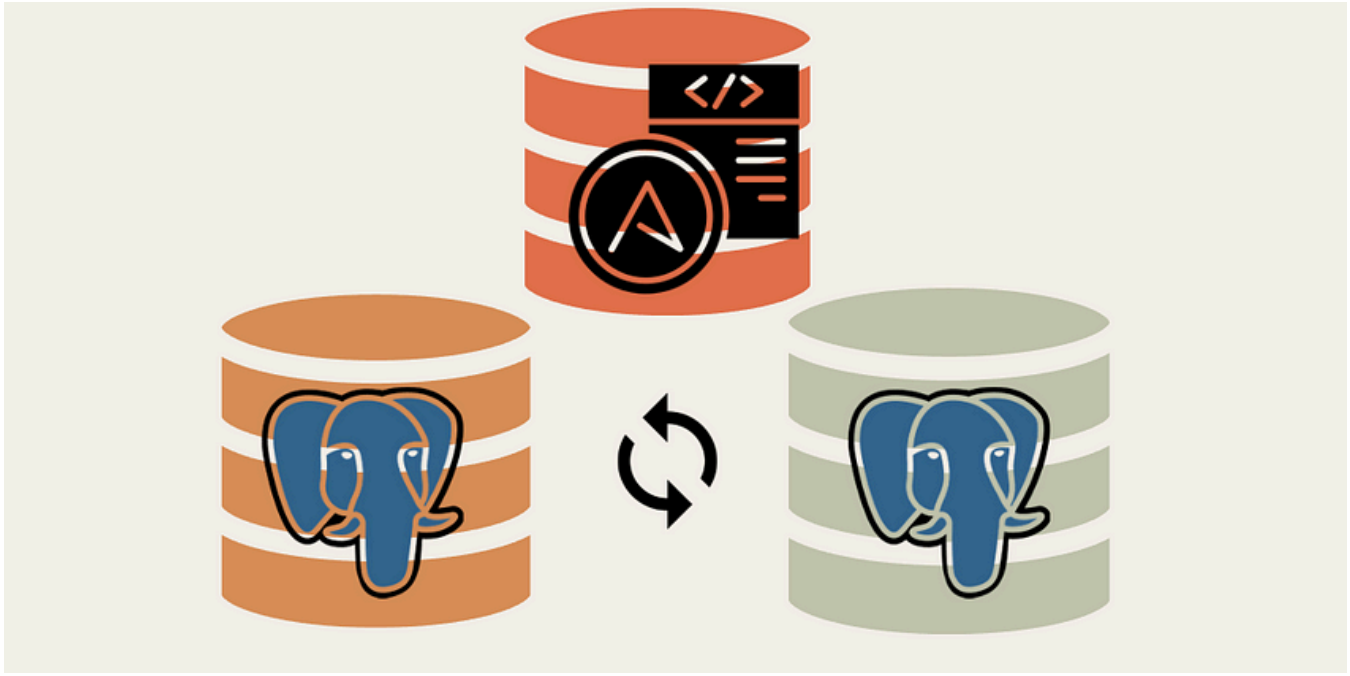


In Data Engineer Things by Vu Trinh

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

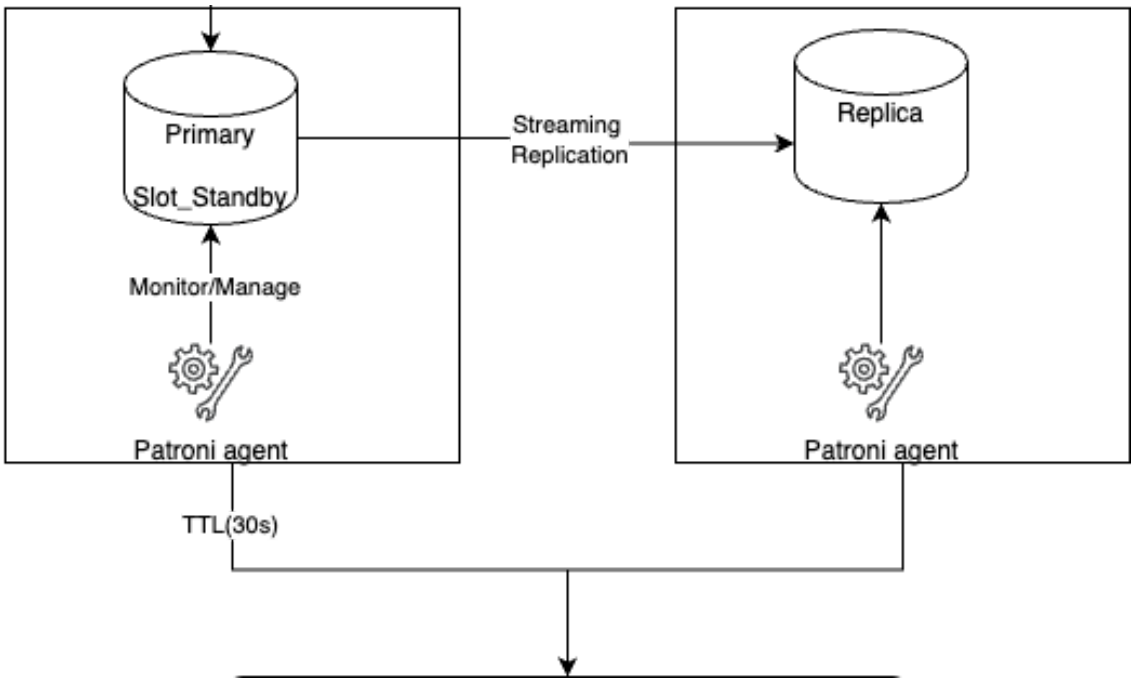


Oz

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



PAWAN SHARMA

PostgreSQL Replication Internals & High Availability with Patroni

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

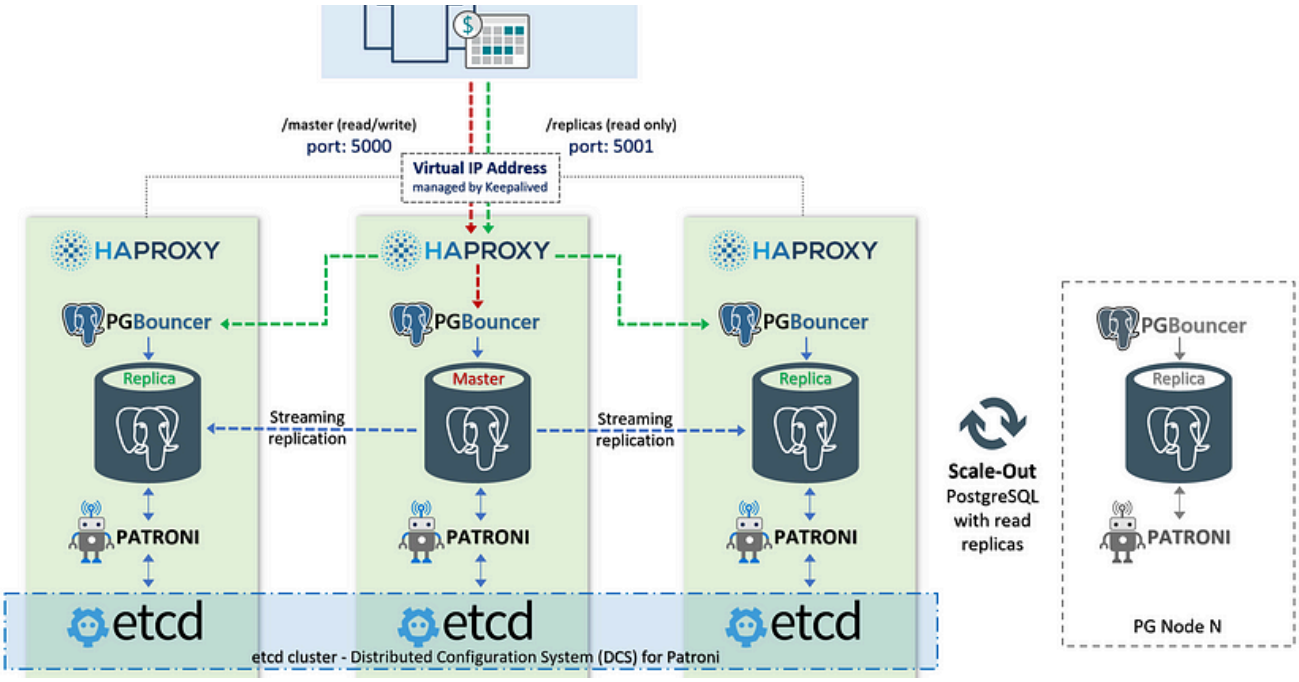
Jul 12 2

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



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