PostgreSQL Log Management: Mastering Logging, Monitoring, and Optimization in PostgreSQL 17

J

Jeyaram Ayyalusamy

Following

18 min read
.
Jun 14, 2025
7

Press enter or click to view image in full size

PostgreSQL is widely recognized for its robustness, extensibility, and standards compliance — but one of its most underappreciated strengths is its **advanced logging system**. Whether you're a DBA hunting down

performance bottlenecks or a developer trying to audit user activity, PostgreSQL offers a comprehensive set of logging tools that provide deep visibility into the database engine.

With the right configuration, PostgreSQL's logs can serve as your:

- ☐ **Debugging toolkit**
- ☐ **Performance profiler**
- ☐ **Audit trail**
- ☐ **Monitoring backend**

In short, PostgreSQL's logging system gives you **insight into exactly what's happening, when, and why**.

□ What You'll Learn in This Guide

This article is your one-stop resource to understand and implement effective PostgreSQL logging. We'll cover everything you need to master the system and make logs truly actionable:

□ PostgreSQL Log Destinations

Learn where PostgreSQL can send its logs:

- File-based logs
- Syslog integration
- CSV logs for structured analysis
- Event logs (on Windows)
- Logging to external collectors via `stderr` piping

We'll explain how to configure each destination and when to use them based on your environment and observability stack.

□ Severity Levels of Logs

PostgreSQL categorizes logs by severity:

- `DEBUG`, `INFO`, `NOTICE`
- `WARNING`, `ERROR`, `FATAL`, and `PANIC`

You'll discover how to filter, tune, and act on these log levels so that important events surface while routine noise stays hidden.

□ Key Logging Parameters in PostgreSQL 17

PostgreSQL 17 introduces several new and refined logging parameters. We'll walk through the most impactful ones, including:

- `log_destination`
- `logging_collector`
- `log_line_prefix`
- `log_statement`, `log_duration`, and `log_min_duration_statement`
- `log_connections`, `log_disconnections`, and `log_lock_waits`

Each of these plays a role in shaping what gets logged, how, and where it goes.

□ Optimized Logging Configurations

We'll explore battle-tested logging setups tailored for:

- Development
- Performance tuning
- High-volume production systems
- Security-sensitive environments

You'll learn how to reduce overhead while capturing high-value diagnostics.

□ Analyzing Logs with Tools Like pgBadger

Raw logs are powerful — but hard to sift through. Enter **pgBadger**, an open-source tool that turns PostgreSQL logs into rich, interactive dashboards. We'll show you how to:

- Install and configure pgBadger
- Use it to visualize query statistics
- Detect slow queries, wait events, and system usage patterns

By the end, you'll be able to transform log data into **actionable insights**.

□ Why It Matters

Database logs are often the **last resort** during incidents — but with PostgreSQL, they can be your **first line of defense**. From preventing performance regressions to meeting compliance requirements, a well-configured logging system turns your PostgreSQL server into a **self-explaining system**.

□ Why Log Management Matters in PostgreSQL

PostgreSQL is a feature-rich, enterprise-grade relational database system — but even the most stable systems can encounter unexpected issues. Whether it's a sudden spike in query time, a failed transaction, or an unapproved schema change, **logs are your primary source of truth**.

That's why **effective log management in PostgreSQL isn't optional — it's essential.**

Let's break down the key reasons why PostgreSQL logging deserves your full attention.

✅1. Troubleshooting Errors and Crashes

When something goes wrong — be it a failed query, connection timeout, or a full-blown crash — your logs are the **first place to look**. PostgreSQL logs capture:

- Query errors with exact SQL
- Stack traces for fatal events
- Authentication failures
- System resource alerts (e.g., out of memory)

With a well-configured logging system, you can **pinpoint the root cause in minutes**, not hours.

✅2. Auditing User Activity and Access

In many environments — especially in finance, healthcare, or enterprise IT — **knowing who did what and when** is a requirement, not a luxury.

Logs can track:

- Logins and disconnections (`log_connections`, `log_disconnections`)
- DDL changes (via `log_statement = 'ddl'`)
- Data-modifying actions (`INSERT`, `UPDATE`, `DELETE`)

This creates an audit trail that supports **compliance**, **security investigations**, and **change control processes**.

✅3. Monitoring Long-Running Queries

Long-running queries can quietly degrade performance, consume resources, and block other transactions.

By enabling settings like:

- `log_min_duration_statement`
- `log_statement = 'mod'` or `'all'`

…you can **automatically capture** any query that exceeds a time threshold. This helps you identify and tune:

- Inefficient joins
- Missing indexes
- Blocking locks

With logs in place, you're not just reacting to slowness — you're **proactively preventing it.**

✅4. Optimizing Performance Bottlenecks

PostgreSQL logs aren't just for errors — they're also a **rich performance analytics tool**.

By analyzing:

- Query durations
- Wait events (e.g., locks, I/O waits)
- Transaction frequency

…you can uncover slow patterns and restructure your queries or indexes. Tools like **pgBadger** make this even easier by visualizing logs into digestible charts and dashboards.

✅5. Capacity Planning with Workload Metrics

Understanding how your PostgreSQL instance behaves under normal and peak loads is crucial for growth.

Logs help answer questions like:

- How many concurrent connections are typical?
- What times of day are busiest?
- Which queries are most frequent or heaviest?

Armed with this data, you can **right-size your infrastructure**, plan for scaling, and justify resource budgets with confidence.

□ Final Thought

A properly tuned PostgreSQL logging setup isn't just about collecting data — it's about **building observability into your database layer**.

Think of logs as:

- Your **black box recorder** in case of incidents
- Your **audit trail** for accountability
- Your **performance profiler** for optimization

In short, effective log management transforms PostgreSQL from a "set-it-and-forget-it" tool into a **proactive, self-reporting system** that keeps your applications resilient, secure, and fast.

□ PostgreSQL Logging Destinations Explained

PostgreSQL offers one of the most flexible and powerful logging systems among modern relational databases. One key feature is the ability to **choose where your log messages go** using the `log_destination` parameter.

Understanding your log destinations is essential for building an efficient and maintainable logging strategy — whether you're developing locally or running a mission-critical system in production.

□ Configuring `log_destination`

The `log_destination` parameter determines **where** PostgreSQL writes its log output. You can configure it to send logs to one or more output targets by using **comma-separated values**.

```
log_destination = 'stderr'          -- default
log_destination = 'stderr, csvlog'  -- multiple destinations
```

Let's break down the available options:

Press enter or click to view image in full size

Destination	Description
`stderr`	Default. Logs are written to the standard error stream. Ideal for quick debugging or when logs are captured by an external service like systemd or Docker.
`csvlog`	Writes structured logs in CSV format. Great for detailed log analysis using tools like **pgBadger** or importing into Excel/Splunk. Requires `logging_collector = on` .
`syslog`	Sends logs to the system's Unix `syslog` daemon. Useful for centralized log collection in Linux environments.
`eventlog`	Windows-only. Sends logs to the **Windows Event Log**, suitable for enterprise Windows deployments.

✓Best Practices for Production Logging

For most production environments, these are the **two most common and effective setups**:

1. `stderr` + `logging_collector = on`

- Logs are redirected from the console to disk files.
- Simple and widely supported by cloud platforms and DevOps tools.

2. `csvlog` + `logging_collector = on`

- Enables structured logging, perfect for log analyzers like **pgBadger**, **ELK Stack**, or **Grafana Loki**.
- Each log line includes fields like timestamp, duration, user, query text, and more.

These options help ensure **reliable, parseable logs** that are easy to rotate, store, and analyze.

□ Enabling the Logging Collector

By default, PostgreSQL logs to `stderr`, which means logs show up in the terminal or service log but **are not saved to files**.

To capture logs in files — **especially required for** `csvlog`—you must enable the `logging_collector`:

```
logging_collector = on
```

Once enabled:

- Logs are redirected from the terminal to **log files**.
- You can specify the folder using:

```
log_directory = 'log'
```

- And control the filename pattern using:

```
log_filename = 'postgresql-%Y-%m-%d.log'
```

By default, the logs are stored under:

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

You can view them using:

```
tail -f /var/lib/pgsql/13/data/log/postgresql-2024-06-14.log
```

□ Important Note:

- `logging_collector` must be enabled to use destinations like `csvlog`.
- Changes to this setting require a **server restart**, not just a reload.

□ Why It Matters

Choosing the right log destination is not just about where the logs go — it's about how easily they can be:

- Parsed and analyzed
- Shipped to centralized systems
- Integrated with monitoring tools
- Rotated and archived

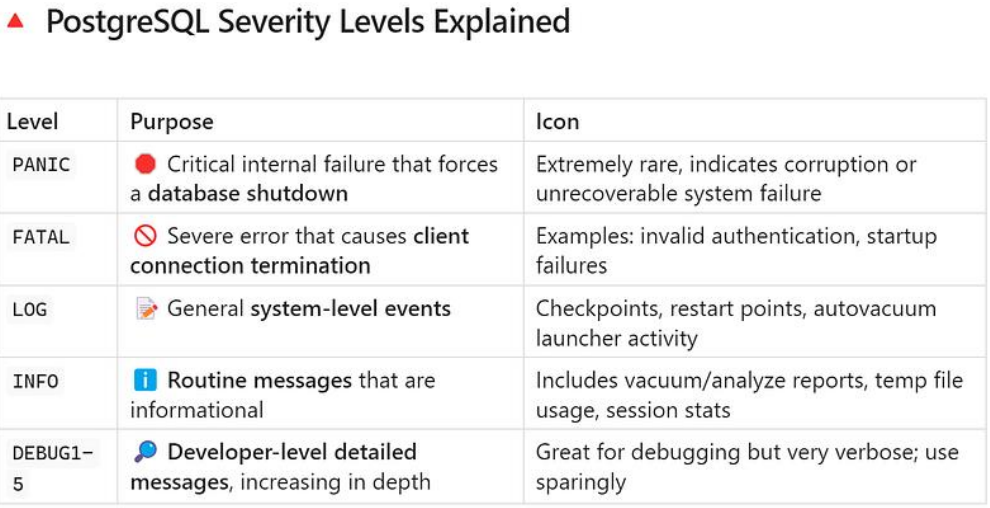
When configured properly, PostgreSQL logging gives you **complete observability** into your system with **minimal overhead**.

□ PostgreSQL Severity Levels: What Gets Logged and Why It Matters

Not all log entries are created equal. PostgreSQL uses a well-defined **severity level system** to classify every log message based on its importance and purpose. Understanding these levels helps you filter your logs effectively, minimize noise, and focus on what really matters.

Let's break them down:

Press enter or click to view image in full size



✅What to Use in Production?

In production environments, the general recommendation is to **log at `INFO` level and above**.

Why?

- `INFO` gives you visibility into important events without too much noise.
- `LOG`, `FATAL`, and `PANIC` ensure you don't miss critical system or connection-level issues.
- `DEBUG1–5`, while useful in development, can **overwhelm your logs** with low-level details.

☐ **Best Practice:**
Avoid `DEBUG` levels in production unless you're actively debugging a specific issue—and remember to turn it off afterward!

🔍 Understanding `current_logfiles`

When you enable the `logging_collector` in PostgreSQL, the system not only captures logs to files, but it also generates a helpful **tracking file** called `current_logfiles`.

This file contains a mapping of the **log destination and the active log file name** for each enabled logging format.

📄 Example Content of `current_logfiles`:

```
stderr log/postgresql.log
csvlog log/postgresql.csv
```

Here's what it tells you:

- PostgreSQL is logging to both `stderr` and `csvlog`
- The actual log files are located at `log/postgresql.log` and `log/postgresql.csv`

✅Why `current_logfiles` Matters

This file is especially useful for:

- ☐ **External log monitoring tools** (e.g., Prometheus exporters, pgBadger, log shippers like Fluentd)
- ☐ **Automated scripts** that rotate or archive logs
- ☐ **Dashboards and alerting systems** that need a dynamic reference to the current log file

It ensures your monitoring tools **don't need hardcoded paths** and can always reference the right files — even after a PostgreSQL restart or log rotation.

□ Final Thoughts

PostgreSQL's log severity levels help you **focus on the right messages at the right time**, while `current_logfiles` makes integration with external tooling smoother and more reliable.

Together, these features turn your PostgreSQL logs into a **scalable, production-grade observability layer** — essential for any serious database operation.

□ Key PostgreSQL Logging Parameters (PostgreSQL 17)

If you're managing a PostgreSQL database, logging isn't just a backend detail — it's a critical layer of observability, debugging, and performance monitoring. With PostgreSQL 17, the logging system remains powerful and highly configurable, but to get the most out of it, you need to understand how to tune it properly.

In this section, we'll break down **the most important logging parameters** that every PostgreSQL DBA should know — and explain how to check and configure them for your environment.

1□ `logging_collector` – Enable File-Based Logging

The `logging_collector` parameter controls whether PostgreSQL collects logs from `stderr` and **writes them to log files**.

```
SHOW logging_collector;
```

□ **Default:**

```
off
```

When it's off, PostgreSQL logs are sent to the console or system journal (e.g., systemd or Docker logs), which is fine for development but **not practical for production**.

✅Recommended Setting:

```
logging_collector = on
```

When set to `on`, PostgreSQL creates log files in the directory defined by `log_directory`. This setting is **required** if you want to use `csvlog` or rotate and archive your logs efficiently.

> ☐ Note: Changing this setting requires a PostgreSQL **restart**, not just a reload.

2 `log_directory` – Where Logs Are Stored

This parameter controls the **folder location** where PostgreSQL stores log files when `logging_collector` is enabled.

```
SHOW log_directory;
```

Default:

```
log
```

This means log files will be placed in a folder named `log` **inside the PostgreSQL data directory** (typically `/var/lib/pgsql/{version}/data/log/`).

✅You Can Customize:

```
log_directory = '/var/log/postgresql'
```

PostgreSQL also supports absolute paths, which is useful if you want to integrate with external storage, central log folders, or cloud-native file systems.

> ☐ Pro Tip: Ensure that the PostgreSQL user (`postgres`) has **write permissions** on the target directory.

3 `log_filename` – How Logs Are Named

The `log_filename` parameter defines the **naming convention** used for each log file. This is especially important if you're rotating logs daily or hourly.

```
SHOW log_filename;
```

Default:

```
postgresql-%Y-%m-%d_%H%M%S.log
```

This uses a **timestamp format** that includes the date and time when the log file was created. It ensures that each log file has a **unique name**, which is essential for archiving and avoiding overwrites.

✅Example File Names:

- `postgresql-2025-06-14_103000.log`
- `postgresql-2025-06-14_120000.log`

You can adjust the format based on your rotation policy. For example:

```
log_filename = 'postgresql-%Y-%m-%d.log'  -- Daily logs
```

4 `log_file_mode` – Set Unix File Permissions for Log Files

This parameter defines the **file permission mode** for newly created log files, using standard Unix-style octal notation.

```
SHOW log_file_mode;
```

Default:

```
0600
```

This means:

- Only the **PostgreSQL server owner** (usually the `postgres` user) can **read and write** log files.
- **No access is granted to group or other users**, which enhances security, especially in multi-user environments.

✅When to Modify:

If you use a **monitoring tool** or **log shipper** (like Fluentd or Filebeat) that runs under a different user and needs access to the log files, you might need to relax this setting slightly, such as:

```
log_file_mode = 0640  -- Allow group read access
```

Just be cautious: broader permissions (like `0666`) are rarely recommended in production due to security risks.

5 `log_rotation_age` – Time-Based Log Rotation

This parameter controls how **frequently log files are rotated based on time**.

```
SHOW log_rotation_age;
```

Default:

```
1d (24 hours)
```

This means PostgreSQL will start a new log file **every 24 hours**, even if the previous log file hasn't reached its size limit.

✅Customize Based On Your Needs:

- For high-volume environments, you may want **hourly rotation**:

```
log_rotation_age = 1h
```

- To **disable time-based rotation** entirely:

```
log_rotation_age = 0
```

This setting works hand-in-hand with `log_filename`. If your filename format includes a timestamp, rotation ensures files don't grow too large and remain easy to manage.

6 `log_rotation_size` – Size-Based Log Rotation

This parameter controls when PostgreSQL rotates log files **based on their size**.

```
SHOW log_rotation_size;
```

Default:

```
10MB
```

Once a log file reaches 10 megabytes, PostgreSQL closes the current file and starts a new one — even if the rotation time interval hasn't passed.

✅To adjust:

- For smaller, more frequent files:

```
log_rotation_size = 5MB
```

- To **disable size-based rotation**:

```
log_rotation_size = 0
```

You can combine this with `log_rotation_age` for **hybrid rotation**:

- Time-based (e.g., daily)
- Size-based (e.g., every 50MB)

This dual control helps you **keep logs manageable** and reduces the chance of oversized files overwhelming your disk or log parser.

□ Why These Settings Matter

Together, these three parameters give you **complete control over log file behavior**:

Parameter Purpose `log_file_mode` Controls file-level permissions `log_rotation_age` Triggers log rotation based on time `log_rotation_size` Triggers log rotation based on file size

Properly tuning them helps you:

- ☐ Enforce security with appropriate permissions
- ☐ Maintain log hygiene with regular rotations
- ☐ Ensure compatibility with external tools that read logs

7□ `log_truncate_on_rotation` – Overwrite vs. Append on Rotation

This setting determines what PostgreSQL does with an existing log file **when it rotates**.

```
SHOW log_truncate_on_rotation;
```

Default:

```
off
```

With this setting:

- `off` **(default)**: PostgreSQL **appends** new logs to the existing file if the filename hasn't changed.
- `on`: PostgreSQL **overwrites** the log file when rotating—erasing previous log content if the same filename is used.

✅Use Case:

If you use a **fixed filename pattern** like `postgresql.log` (no timestamp), and you want to start fresh with each rotation, set:

```
log_truncate_on_rotation = on
```

If your log filenames already include timestamps (e.g., `postgresql-%Y-%m-%d.log`), this setting has no effect and can safely remain `off`.

> ☐ Caution: Enabling this with a fixed filename may cause **log loss** if not archived beforehand.

8 `syslog_facility` – Define Syslog Category for PostgreSQL Logs

If you've configured PostgreSQL to use `syslog` as a log destination, the `syslog_facility` parameter lets you specify **which syslog category (facility)** the messages will use.

```
SHOW syslog_facility;
```

Default:

```
LOCAL0
```

✅Available Values:

- `LOCAL0` to `LOCAL7`

Each value corresponds to a different **custom syslog stream**, allowing your logs to be tagged and filtered easily by external syslog daemons (like `rsyslog`, `syslog-ng`, or a SIEM).

✅Use Case:

You might configure PostgreSQL to use `LOCAL1` and your web server to use `LOCAL2`—making it easier to route logs into separate files or forward them differently.

Example in `postgresql.conf`:

```
log_destination = 'syslog'
syslog_facility = 'LOCAL3'
```

> ☐ This setting is only relevant **if** `log_destination` **includes** `syslog`.

9 `log_min_messages` – Minimum Severity Level to Log

This parameter controls the **lowest severity level** that will be written to the log.

```
SHOW log_min_messages;
```

Default:

```
WARNING
```

This means only messages of level `WARNING`, `ERROR`, `FATAL`, or `PANIC` will be logged. Less severe messages (like `INFO` or `DEBUG`) will be ignored.

✅Full Range of Values (from most to least verbose):

Value Logs messages at this level and more severe `DEBUG5` Extremely detailed debugging (lowest threshold) `DEBUG4` `DEBUG3` `DEBUG2` `DEBUG1` `INFO` Informational messages (e.g., vacuum progress) `NOTICE` Notices for user-facing events `WARNING` Warnings (default) `ERROR` Errors that caused a command to fail `FATAL` Connection-ending errors `PANIC` Database-shutdown-causing errors

✅Use Case:

- For **routine production use**, `WARNING` or `ERROR` is recommended.
- For **performance tuning or deep debugging**, temporarily lower it to `INFO` or even `DEBUG1`.

> ☐ Lower levels like `DEBUG5` generate **a massive volume of logs** and should only be used in isolated debugging scenarios.

□ `log_min_error_statement` – Log SQL That Triggers Errors

```
SHOW log_min_error_statement;
```

This parameter determines the **minimum severity level** at which PostgreSQL will log the **entire SQL statement** that caused the error.

□ **Default:**

```
ERROR
```

When an error occurs — say, a `constraint violation` or a `division by zero`—PostgreSQL logs the error message. But unless this parameter is set appropriately, you won't see **which exact query caused it**.

✅**Example:**

```
INSERT INTO users (id, name) VALUES (1, NULL);  -- violates NOT NULL constraint
```

Log output (with default setting):

```
ERROR:  null value in column "name" violates not-null constraint
STATEMENT:  INSERT INTO users (id, name) VALUES (1, NULL);
```

This makes it much easier to trace and debug user errors or application bugs.

✅**Recommended Usage:**

- Use `ERROR` in production to **log meaningful failures**.
- Consider `WARNING` in dev/test environments to catch more detail (e.g., usage of deprecated features).

1□1□ `log_min_duration_statement` – Detect and Log Slow Queries

```
SHOW log_min_duration_statement;
```

This parameter is a **game-changer** for performance tuning. It logs any SQL statement that runs longer than a specified duration, measured in **milliseconds**.

□ **Default:**

```
-1  -- Disabled
```

✅**Example:**

Set it to log queries that take more than 500 milliseconds:

```
log_min_duration_statement = 500
```

Then run:

```
SELECT * FROM orders WHERE status = 'pending';  -- assume large table
```

If it runs for 600ms, you'll see:

```
LOG:  duration: 600.231 ms  statement: SELECT * FROM orders WHERE status =
'pending';
```

□ Why It Matters:

- Helps you **find expensive queries**
- Identifies **missing indexes**
- Tracks **query spikes during traffic peaks**

☐ **Pro Tip**: In QA or development, set it to `0` to log **all query durations**. In production, tune it to something realistic—e.g., `100-500 ms`.

1□2□ `application_name` – Track Which App Ran the Query

```
SHOW application_name;
```

This isn't a log setting per se, but it has major implications for logging clarity. PostgreSQL allows clients to specify an **application name**, which then appears in logs (if `log_line_prefix` includes `%a`).

✅How to Set:

From within SQL:

```
SET application_name = 'api_service';
```

Or via connection string:

```
postgresql://user:pass@host/db?application_name=reporting_tool
```

✅Use Cases:

- Monitor behavior of **specific microservices**
- Separate logs per environment (e.g., dev, test, prod)
- Troubleshoot slowdowns linked to **a specific app**, not the database

☐ Combine this with `log_line_prefix = '%t [%p]: [%a] %u@%d '` to print the application name on each log line.

1□3□ `log_checkpoints` – Log System-Level Checkpoint Events

```
SHOW log_checkpoints;
```

Checkpoints are periodic operations where PostgreSQL **writes dirty buffers (modified data)** from memory to disk. While critical for durability, they can slow down queries or cause I/O spikes if not tuned.

□ Default:

```
off
```

Set it to:

```
log_checkpoints = on
```

Now you'll see entries like:

```
LOG:  checkpoint complete: wrote 3050 buffers (22.5%); write=3.421 s, sync=0.015 s, total=3.436 s
```

✅Why It's Important:

- Reveals **how much data was flushed**
- Shows **how long writes and syncs took**
- Helps tune checkpoint parameters (`checkpoint_timeout`, `checkpoint_completion_target`, etc.)

This is especially useful if you're experiencing:

- ☐ **Sudden performance drops**
- ☐ **Disk I/O bottlenecks**
- ☐ **High-frequency checkpoints due to small `shared_buffers` or aggressive `autovacuum`**

1□4□ `log_connections` – Log Every Incoming Client Connection

```
SHOW log_connections;
```

This parameter controls whether PostgreSQL logs a message **every time a client successfully connects** to the database.

□ **Default:**

```
off
```

When enabled:

```
log_connections = on
```

You'll see entries like:

```
LOG:  connection authorized: user=app_user database=mydb application_name=myapp
```

✅Why It Matters:

- **Tracks login activity** for auditing and compliance
- Helps identify **excessive connection churn** (e.g., apps not using connection pooling)
- Useful for spotting **unauthorized or unexpected access attempts**

☐ **Pro Tip**: Combine with `application_name` to know which app or service is connecting.

1□5□ `log_disconnections` – Log Session Termination Details

```
SHOW log_disconnections;
```

This parameter logs a message **when a client disconnects** from the database.

□ Default:

```
off
```

When enabled:

```
log_disconnections = on
```

You'll see entries like:

```
LOG:  disconnection: session time: 0:01:32.456 user=app user database=mydb
host=10.0.0.5 port=53044
```

✅Why It's Useful:

- Tracks **connection duration** and total session time
- Helps detect **short-lived or high-frequency sessions** (often signs of bad connection management)
- Pairs well with `log_connections` to give a **full connection lifecycle audit**

☐ This is particularly helpful for **security monitoring** and **debugging flaky apps** that open and drop connections frequently.

1□6□ `log_duration` – Log Execution Time for Every Query

```
SHOW log_duration;
```

This parameter logs the **execution time** of each SQL statement — regardless of whether it's fast or slow.

□ Default:

```
off
```

When enabled:

```
log_duration = on
```

You'll see log lines like:

```
LOG:  duration: 12.541 ms
```

If combined with `log_statement = 'all'`, the logs will show:

```
LOG:  duration: 142.231 ms
LOG:  statement: SELECT * FROM users WHERE id = 123;
```

□ `log_duration` vs. `log_min_duration_statement`:

Feature `log_duration` `log_min_duration_statement` Logs all queries? ✓ Yes ✗ Only those slower than threshold Volume of logs ☐ High ☐ Tunable (based on duration) Best for Dev/testing environments Production performance tuning

□ Caution:

While `log_duration` offers full visibility, it can generate **significant log volume**, especially in systems with high query throughput. Use with care in production systems.

□ Summary: Monitor the Full Session Lifecycle

Together, these three parameters help you **monitor the lifecycle of client sessions and their query behavior**:

Parameter What It Does Best Use Case `log_connections` Logs every incoming connection Track who is connecting `log_disconnections` Logs when each session ends, with session duration Audit session lifecycle, spot short-lived apps `log_duration` Logs how long each query takes Full query profiling (best in dev/test)

✅Final Thoughts

If you want to fully understand **how users and applications interact** with your PostgreSQL database — from login to logout, and everything in between — these parameters are must-haves.

They allow you to:

- ☐☐ Trace user activity
- ☐ Measure session and query durations
- ☐ Strengthen security visibility
- ☐ Improve app connection efficiency

Whether you're tuning performance, debugging connection issues, or auditing behavior for compliance, these logging parameters give you the observability you need to act with confidence.

□ Bonus: Optimized Logging Setup for pgBadger Analysis

If you're serious about PostgreSQL performance tuning and auditability, you'll eventually want to analyze your logs using a visualization tool. One of the most powerful and widely used tools for this purpose is **pgBadger**.

pgBadger is a PostgreSQL log analyzer that turns raw PostgreSQL logs into insightful, interactive dashboards — showing slow queries, lock contention, checkpoint frequency, autovacuum activity, and more.

To get the **best results from pgBadger**, you need to feed it **clean, structured, and comprehensive logs** — without overwhelming it with noise. Here's a **battle-tested configuration** for that:

```
log_checkpoints = on                    -- Log each checkpoint (duration, buffers,
etc.)
log_connections = on                    -- Log every new client connection
log_disconnections = on                 -- Track when users disconnect and session
durations
log_lock_waits = on                     -- Log statements that wait for locks (very
useful!)
log_temp_files = 0                      -- Log all temporary file creations (helps
spot memory issues)
log_autovacuum_min_duration = 0         -- Log all autovacuum events (for performance
tuning)
log_error_verbosity = default           -- Keep error output concise but informative
log_statement = off                     -- Avoid logging every query (too noisy in
production)
log_min_duration_statement = 10         -- Log all queries taking longer than 10 ms
```

✅Why this setup works:

- ☐ **Sufficient detail** for performance analysis and security audits
- ☐ **Balanced noise vs. visibility** — you see what matters, not every SELECT
- ☐ **Fully compatible** with pgBadger's expected log structure and analysis tools

This setup gives you **just enough insight** without overwhelming storage or generating gigabytes of logs per hour in busy systems.

☐ Log Rotation Best Practices

As you begin capturing more detailed logs, managing log file size and retention becomes critical. Without proper rotation, your log directory can quickly balloon and cause disk space issues.

Here are best practices to keep your log storage healthy and manageable:

□ Use both time-based and size-based rotation:

```
log rotation age = 1d       -- Rotate logs daily
log_rotation_size = 100MB   -- Or when the file reaches 100MB
```

Using both ensures:

- ☐ Logs rotate **regularly**, even during low traffic
- ☐ Large spikes in activity won't create **massive single log files**

□ Ensure disk space is monitored:

- Store logs on a **dedicated volume** if possible
- Use tools like `logrotate`, `cron`, or **cloud-native log management** systems for retention and archiving

□ Automate log cleanup:

Set up scheduled jobs to:

- Compress old logs
- Archive or delete logs after a retention period (e.g., 30 days)
- Ship logs to centralized storage or analysis tools

This keeps your logging pipeline lean, compliant, and performant.

□ Key Takeaways

PostgreSQL's logging system is **one of the most powerful among relational databases**, but to unlock its full potential, it requires **intentional setup**.

Here's what you should remember:

✅1. Enable `logging_collector` in production

Without this, logs won't be written to files — making analysis and troubleshooting nearly impossible.

✅2. Use `csvlog` for structured logs

Structured logs make parsing and analysis easier for tools like pgBadger, Datadog, or ELK Stack.

✅3. Set `log_min_duration_statement`

This is your secret weapon for finding slow queries. Even a value like `100 ms` can uncover performance hotspots.

✅4. Use monitoring tools

Don't sift through raw logs manually.
Leverage **pgBadger**, **ELK**, **Prometheus**, or **Grafana** to visualize trends and anomalies.

✅5. Avoid logging everything

While `log_statement = 'all'` gives you full visibility, it's overkill for production and can cause performance issues and log flooding. Use targeted logging instead.

□ Pro Tip:

> “You can’t fix what you can’t see. Your logs are your best friend when your database whispers its secrets.”

Logging is more than just output — it’s how your database talks to you. With the right configuration, PostgreSQL’s logs become a **real-time feedback loop**, telling you what’s working, what’s failing, and what’s silently becoming a problem.

☐ Now that you’re equipped with everything you need to configure logging in PostgreSQL, go ahead — **tune your system, visualize your queries, and stay ahead of the curve.** Your future self (and your production environment) will thank you.