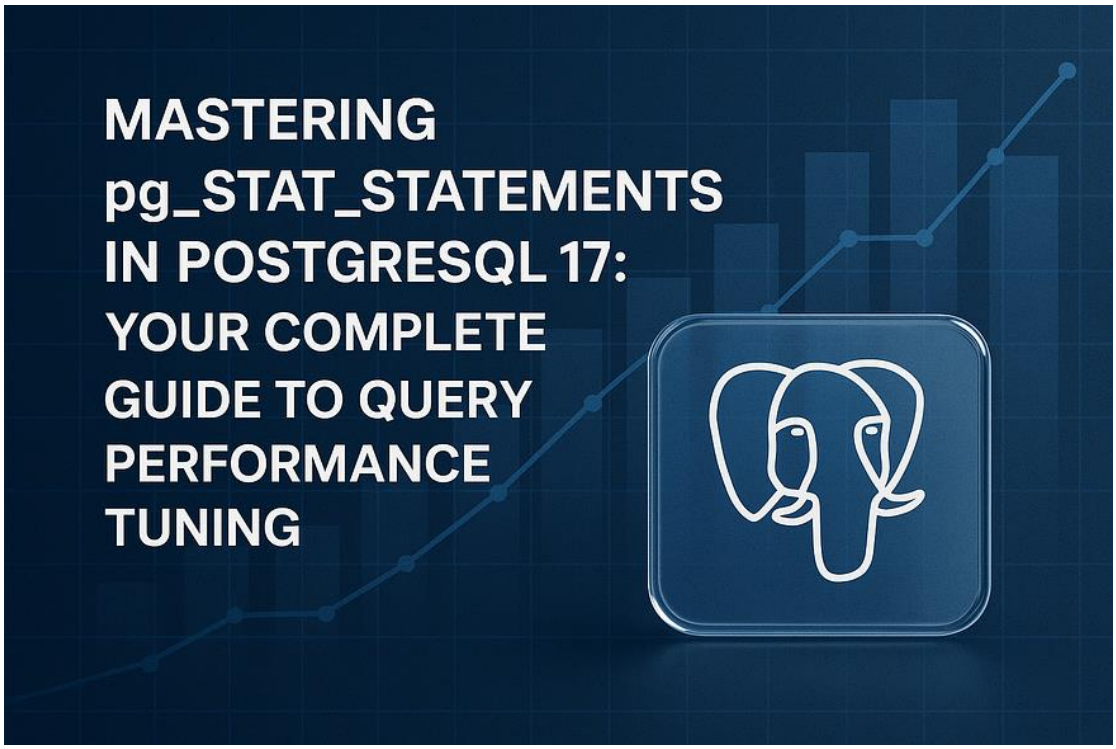
Mastering `pg_stat_statements` in PostgreSQL 17: Your Complete Guide to Query Performance Tuning



PostgreSQL 17 gives database administrators one of the most powerful performance monitoring tools available: `pg_stat_statements`.

If you're serious about PostgreSQL query optimization, workload analysis, or simply diagnosing performance issues, you need to understand how to enable, configure, and leverage `pg_stat_statements` to its full potential.

Let's deep dive. ☐

☐ What Is `pg_stat_statements` in PostgreSQL?

When it comes to performance tuning in PostgreSQL, **understanding how your queries behave** is half the battle. While PostgreSQL logs can give you raw data, what you really need is an organized, actionable overview of query performance across your database.

This is where the powerful extension `pg_stat_statements` comes in.

□ What Is `pg_stat_statements`?

`pg_stat_statements` is an **official PostgreSQL extension** that tracks execution statistics for all SQL statements run on the server. Unlike traditional logging, it **aggregates and normalizes** SQL queries—giving you a high-level, query-centric view of what's really happening in your system.

It's like having an intelligent dashboard that tells you:

- What's slow
- What's frequent
- What's consuming your resources

And it does this without flooding your logs or requiring heavy instrumentation.

✅What Can `pg_stat_statements` Tell You?

Once enabled, `pg_stat_statements` helps answer critical questions such as:

□ Which queries are consuming the most execution time?

By sorting queries based on **total execution time**, you can quickly find bottlenecks and long-running queries — even if they happen infrequently.

□ Which queries are executed most frequently?

If you're experiencing CPU or memory pressure, the culprit might not be one heavy query — but rather **thousands of small queries** executed repeatedly. `pg_stat_statements` helps you detect these patterns instantly.

🔍 Where are your biggest I/O or memory bottlenecks?

The extension provides details like:

- **Shared block reads/writes**
- **Temporary block usage**
- **I/O timing metrics**

This makes it a powerful tool for diagnosing storage-related slowdowns.

🧠 Understanding Query Normalization and Using `pg_stat_statements` in PostgreSQL

In PostgreSQL, performance tuning often starts with a single question: **"Which queries are slowing down my system?"**

That's exactly what the `pg_stat_statements` extension helps answer. But one of its most powerful features—often overlooked—is **query normalization**.

Let's break down how it works, how to enable it, and why it should be part of every PostgreSQL admin's toolbox.

🔍 How It Works: Query Normalization

One of the key advantages of `pg_stat_statements` is that it **normalizes SQL queries before logging them**.

🧪 What does that mean?

Suppose your application runs these queries:

```
SELECT * FROM users WHERE id = 42;
SELECT * FROM users WHERE id = 87;
```

In traditional logging, these would be recorded as **two different statements**, even though structurally they're the same.

But `pg_stat_statements` records a **normalized version** like:

```
SELECT * FROM users WHERE id = $1;
```

This version **strips out literal values** — such as integers, strings, or timestamps — and replaces them with placeholders.

✅Why it matters:

- **Groups similar queries together**
  Avoids clutter from variations of the same query structure with different values.
- **Accurate performance analysis**
  Helps you find the *real* top-consuming queries, not just one-off events.
- **Actionable optimization**
  With normalized patterns, you can target indexes, query rewrites, and caching strategies more effectively.

⚙️ How to Enable `pg_stat_statements`

Ready to unlock its power? Here's how to safely enable and start using it.

Step 1: Modify `postgresql.conf`

Edit the PostgreSQL config file and add:

```
shared_preload_libraries = 'pg_stat_statements'
```

This line tells PostgreSQL to load the extension when the server starts.

> ☐ **Note**: `shared_preload_libraries` requires a full PostgreSQL restart.

Step 2: Restart PostgreSQL

After saving your configuration changes, restart the PostgreSQL service:

```
sudo systemctl restart postgresql
```

Step 3: Create the Extension

Once the server is back up, connect to your database and run:

```
CREATE EXTENSION pg_stat_statements;
```

This registers the extension and starts collecting query statistics.

Step 4: Query the View

You can now access detailed metrics with:

```
SELECT *
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

This query shows the **top 10 most time-consuming SQL statements** — a goldmine for performance diagnostics.

□ Final Thoughts

`pg_stat_statements` is one of the most effective and lightweight tools available for PostgreSQL performance tuning.

By **normalizing queries**, it gives you clear, high-level insights into which operations are truly impacting your database — without drowning in repetitive details.

> If you're managing a PostgreSQL system without `pg_stat_statements`, you're essentially **tuning in the dark**.

Enable it. Use it. Optimize smartly.

□ Prerequisites: How to Enable `pg_stat_statements` in PostgreSQL

The `pg_stat_statements` extension is one of PostgreSQL's most powerful tools for performance monitoring—but it's **not enabled by default**. Before you can start analyzing query execution metrics, there are a few steps you need to complete to activate it.

This guide walks you through the **3 essential steps** to enable `pg_stat_statements` and get your PostgreSQL server ready for advanced performance tuning.

✅Overview: What You Need to Do

To enable `pg_stat_statements`, you must:

1. Modify the `postgresql.conf` file to preload the extension
2. Restart the PostgreSQL service so the changes take effect

3. Create the extension inside your database

Let's walk through each of these steps in detail.

1 Add `pg_stat_statements` to `shared_preload_libraries`

First, open your PostgreSQL configuration file:

```
sudo vi /var/lib/pgsql/17/data/postgresql.conf
```

Then find the line that starts with:

```
#shared_preload_libraries = ''
```

Uncomment and modify it to include:

```
shared_preload_libraries = 'pg_stat_statements'
```

> **Why is this required?**
> Because `pg_stat_statements` uses **shared memory**, and PostgreSQL only loads shared memory libraries during **startup**. This setting ensures the extension is preloaded into memory each time the server starts.

2 Restart PostgreSQL to Apply Changes

Once you've saved your changes to the config file, restart the PostgreSQL service so it can load the new library:

```
sudo systemctl restart postgresql-13
```

☐ This restart is mandatory. Without it, the extension won't be loaded, and the next step will fail.

3☐ Create the Extension in Your Database

Now that the server is preloading `pg_stat_statements`, the final step is to **register the extension** in your database.

Connect to PostgreSQL:

```
psql -U postgres
```

Then run:

```
CREATE EXTENSION pg_stat_statements;
```

If successful, this command tells PostgreSQL to begin collecting and storing normalized query execution statistics.

☐ Success! You're Now Ready to Monitor Query Performance

Once the extension is enabled, you can start running queries like:

```
SELECT *
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

This gives you immediate visibility into the most expensive queries on your system — based on actual execution time and frequency.

□ Final Thoughts

Enabling `pg_stat_statements` is a simple but essential setup task for any PostgreSQL administrator who wants to gain insight into query performance.

> Think of it as flipping on the lights in a dark room:
> suddenly, you can **see exactly where your database is spending its time**, which queries need tuning, and how to optimize your workload.

Once enabled, this extension becomes your go-to performance diagnostic tool — especially in production environments.

□ Fine-Tuning and Analyzing Performance with `pg_stat_statements` in PostgreSQL

Once you've enabled the powerful `pg_stat_statements` extension in PostgreSQL, you gain access to rich query performance metrics. But to get the most value from it, it's important to know **how to tune its behavior** and **how to analyze the data it provides**.

Let's walk through the **core parameters** you can configure and the **SQL queries** you can use to extract real insights from your workload.

□ Core Parameters to Tune in `postgresql.conf`

The behavior of `pg_stat_statements` can be customized through several key parameters in your `postgresql.conf` file. These settings allow you to control how much data is collected, which queries are tracked, and whether statistics persist across restarts.

□ Key Parameters:

Parameter Purpose Default `pg_stat_statements.max` Maximum number of **distinct statements** to track `5000` `pg_stat_statements.track` What types of statements to track (`all`, `top`, `none`) `top` `pg_stat_statements.track_utility` Whether to include **utility statements** (e.g., DDL) `on` `pg_stat_statements.save` Whether to **retain stats after restart** `on`

□ Parameter Breakdown:

- **`pg_stat_statements.max`**:
  This determines how many unique normalized query entries PostgreSQL will store. If your application runs many distinct queries, you may need to increase this value.

- **`pg_stat_statements.track`**:
  Controls the scope of what gets tracked:

- `all` – track all queries (including nested and function calls)

- `top` – track only top-level queries (default, and most commonly used)

- `none` – disables tracking

- **`pg_stat_statements.track_utility`**:
  When enabled, this tracks **utility commands** such as `CREATE`, `DROP`, and `VACUUM`, which can be helpful for identifying non-query overhead.

- **`pg_stat_statements.save`**:
  If set to `on`, query statistics are saved across restarts, allowing for long-term trend analysis without data loss.

□ Analyzing SQL Performance with `pg_stat_statements`

Once the extension is up and running, you can start using SQL queries to explore your workload. Here are some of the most useful ways to interpret the data.

□ Top Queries by Total Execution Time

Want to know which queries are consuming the most resources overall? Start here:

```
SELECT
  query,
  calls,
  total_time,
  rows,
  mean_time,
  stddev_time
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

□ What These Columns Mean:

- `query` – the normalized SQL text
- `calls` – how many times the query has been executed
- `total_time` – total time (in milliseconds) spent running the query
- `mean_time` – average time per execution
- `stddev_time` – variation in execution time (useful for spotting inconsistent performance)

This query gives you a high-level view of **which queries are using the most time overall**, even if they're not individually slow.

□ Top Queries by Average Runtime

To identify the **most expensive queries per execution**, use this query:

```
SELECT
  (total_time / 1000 / 60) AS total_minutes,
  (total_time / calls) AS avg_time_per_call,
  query
FROM pg_stat_statements
```

```
ORDER BY avg_time_per_call DESC
LIMIT 100;
```

□ Why It's Useful:

- Pinpoints queries that are **slow each time they run**, even if they don't run often
- Helps you catch inefficient queries, large table scans, or poor index usage
- A great starting point for optimization and query rewriting

□ Final Thoughts

The true power of `pg_stat_statements` lies not just in enabling it—but in **customizing what it tracks** and knowing **how to interpret the results**.

By adjusting the core parameters and using thoughtful analysis queries, you gain deep visibility into your PostgreSQL workload — allowing for **precise tuning, better indexing strategies, and improved overall performance**.

> ☐ Pro Tip: Combine `pg_stat_statements` with visual tools like **pgBadger** or integrate it into **Grafana dashboards** for even richer insights.

□ Deeper Analysis with Advanced Queries in `pg_stat_statements`

Once you've enabled the `pg_stat_statements` extension and started collecting query performance data, the next step is to **go beyond the basics**. While identifying top queries by execution time or frequency is useful, deeper analysis can reveal **hidden performance bottlenecks**—especially around disk I/O.

In this article, we'll explore two advanced SQL queries that help you identify:

- The most I/O-intensive queries
- The queries with the highest total execution time

These insights are crucial when optimizing large, high-throughput PostgreSQL environments.

□ Top 10 I/O-Intensive Queries

When your PostgreSQL database starts to feel sluggish, **disk I/O** is often a major culprit. PostgreSQL tracks two important metrics:

- `blk_read_time` – time spent reading data blocks from disk
- `blk_write_time` – time spent writing data blocks to disk

These metrics help you pinpoint queries that are generating the most **physical I/O load**, which often correlates with slow performance, especially on systems without fast SSDs or sufficient caching.

□ Query to Run:

```
SELECT
  userid::regrole,
  dbid,
  query
FROM pg_stat_statements
ORDER BY (blk_read_time + blk_write_time) / calls DESC
LIMIT 10;
```

□ What It Shows:

- `userid::regrole` – the role (username) that executed the query
- `dbid` – the internal database ID where the query ran

- `query` – the normalized SQL text
- The **ORDER BY clause** ranks queries by their **average I/O time per call**

This query helps identify **expensive queries at the storage layer** — even if they aren't the slowest in total time.

> ✅ Use this to optimize indexes, reduce full table scans, or re-architect high-I/O transactions.

🔹 Top 10 Queries by Total Execution Time

To find the queries consuming the most total time — regardless of how fast they are per execution — you can look at **aggregate execution time**.

This helps surface:

- Frequently used queries that contribute to overall system load
- Candidates for caching or denormalization
- Hotspots in high-volume workloads

🔹 Query to Run:

```
SELECT
  userid::regrole,
  dbid,
  query
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

> ☐ `total_exec_time` is the total time (in milliseconds) spent executing each normalized query pattern since stats were last reset.

□ Why These Queries Matter

- ☐ **Performance isn't just about individual query speed** — it's also about total impact over time.
- ☐ These advanced queries help you identify **systemic inefficiencies**, not just outliers.
- ☐ They provide a solid foundation for making **data-driven decisions** about indexing, query rewriting, caching, or workload balancing.

✅Final Takeaway

With these deeper insights from `pg_stat_statements`, you can move from reactive query troubleshooting to **proactive database optimization**.

> If you're relying only on execution time or query frequency — you might be missing the real performance drains hiding in I/O-heavy or high-volume operations.

□ Top 10 by Query Variability (Unstable Query Times)

Some queries behave unpredictably. They run fast most of the time, but occasionally take much longer — leading to **intermittent performance spikes** that are hard to diagnose.

To detect these problematic queries, sort by the **standard deviation of execution time**, which measures how much a query's execution time varies.

□ Query:

```
SELECT userid::regrole, dbid, query
FROM pg stat statements
ORDER BY stddev exec time DESC
LIMIT 10;
```

□ Why It Matters:

- High variability often indicates **locking issues**, **parameter-sensitive plans**, or **external dependencies** (like waiting for I/O).
- Stable systems run stable queries. Identifying and optimizing unstable queries can improve overall **predictability and performance**.

□ Top 10 Memory-Intensive Queries

Memory consumption is another key factor in database performance. Queries that access or modify a lot of data in memory can impact cache efficiency and increase pressure on shared buffers.

You can detect **memory-heavy queries** by looking at two important counters:

- `shared_blks_hit`: blocks read from PostgreSQL's shared memory (cache)
- `shared_blks_dirtied`: blocks modified and marked as dirty in shared memory

Combining these gives a rough estimate of how intensively a query interacts with the buffer pool.

□ Query:

```
SELECT userid::regrole, dbid, query
FROM pg_stat_statements
ORDER BY (shared_blks_hit + shared_blks_dirtied) DESC
LIMIT 10;
```

□ Why It Matters:

- High values here suggest that queries are **caching large amounts of data** or making many changes to memory-resident blocks.

- These queries may benefit from **index tuning**, **batching logic**, or **reducing result set size**.

□ Top 10 Temp Space Consumers

When PostgreSQL can't fit intermediate results (like for sorts or hashes) into memory, it spills them to **temporary disk space**, which is much slower.

These spills are tracked using `temp_blks_written`, and high values can reveal queries that are:

- Sorting large datasets
- Joining large tables
- Using more memory than allocated (e.g., under-tuned `work_mem`)

□ Query:

```
SELECT userid::regrole, dbid, query
FROM pg stat statements
ORDER BY temp_blks_written DESC
LIMIT 10;
```

□ Why It Matters:

- Temp space usage is a common **silent performance killer**, especially under concurrent workloads.
- You can reduce temp writes by increasing `work_mem`, rewriting queries, or breaking complex operations into smaller steps.

□ Final Thoughts

PostgreSQL performance tuning isn't just about which queries are slow — it's about **why** they're slow, and **how** they affect your system at a deeper level.

By using these advanced `pg_stat_statements` diagnostics:

- ☐ You uncover hidden inefficiencies in memory and temp space usage.
- ☐ You identify queries with inconsistent runtimes that disrupt user experience.
- ☐ You gain the visibility needed to fine-tune your database for reliability, scalability, and speed.

> Don't just look at the surface — **dig deeper to optimize smarter**.

□ Resetting Statistics and Configuring Remote Access for `pg_stat_statements` in PostgreSQL

Once you've started using the `pg_stat_statements` extension to analyze query performance, there will be times when you want to **reset the collected statistics**—for example, after major query optimization, schema changes, or benchmarking. Additionally, if you want to monitor this data using **remote tools** like pgAdmin or custom dashboards, you'll need to enable external access to your PostgreSQL server.

Let's walk through both tasks step by step.

□ Reset Statistics (Clear History)

By default, `pg_stat_statements` continuously accumulates query performance data. While this is useful for long-term analysis, there are scenarios where you may want to **start fresh**:

- After deploying major performance improvements
- Before conducting a controlled benchmark
- To remove outdated or irrelevant data

□ To reset all statistics:

```
SELECT pg_stat_statements_reset();
```

□ Important Notes:

- Only **superusers** (such as `postgres`) can run this command.
- This action **clears all accumulated query statistics** — it cannot be undone.
- You can run it inside `psql`, pgAdmin, or any connected SQL client.

> ☐ Tip: Consider scripting periodic resets in non-production environments to keep the performance view clean during development.

□ Configure `pg_hba.conf` for Remote Access (Optional)

If you're managing your PostgreSQL server from a remote location or wish to integrate `pg_stat_statements` with **external monitoring tools** (like pgAdmin, Zabbix, or Grafana), you'll need to **allow remote connections**.

This is controlled via the PostgreSQL **Host-Based Authentication file**, `pg_hba.conf`.

✓Step-by-Step: Enable Remote Access

1□ Open the `pg_hba.conf` file:

```
sudo vi /var/lib/pgsql/13/data/pg_hba.conf
```

This file determines **who can connect**, **from where**, and **using what method**.

2 Add a remote access rule:

```
host    all    all    172.31.40.56/24    md5
```

Here's what each part means:

- `host`: This rule applies to TCP/IP connections
- `all`: Applies to all databases
- `all`: Applies to all users
- `172.31.40.56/24`: The IP range allowed to connect (adjust to match your environment)
- `md5`: Use password authentication (you can also use `scram-sha-256` if supported)

> ☐ Be careful to **only allow trusted IP ranges**, and always use strong passwords or certificate-based authentication.

3 Reload PostgreSQL to apply the changes:

```
sudo systemctl restart postgresql-13
```

Alternatively, for a safer (non-disruptive) option:

```
sudo systemctl reload postgresql-13
```

□ Final Thoughts

- Use `pg_stat_statements_reset()` sparingly and only when you have a clear reason to start fresh.
- Enabling remote access to PostgreSQL can unlock powerful monitoring tools — but must be done **securely** and **responsibly**.
- Always document IP ranges and audit who has access, especially in production.

> Whether you're benchmarking new queries or scaling your monitoring infrastructure, these small configuration steps give you better control and insight into your PostgreSQL environment.

□ Bonus Tuning for Better Metrics + Why `pg_stat_statements` Is a Must-Have for Every PostgreSQL DBA

As you fine-tune your PostgreSQL performance monitoring setup, it's worth going a step further. PostgreSQL offers a few additional parameters that can **enhance the quality of data** captured by `pg_stat_statements`—giving you deeper insights into I/O, query text, and resource usage.

Let's take a look at these **bonus settings**, and wrap up with why `pg_stat_statements` should be a default part of every serious PostgreSQL deployment.

□ Bonus: Additional Parameters for Better Metrics

While `pg_stat_statements` is powerful out of the box, enabling a few additional settings can make your diagnostics even more precise.

☐ `track_io_timing`

```
track_io_timing = on
```

This setting enables PostgreSQL to collect **timing data for I/O operations**, including:

- How long it takes to read data blocks from disk
- How much time is spent writing dirty pages

> ☐ Why it matters:
> With this enabled, you'll get access to `blk_read_time` and `blk_write_time` in `pg_stat_statements`, helping you identify **I/O bottlenecks** that might otherwise go unnoticed.

☐ `track_activity_query_size`

```
track_activity_query_size = 2048
```

This parameter defines how much of the currently running SQL query text PostgreSQL stores in memory. The default is often quite low (e.g., 1024 bytes), which may truncate complex queries.

Increasing it ensures you can **see the full query text**, which is especially useful for debugging or monitoring long-running analytics or dynamically generated queries.

> ☐ Pro Tip: Set this high enough to capture full query payloads, but avoid excessive values that could bloat memory usage unnecessarily.

□ Why `pg_stat_statements` Is a Game-Changer

PostgreSQL has come a long way in terms of observability — and `pg_stat_statements` sits at the **heart of modern performance tuning**. Here's why every DBA should make it part of their default toolkit:

✅Eliminates guesswork during query tuning

Instead of sifting through logs or relying on anecdotal evidence, you get **real, aggregated performance data**.

✅Quickly identifies worst-performing queries

Find long-running queries, I/O-heavy transactions, or frequent offenders with just a few lines of SQL.

✅Enables passive monitoring without verbose logging

Unlike detailed log-based monitoring, `pg_stat_statements` collects stats **silently and efficiently**, without bloating your log files.

✅Built for production environments

It's light on overhead and robust enough to handle real-time insights in mission-critical systems.

✅Integrates seamlessly with monitoring tools

Tools like **pgAdmin**, **pgBadger**, and **Prometheus exporters** can pull directly from `pg_stat_statements`, powering powerful dashboards and alerts.

□ Pro DBA Tip

When analyzing query performance, don't just focus on the "most frequent" queries.

> **Also watch out for rare but expensive queries.**
> A single poorly-written report or ad-hoc analytics job can bring your system to its knees — even if it only runs once a day.

□ Summary

If there's one extension every PostgreSQL DBA should enable, it's `pg_stat_statements`.

- Set it up **early**, before performance problems arise.
- Use its rich dataset to **monitor**, **analyze**, and **optimize** queries in real-time.
- Pair it with additional PostgreSQL parameters like `track_io_timing` for deeper visibility.
- Leverage it with tools like **pgBadger**, **Grafana**, and **custom scripts** to automate performance monitoring.

As PostgreSQL continues to evolve (especially with upcoming improvements in PostgreSQL 17), tools like `pg_stat_statements` remain **essential for scalable, high-performance systems**.

> Whether you're running a SaaS platform, data warehouse, or OLTP app — this is the extension that helps you stay ahead of problems.