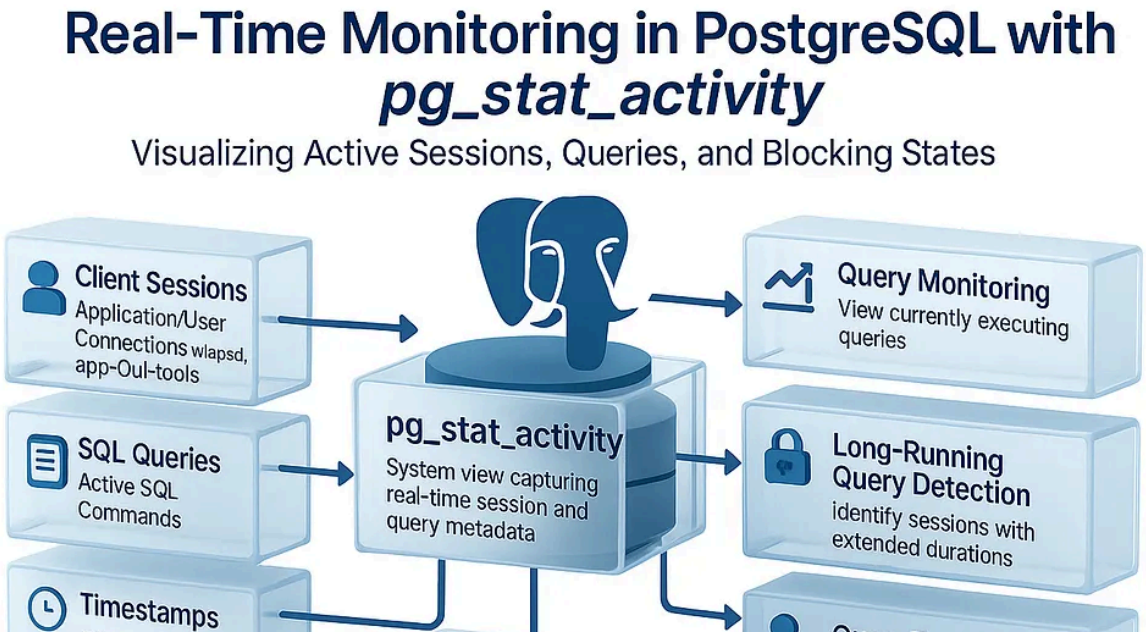
Monitoring Active Queries in PostgreSQL: Real-Time Performance Diagnostics Using `pg_stat_activity`

17 min read · Jul 12, 2025

Jeyaram Ayyalusamy Following



PostgreSQL administrators often face the challenge of diagnosing slow performance, managing long-running queries, and identifying blocked sessions —

especially in production environments where every second counts. Fortunately, PostgreSQL provides a powerful built-in system view called `pg_stat_activity`, which enables **real-time visibility into active database sessions.**

In this guide, we'll walk through a **practical PostgreSQL monitoring framework** using SQL queries and scripts that leverage `pg_stat_activity` —ideal for production troubleshooting, performance diagnostics, and query auditing.

📘 What Is `pg_stat_activity`?

`pg_stat_activity` is a system catalog view in PostgreSQL that **exposes real-time information about all active sessions** connected to the database. It includes details such as:

- Session PID (process ID)
- Username and client address
- Query text and current state
- Backend start and query start time
- Wait events and lock information

This view is invaluable for understanding what your PostgreSQL instance is doing at any given moment.

> 🧠 Tip: You must be a superuser or have proper monitoring privileges to access all session data.

🧪 Core Diagnostic Query: View Active Queries

To monitor currently running queries:

```
SELECT pid,
       usename,
```

```
        datname,
        client_addr,
        application_name,
        state,
        wait_event_type,
        wait_event,
        backend_start,
        query_start,
        now() - query_start AS runtime,
        query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY runtime DESC;
```

```
  pid   | usename  | datname  | client_addr | application_name | state  | wait_e
-------+----------+----------+-------------+------------------+--------+-------
---------------
 38820 |          | postgres |             |                  | active | Timeou
c.big_table
 38841 | postgres | postgres |             | psql             | active | IO
_table        +
       |          |          |             |                  |        |
 38840 | postgres | postgres |             | psql             | active | IO
_table        +
       |          |          |             |                  |        |
 38792 | postgres | postgres |             | psql             | active | IO
_table        +
       |          |          |             |                  |        |
 38776 | postgres | postgres |             | psql             | active |
              +
       |          |          |             |                  |        |
              +
       |          |          |             |                  |        |
              +
       |          |          |             |                  |        |
              +
       |          |          |             |                  |        |
              +
       |          |          |             |                  |        |
              +
       |          |          |             |                  |        |
              +
       |          |          |             |                  |        |
              +
```

```
        |          |          |              |                    |        |
               +
        |          |          |              |                    |        |
               +
        |          |          |              |                    |        |
rt AS runtime,+
        |          |          |              |                    |        |
               +
        |          |          |              |                    |        |
               +
        |          |          |              |                    |        |
               +
        |          |          |              |                    |        |
(5 rows)
```

✅ What This Tells You:

- Who is running queries right now
- Which queries are long-running
- What those queries are doing (text + wait state)

> 📌 Best Practice: Run this query periodically or integrate it into a dashboard for continuous visibility.

⏱️ Identify Long-Running Queries

Long-running queries can lock resources or block other sessions. Use the following query to isolate them:

```
SELECT pid,
       now() - query_start AS duration,
       usename,
       query
FROM pg_stat_activity
WHERE state = 'active'
  AND now() - query_start > interval '5 minutes'
ORDER BY duration DESC;
```

```
 pid   |    duration     | usename  |                  query
-------+-----------------+----------+---------------------------------------
 1824  | 00:04:00.363051 | postgres | INSERT INTO big_table (data)         +
       |                 |          | SELECT repeat('x', 1000)             +
       |                 |          | FROM generate_series(1, 10000000000);
(1 row)

postgres=#
```

✅ Best Practice:

Set the time threshold (`interval '5 minutes'`) based on your workload expectations.

> 🛑 Consider terminating queries that consistently exceed normal execution time and are impacting system performance.

🔒 Detect Blocking Sessions and Wait Events

PostgreSQL sessions can be blocked waiting for locks or other resources. Identify blockers and blocked processes using:

```
SELECT blocked.pid AS blocked_pid,
       blocked.query AS blocked_query,
       blocking.pid AS blocking_pid,
       blocking.query AS blocking_query
FROM pg_stat_activity blocked
JOIN pg_locks blocked_locks
  ON blocked_locks.pid = blocked.pid AND NOT blocked_locks.granted
JOIN pg_locks blocking_locks
  ON blocking_locks.locktype = blocked_locks.locktype
 AND blocking_locks.database IS NOT DISTINCT FROM blocked_locks.database
 AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
 AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
 AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
 AND blocking_locks.pid != blocked_locks.pid
```

```
  JOIN pg_stat_activity blocking
    ON blocking.pid = blocking_locks.pid;
```

✅ What This Reveals:

- Which session is blocked
- Who is blocking it
- What queries are involved on both sides

> 🧩 Use this insight to resolve lock contention and prevent cascading performance issues.

🔄 Kill Problematic Sessions (With Caution)

To terminate a session that is blocking others or running a problematic query:

```
SELECT pg_terminate_backend(<pid>);
```

Replace `<pid>` with the actual process ID from your diagnostic queries.

> ⚠️ Warning: Only terminate sessions after reviewing their impact and confirming that they are not part of critical transactions.

🖥️ Automate with Cron or Monitoring Tools

Integrate your `pg_stat_activity` queries with:

- **Cron jobs** to log or alert on long-running queries
- **pgAdmin dashboards** for real-time visualization
- **Monitoring stacks** like Prometheus + Grafana (via PostgreSQL exporters)

> 💡 Tip: Maintain audit logs of terminated sessions for compliance and root-cause analysis.

📊 Sample Monitoring Strategy

Use Case Query Type Frequency Active session audit All active queries Every minute Long-running query detection Duration > 5 mins Every 5 mins Blocking session detection Lock wait analysis via joins On alert only Session termination `pg_terminate_backend(pid)` Manual review Logging & Reporting Save results to table or log file Nightly

✅ Best Practices for Using `pg_stat_activity`

- **Filter by `state = 'active'`** to focus only on running queries.
- **Use `now() - query_start`** to calculate execution duration.
- **Monitor `wait_event_type` and `wait_event`** to diagnose blocking and IO issues.
- **Avoid terminating sessions blindly** — always investigate the context.
- **Combine with `pg_locks`** for full lock conflict analysis.

🔍 Monitoring PostgreSQL in Real Time with `pg_stat_activity`: Advanced Diagnostics for DBAs

PostgreSQL is widely recognized for its performance, extensibility, and reliability in handling modern application workloads. However, as database complexity increases, so does the need for visibility into **live query activity**, **session management**, and **transaction monitoring** — especially in production environments where uptime and performance are critical.

Fortunately, PostgreSQL provides a powerful system view called **`pg_stat_activity`**, which offers real-time access to vital session-level details such as active queries,

client connections, session states, and lock conditions.

In this article, we will take an in-depth look at how to use `pg_stat_activity` to build a **comprehensive PostgreSQL monitoring framework**, complete with detailed SQL queries and best practices for real-time database diagnostics.

🧰 Why Use `pg_stat_activity`?

The `pg_stat_activity` system view captures runtime information about every session connected to the PostgreSQL server. It is one of the most powerful diagnostic tools available to PostgreSQL DBAs and developers.

🔍 What It Exposes:

- **Current database connections** (database name, user name, process ID)
- **Client details** (IP address, application name, port)
- **Running SQL queries** per session
- **Query state and start time**
- **Transaction and backend start times**
- **Wait events and lock statuses**

🧠 Why It Matters:

Using this live view, you can:

- **Detect long-running or resource-intensive queries** before they cause application slowdowns
- **Trace blocked sessions** and investigate lock contention
- **Identify idle-in-transaction sessions** that delay vacuuming and increase table bloat
- **Proactively terminate problematic backends** in high-load scenarios
- **Audit active users and client tools** interacting with the database

> 📘 *Access to `pg_stat_activity` requires superuser privileges or specific role-level permissions (e.g., `pg_monitor`).*

🚦 Key Monitoring Queries Using `pg_stat_activity`

The following queries are designed to give you immediate visibility into various aspects of PostgreSQL runtime behavior. These can be run directly in `psql`, scripted for automation, or integrated into dashboards and alerts.

1️⃣ View Connected Users and Client IPs

Understanding who is connected and from where is the foundation of session management.

```
echo "View the connected users and their clients"
psql -c "
SELECT
  datname,
  usename,
  client_addr,
  client_port
FROM pg_stat_activity;"
```

```
View the connected users and their clients
 datname  | usename  | client_addr | client_port
----------+----------+-------------+-------------
 postgres | postgres |             |          -1
 postgres | postgres |             |          -1
          |          |             |
          | postgres |             |
          |          |             |
          |          |             |
          |          |             |
```

```
(7 rows)

[postgres@ip-172-31-89-173 ~]$
```

🔍 **Explanation:**

- `datname` : The name of the database the user is connected to.
- `usename` : Username used by the client.
- `client_addr` : IP address of the client.
- `client_port` : Port on the client side.

✅ **Use Case:**
Identify all active sessions and validate legitimate access, especially when investigating security events or session spikes.

> 🔒 *Look for unknown IPs or users connected during off-hours as part of routine audits.*

2️⃣ View Currently Active Queries

Quickly inspect which SQL statements are being executed in real time.

```
echo "View active queries"
psql -c "
SELECT
  datname,
  usename,
  query
FROM pg_stat_activity
WHERE state != 'idle';"
```

```
View active queries
 datname  | usename  |                query
```

```
----------+----------+---------------------------------------
 postgres | postgres | INSERT INTO big_table (data)          +
          |          | SELECT repeat('x', 1000)              +
          |          | FROM generate_series(1, 10000000000);
 postgres | postgres |                                       +
          |          | SELECT                                +
          |          |   datname,                            +
          |          |   usename,                            +
          |          |   query                               +
          |          | FROM pg_stat_activity                 +
          |          | WHERE state != 'idle';
(2 rows)

[postgres@ip-172-31-89-173 ~]$
[postgres@ip-172-31-89-173 ~]$
```

🔍 **Explanation:**

- Filters out connections in an idle state.
- Displays only sessions actively executing SQL commands.

✅ **Use Case:**

Live query monitoring during application load testing or peak usage hours to assess system throughput and detect unusual queries.

> 🧩 Use this query to populate lightweight dashboards or auto-refresh terminal views.

3️⃣ Identify Long-Running Queries

Track queries that may be stuck, looping, or consuming excessive resources.

```
echo "Long running queries"
psql -c "
SELECT
  current_timestamp - query_start AS runtime,
  datname,
  usename,
  query
FROM pg_stat_activity
```

```
WHERE state != 'idle'
ORDER BY runtime DESC;"
```

```
Long running queries
 runtime  | datname  | usename  |                        query
----------+----------+----------+-----------------------------------------------
 00:00:00 | postgres | postgres |
          |          |          | SELECT
          |          |          |   current_timestamp - query_start AS runtime,
          |          |          |   datname,
          |          |          |   usename,
          |          |          |   query
          |          |          | FROM pg_stat_activity
          |          |          | WHERE state != 'idle'
          |          |          | ORDER BY runtime DESC;
(1 row)

[postgres@ip-172-31-89-173 ~]$
```

🔍 **Explanation:**

- Calculates the duration each query has been running.
- Sorts sessions with the longest runtimes at the top.

✅ **Use Case:**

Detect and isolate slow-performing queries in batch jobs, user-generated reports, or ad hoc workloads.

> ⚠️ *Establish performance thresholds (e.g., 5–10 minutes) to automatically flag or terminate unusually long queries.*

4️⃣ Detect Blocked Processes and Lock Contention

PostgreSQL uses row-level and table-level locking mechanisms. Unresolved locks can stall entire workflows.

```
echo "Blocked only due to lock waits"
psql -c "
SELECT
  pid,
  backend_type
FROM pg_stat_activity
ORDER BY backend_type, pid;"
```

```
Blocked only due to lock waits
 pid  |         backend_type
------+------------------------------
 1769 | autovacuum launcher
 1766 | background writer
 1765 | checkpointer
 1824 | client backend
 1963 | client backend
 1770 | logical replication launcher
 1768 | walwriter
(7 rows)

[postgres@ip-172-31-89-173 ~]$
```

🔍 **Explanation:**

- Lists session process IDs grouped by backend type (e.g., autovacuum, client backend).
- Helps correlate backend activity with lock types.

✅ **Use Case:**

Identify which backend processes are involved in locking and which queries may be causing congestion or deadlocks.

> 🔗 Combine with `pg_locks` and `pg_blocking_pids()` for a full lock wait graph.

5️⃣ Measure Transaction Durations

Open transactions that are not closed promptly can interfere with vacuum operations and contribute to table bloat.

```
echo "Processes ordered by current txn_duration"
psql -c "
SELECT
  datname,
  pid,
  application_name,
  state,
  query,
  now() - xact_start AS txn_duration
FROM pg_stat_activity
ORDER BY txn_duration DESC;"
```

```
Processes ordered by current txn_duration
 datname  | pid  | application_name | state  |                 query
----------+------+------------------+--------+----------------------------------
          | 1770 |                  |        |
          | 1766 |                  |        |
          | 1768 |                  |        |
          | 1769 |                  |        |
          | 1765 |                  |        |
 postgres | 1824 | psql             | active | INSERT INTO big_table (data)
          |      |                  |        | SELECT repeat('x', 1000)
          |      |                  |        | FROM generate_series(1, 1000000)
 postgres | 1968 | psql             | active |
          |      |                  |        | SELECT
          |      |                  |        |   datname,
          |      |                  |        |   pid,
          |      |                  |        |   application_name,
          |      |                  |        |   state,
          |      |                  |        |   query,
          |      |                  |        |   now() - xact_start AS txn_dura
          |      |                  |        | FROM pg_stat_activity
          |      |                  |        | ORDER BY txn_duration DESC;
(7 rows)
```

```
[postgres@ip-172-31-89-173 ~]$
```

🔍 **Explanation:**

- Calculates how long each transaction has been active.
- Identifies applications and users associated with those transactions.

✅ **Use Case:**

Detect “idle in transaction” problems and prevent scenarios where autovacuum skips cleanup because of lingering transactions.

> 🧠 *Regularly review long transaction durations to ensure application logic commits or rolls back properly.*

🧪 Sample Automation Integration

Integrate these queries into your system monitoring pipeline:

- **Cron-based reporting:**
- Run and log every 15 minutes.
- Email alerts if query runtime or txn duration exceed thresholds.
- **Prometheus & Grafana Dashboards:**
- Use PostgreSQL exporters with queries tied to visual thresholds.
- **Custom admin views in pgAdmin or DataGrip:**
- Embed queries into user-defined dashboards.

✅ Best Practices When Using `pg_stat_activity`

Practice	Recommendation
Limit result sets	Always filter by `state != 'idle'` to reduce noise
Protect system performance	Avoid full table scans of `pg_stat_activity` under load
Watch for idle in transaction	`state = 'idle in transaction'` should be investigated
Use process IDs (PID) carefully	Only terminate sessions after reviewing potential impact
Pair with `pg_locks` for lock analysis	Combine session and lock views for deadlock diagnosis
Audit user behavior	Track `application_name` and `client_addr` for usage and anomaly detection

🏁 Conclusion

Monitoring PostgreSQL in real time is essential for maintaining performance, ensuring stability, and responding proactively to issues. The `pg_stat_activity` view is a built-in observability tool that provides immediate insight into session-level behavior, enabling you to:

- 🎯 Detect problematic queries before they disrupt service
- 🔒 Identify blocking and waiting sessions during lock contention
- ⏱️ Pinpoint long transactions that delay cleanup or introduce risk
- 👩‍💻 Trace user activity across client applications

By incorporating the diagnostic queries shown above into your **daily monitoring practices**, **automated alerts**, or **real-time dashboards**, you equip yourself with a reliable framework to manage PostgreSQL health at scale.

> 📌 Start small with manual checks, scale up with automation, and integrate deeply with your team's operational workflows.

🛠️ Advanced PostgreSQL Monitoring: Terminating Sessions, Logging Insights, and Performance Simulations

PostgreSQL is an enterprise-class open-source database trusted by thousands of organizations to power transactional workloads. To ensure smooth operations in high-concurrency and high-throughput environments, it's crucial for DBAs and engineers to proactively monitor database activity, manage long-running queries, and configure systems for observability.

This article focuses on **actionable monitoring techniques** beyond passive observation — such as **terminating runaway sessions**, **enabling full logging**, and **generating test datasets for benchmarking**. These are best practices every PostgreSQL administrator should be familiar with.

🔧 Managing Problematic Queries in PostgreSQL

While monitoring tools like `pg_stat_activity` provide visibility into what's happening, you sometimes need to **take corrective action**—especially when queries block others, run longer than expected, or consume excessive resources.

📌 How to Terminate a Session

You can terminate a backend session safely using the `pg_terminate_backend()` function.

```
echo "Terminate the idle transaction. Replace <pid> with the PID of the process
psql -c "SELECT pg_terminate_backend(1824);"
```

> ⚠ Replace `<pid>` with the actual process ID of the session, which you can get from `pg_stat_activity`.

```
psql -c "SELECT pg_terminate_backend(1824);"
Terminate the idle transaction. Replace <pid> with the PID of the process to te
 pg_terminate_backend
----------------------
 t
(1 row)

[postgres@ip-172-31-89-173 ~]$
```

```
postgres=# INSERT INTO big_table (data)
SELECT repeat('x', 1000)
FROM generate_series(1, 1000000);
INSERT 0 1000000
postgres=#
postgres=#
postgres=# select count(*) big_table;
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
postgres=#
postgres=#
```

🔍 When to Use It:

- A query is stuck in a join, loop, or cartesian explosion
- An application holds a lock on a critical table and won't release it
- Sessions are idle in transaction, preventing autovacuum from running
- A rogue analytics dashboard launches resource-intensive queries

✅ Best Practices:

- Always inspect the query and session metadata first (e.g., user, query text, backend start time)
- If possible, notify the user or log the termination for auditing
- Avoid terminating autovacuum workers unless absolutely necessary

🖥️ Configuring PostgreSQL for Deeper Monitoring and Logging

A default PostgreSQL installation does not include verbose logs or detailed query tracking. As part of **observability hardening**, DBAs should configure the following settings:

📝 Enable Full Logging for Session Context

PostgreSQL logs can be customized using `log_line_prefix` to add context like timestamps, users, client IPs, and application names. This is useful for **debugging, auditing, and post-incident forensics.**

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

Log Breakdown:

- `%t` = Timestamp
- `%p` = Process ID
- `%u` = User
- `%d` = Database
- `%a` = Application name
- `%h` = Client hostname or IP

> 💡 After applying this change, reload the configuration with `SELECT pg_reload_conf();` or restart the PostgreSQL service.

✅ Benefits:

- Associate slow queries with specific users or applications
- Identify IP addresses generating connection floods
- Provide context-rich logs for security and compliance

📊 Enable `pg_stat_statements` for Historical Query Analytics

The `pg_stat_activity` view provides real-time insight, but it does not store **historical query statistics**. For that, you can enable the `pg_stat_statements` extension.

🔌 Installation & Setup:

```
psql -c "ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';"
sudo systemctl restart postgresql-17
psql -c "CREATE EXTENSION pg_stat_statements;"
```

```
[postgres@ip-172-31-89-173 ~]$ psql -c "ALTER SYSTEM SET shared_preload_librari
ALTER SYSTEM
[postgres@ip-172-31-89-173 ~]$
[postgres@ip-172-31-89-173 ~]$ sudo systemctl restart postgresql-17
[postgres@ip-172-31-89-173 ~]$
[postgres@ip-172-31-89-173 ~]$ psql -c "CREATE EXTENSION pg_stat_statements;"
CREATE EXTENSION
[postgres@ip-172-31-89-173 ~]$
```

🔍 Query Example:

```
SELECT query, calls, total_exec_time, mean_exec_time
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

```
                                query                                 | calls |  tc
----------------------------------------------------------------------+-------+----
 SELECT COUNT(*) FROM big_table                                      +|     1 |
 WHERE data LIKE $1                                                   |       |
 SELECT COUNT(*) FROM big_table                                       |     1 |
 CREATE EXTENSION pg_stat_statements                                  |     1 |
 SELECT query, calls, total_exec_time, mean_exec_time                +|    12 | 4.2
 FROM pg_stat_statements                                             +|       |
 ORDER BY total_exec_time DESC                                       +|       |
 LIMIT $1                                                             |       |
 ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements'     |     1 |
(5 rows)

postgres=#
```

✅ Use Cases:

- Identify **frequently executed queries**
- Detect **queries with the highest total or mean execution time**
- Perform **query normalization and optimization**
- Understand **query patterns across multiple users or applications**

> 🧠 Best Practice: Periodically reset statistics during maintenance windows using `SELECT pg_stat_statements_reset();` to keep the view manageable.

🧪 Test Dataset for Performance Simulation

In development or testing environments, you'll often need to simulate real-world data volumes to validate indexing, tuning, or application behavior under load.

🧱 Generate 1 Million Rows Using `generate_series()`

PostgreSQL's built-in `generate_series()` function makes it easy to create synthetic datasets.

```
psql -h localhost -U pguser -d pgdb \
  -c "CREATE TABLE products AS SELECT * FROM generate_series(1, 1000000);"
```

✅ Purpose:

- Simulate **large datasets** for benchmarking
- Test **index creation** and performance
- Analyze **query planner behavior**
- Validate **data migration tooling**

📝 Tip:

To better mimic real-world schemas, wrap `generate_series()` with additional columns, such as timestamps, status fields, or JSON payloads.

📈 Simulating Read and Write Workloads

Testing how PostgreSQL behaves under load is a critical part of performance validation. Once you've created test data, perform the following operations:

🔍 Count Rows (Read Test)

```
psql -h localhost -U pguser -d pgdb \
  -c "SELECT COUNT(*) FROM products;"
```

✅ Use Case:

- Monitor how PostgreSQL utilizes indexes or sequential scans
- Validate caching effectiveness (repeated queries)
- Observe I/O and CPU behavior during reporting scenarios

➕ Insert New Rows (Write Test)

```
psql -h localhost -U pguser -d pgdb \
  -c "INSERT INTO products SELECT * FROM generate_series(1000001, 1200001);"
```

✅ Use Case:

- Stress test write throughput
- Observe behavior of WAL (Write-Ahead Logging)
- Validate performance of partitioned tables or bulk ingestion patterns

> 📘 Wrap in a transaction if needed and use `EXPLAIN ANALYZE` for deeper insight into execution plans.

🧠 Summary of Monitoring Actions

Task	Action	Tool / Query
Terminate bad sessions	`SELECT pg_terminate_backend(<pid>);`	Manual
Add logging metadata	Set `log_line_prefix` for detailed context	`ALTER SYSTEM`
Enable query tracking	Enable `pg_stat_statements` extension	Extension
Generate test data	Use `generate_series()` to create millions of rows	SQL
Simulate workloads	Run read/write SQL for performance testing	SQL

🏁 Conclusion

PostgreSQL monitoring isn't just about dashboards — it's about having the ability to **act decisively**, **log intelligently**, and **simulate workloads** that reflect production complexity. By incorporating the techniques shared in this guide, you can:

- 🛠️ Terminate sessions that threaten database stability
- 📋 Improve log clarity and postmortem analysis
- 📊 Gain visibility into normalized, historical query patterns
- 🔬 Perform realistic performance tests and tuning exercises

These are **core competencies** for PostgreSQL DBAs, Site Reliability Engineers, and performance-conscious developers.

> Start applying these monitoring techniques today to proactively manage your PostgreSQL workloads with precision and confidence.

🔍 Putting It All Together: Real-Time PostgreSQL Monitoring with `pg_stat_activity`

PostgreSQL is a powerful open-source relational database that supports a wide range of workloads. One of its most valuable features for operational monitoring is the `pg_stat_activity` view—a real-time window into what is happening inside your database. When combined with diagnostic scripts and structured practices, it becomes an essential part of a DBA's performance and stability toolkit.

This section summarizes the core benefits and operational outcomes of using `pg_stat_activity` and related utilities for efficient database health checks.

🧰 Why It Matters: End-to-End Monitoring Workflow

With the right set of queries and scripts, PostgreSQL administrators and performance engineers can gain actionable insights and maintain high database uptime. Specifically, the capabilities outlined allow you to:

✅ Monitor Real-Time SQL Activity

You can continuously observe how your database is being used by tracking active and idle sessions. This visibility allows administrators to confirm whether PostgreSQL is handling expected workloads correctly or if there are anomalies requiring further investigation.

✅ Investigate Slow or Blocked Queries

Long-running or blocked queries can severely affect throughput and user experience. Having visibility into these issues lets you drill down into the cause — whether it's inefficient SQL, missing indexes, or session-level locks — and enables quicker remediation.

✅ Identify Sessions to Terminate Safely

Not all sessions are equal. Some may be idle in transaction, consuming locks or preventing autovacuum from cleaning up dead tuples. Others may be generating excessive load or holding up business-critical processes. With sufficient monitoring data, problematic sessions can be identified and safely terminated without impacting the broader system.

✅ Simulate Load and Test Performance Tuning

Query performance and database throughput should not only be observed in production but also validated in staging or dev environments. By simulating large

datasets and transactional behavior, teams can test indexing strategies, validate query plans, and ensure tuning changes are effective before rolling them out.

✅ Build Automation Around PostgreSQL Health Checks

Once reliable patterns are identified, they can be automated. Scheduled health check scripts, alerting mechanisms, and dashboards can help detect anomalies earlier — freeing up engineers from constant manual monitoring and reducing time-to-resolution for incidents.

🎯 Final Thoughts

The `pg_stat_activity` view in PostgreSQL remains one of the most **underutilized but powerful tools** available to DBAs.

✅ Real-Time + Historical = Complete Observability

By combining `pg_stat_activity` with tools like `pg_stat_statements` and enhanced logging, teams can move beyond ad-hoc monitoring and achieve a full-stack observability model. This allows for both real-time diagnostics and historical performance analysis.

✅ Proactive Monitoring Prevents Downtime

Waiting until a performance issue causes user impact is already too late. Teams that proactively monitor query behavior — before it becomes a bottleneck — are better equipped to maintain application availability, user satisfaction, and business continuity.

🔗 Let's Connect!

If you enjoyed this post or would like to connect professionally, feel free to reach out to me on LinkedIn:
👉 Jeyaram Ayyalusamy

I regularly share content on **PostgreSQL, database administration, cloud technologies, and data engineering.** Always happy to connect, collaborate, and discuss ideas!

Postgresql AWS Open Source Oracle MySQL

J

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

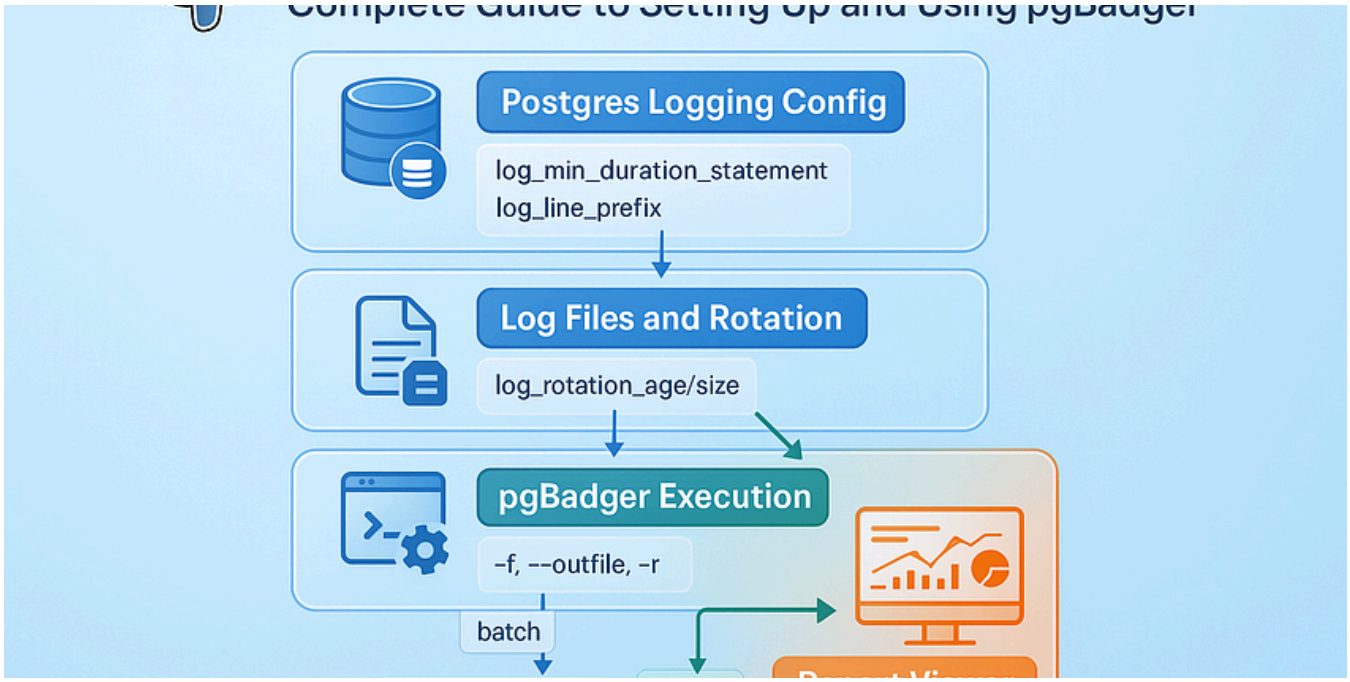
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

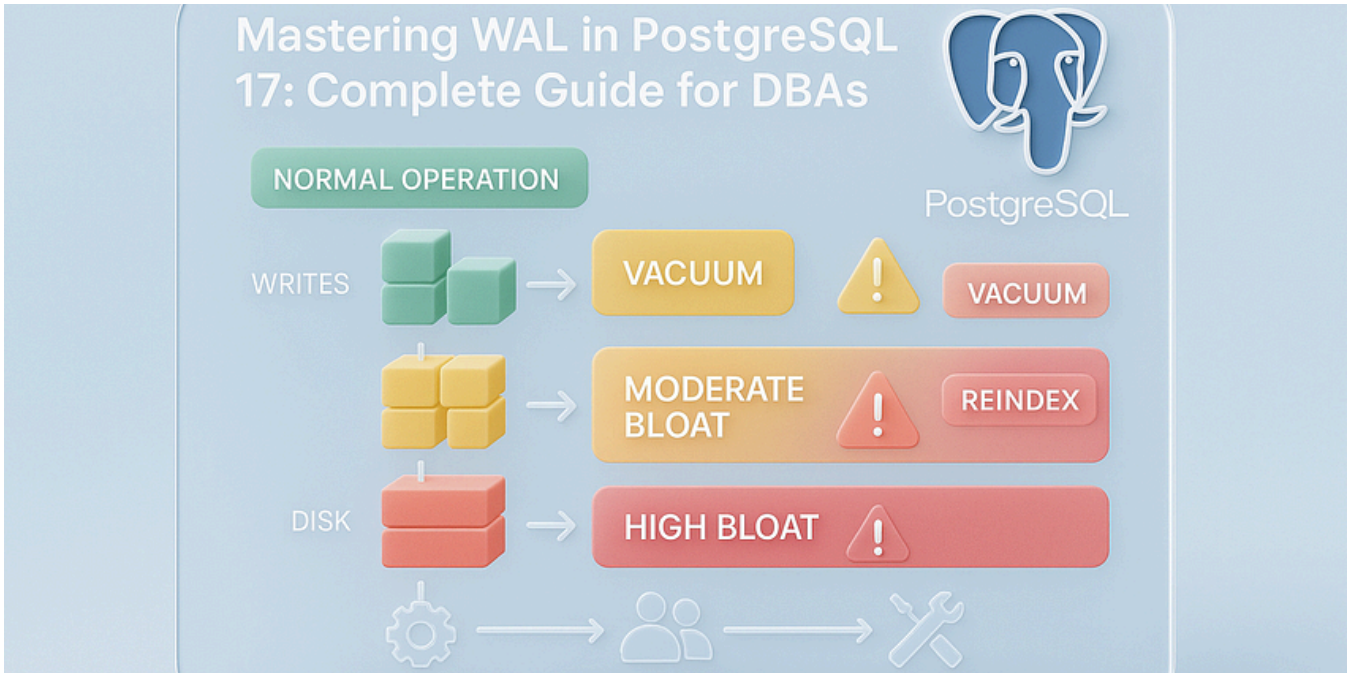


Jeyaram Ayyalusamy

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

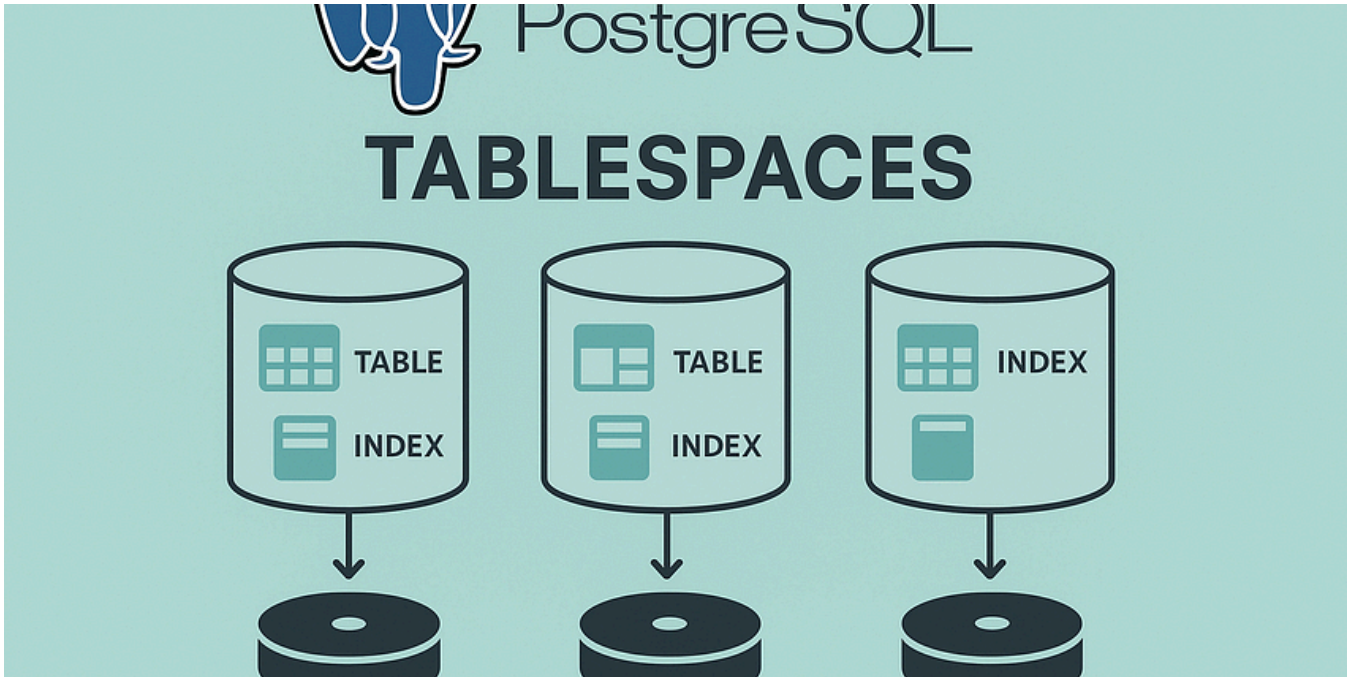


Jeyaram Ayyalusamy

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

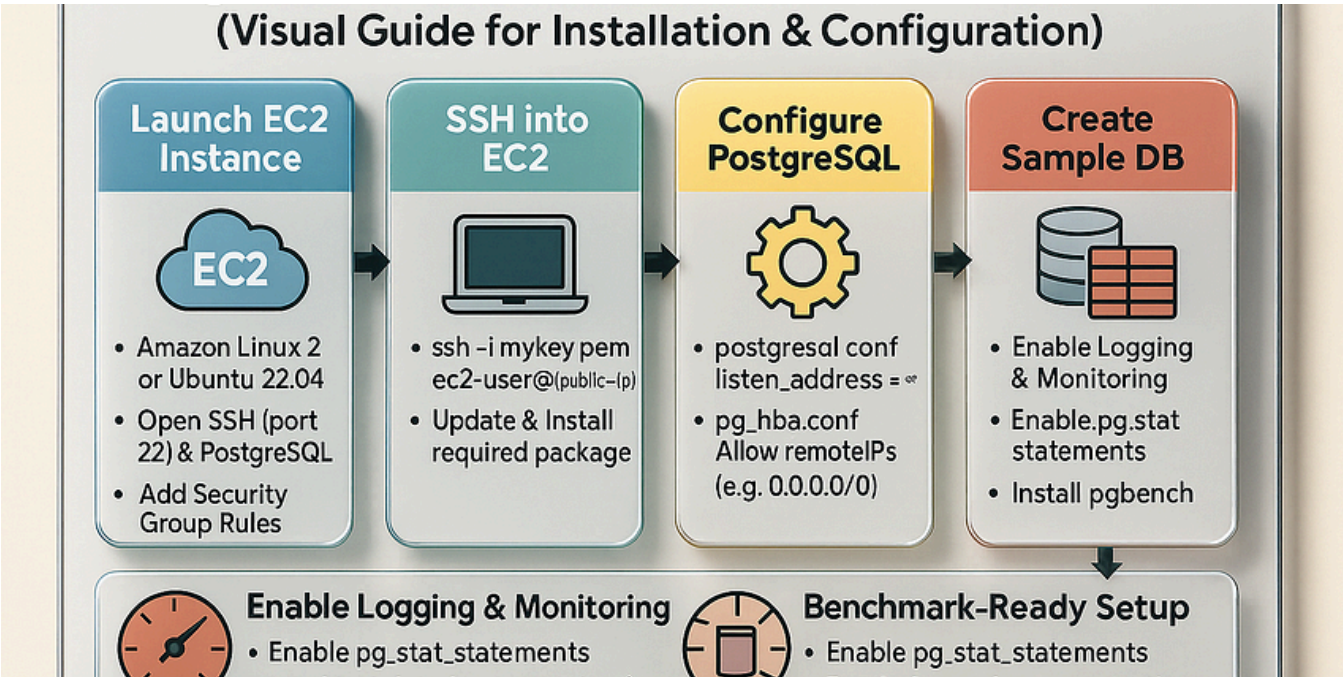


Jeyaram Ayyalusamy

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

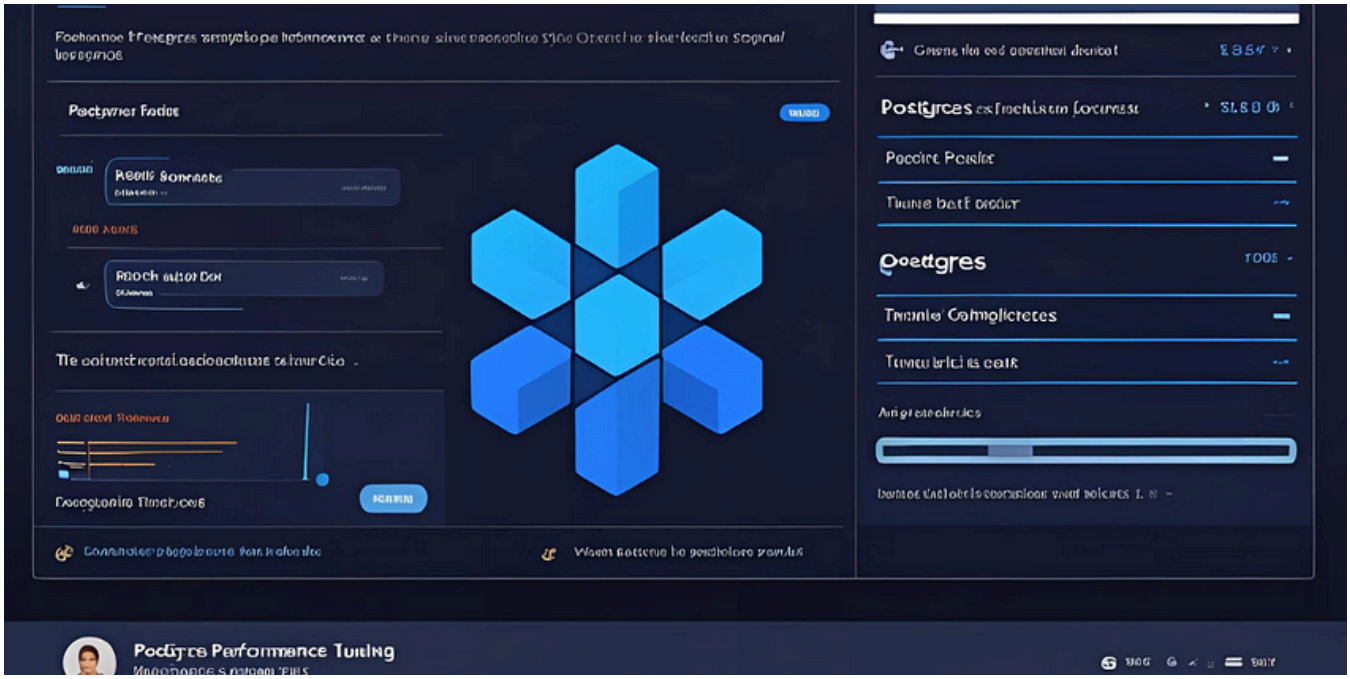
Recommended from Medium

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



Rizqi Mulki

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

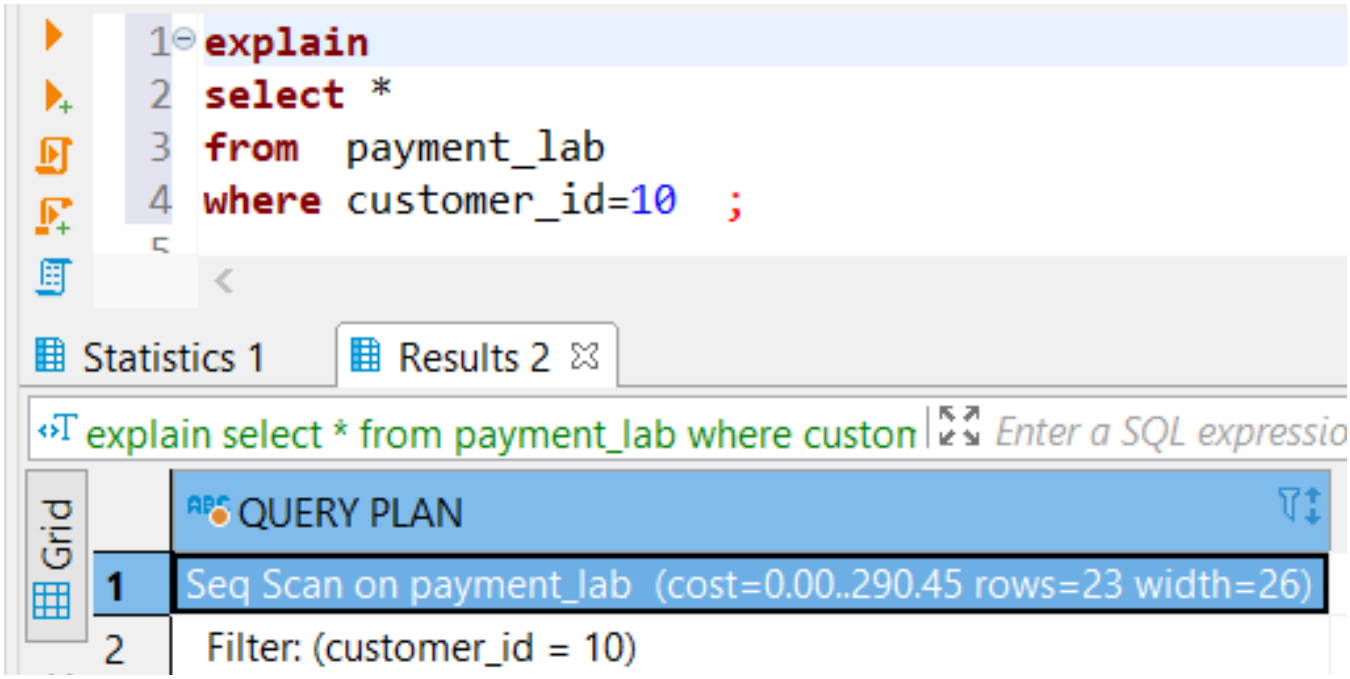
6d ago 55

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



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

Harishsingh

PostgreSQL 18 in Microservices: You Don’t Need a Separate DB for Everything

Introduction: The Myth of Database-Per-Service

Jul 13 11 1

techWithNeeru

This SQL Query Took 20 Seconds. Now It Takes 20ms. Here's What I Changed

Last week, I was troubleshooting a performance issue that was driving our team crazy. Our analytics dashboard was timing out, users were...

Jul 10 66