

PostgreSQL 17 Logging Best Practices: How to Monitor, Troubleshoot & Tune Your Database with Confidence



Jeyaram Ayyalusamy

Following

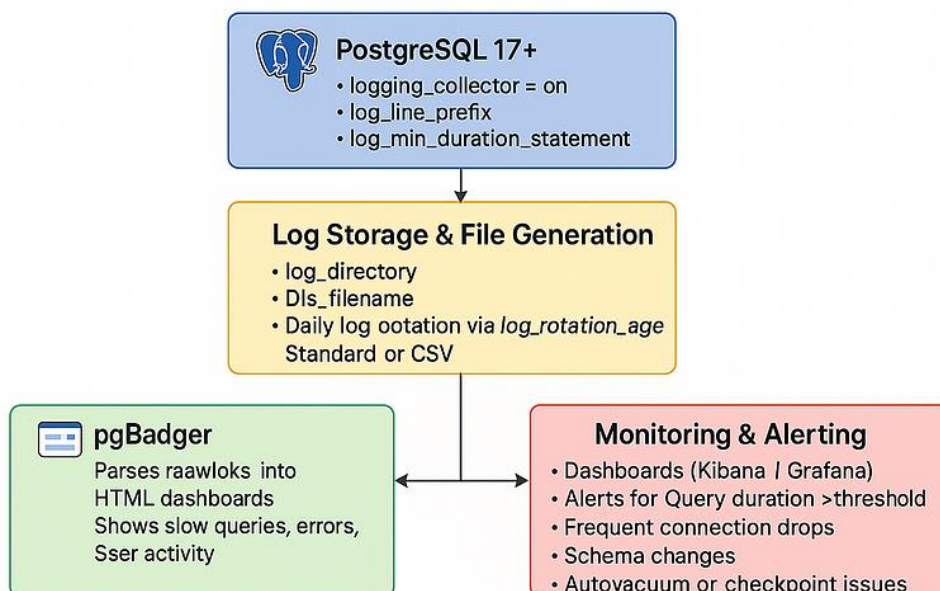
23 min read

Jul 11, 2025

2

Press enter or click to view image in full size

PostgreSQL Logging Architecture for Observability & Performance Tuning



PostgreSQL is powerful, stable, and reliable — a favorite among developers and database administrators worldwide. But even the most trusted systems require visibility under the hood.

When things go wrong — whether it's a sudden performance drop, failed transaction, or strange application behavior — your logs are the first place to look. Without them? You're flying blind.

Observability is not optional — it's mission-critical.

In production environments, troubleshooting without proper logging is like debugging in the dark. You miss key details, waste time guessing, and risk repeating the same issues. The ability to **trace events, catch anomalies, and understand query behavior** depends entirely on having a solid logging setup.

□ **Why Logging Configuration Matters**

PostgreSQL ships with logging features out of the box — but the default configuration often leaves a lot to be desired. It's not about just having logs — it's about making them **actionable, readable, and useful**.

That's exactly what we'll focus on in this post.

Whether you're a developer deploying your first database or a seasoned DBA managing a fleet of nodes, tuning PostgreSQL's logging can help you:

- Pinpoint slow queries
- Understand user behavior
- Monitor application connections
- Track schema changes
- Perform better root cause analysis during incidents

□ **What You'll Learn**

In this post, I'll show you **exactly how to configure PostgreSQL logging** — step-by-step — using best practices that are:

- Proven in production environments
- Easy to implement
- Focused on clarity, not noise

By the end, you'll be able to turn raw logs into valuable insights — no guesswork, no overwhelm.

□ Why PostgreSQL Logging Matters (And Why Defaults Aren't Enough)

PostgreSQL is a world-class open-source database, but what truly unlocks its full potential is observability — the ability to see what's happening inside your system in real time.

When properly configured, PostgreSQL logging gives you deep insight into your database behavior. From catching slow queries to tracing suspicious activity, logs become an essential tool in your operational toolkit. But here's the catch:

Out of the box, PostgreSQL logging isn't very useful.

You have to **configure it** to get value from it.

Let's look at **why logging matters** in the first place □

□ Proactive Troubleshooting

PostgreSQL logging lets you catch issues before they become outages. By logging long-running queries or unexpected errors, you can address performance or data problems early — before users feel the pain.

□ Real-Time Monitoring

Logs help you **detect application or connection issues as they happen**. For example, a spike in failed logins, connection drops, or timeouts can point to issues in the app or infrastructure. Good logs make it easy to correlate these patterns quickly.

□ Performance Analysis

Ever wondered what's slowing down your database? Logging lets you identify the exact queries causing **I/O or CPU bottlenecks**. When combined with thresholds (like duration-based logging), you get a clear picture of what needs tuning.

□ Auditing

Need to know **who changed what** in your database? Logging DDL statements (like CREATE, ALTER, DROP) provides a lightweight audit trail. This is crucial in regulated environments or teams with multiple developers pushing schema changes.

□ Security Visibility

From unauthorized access attempts to unusual client connection patterns, logging plays a key role in **monitoring database security**. It gives you the first clues when something suspicious is happening — and helps you respond faster.

□ But Here's the Problem...

PostgreSQL's **default logging configuration is minimal**. It might not record slow queries, statement types, or even connection errors unless explicitly enabled.

That's why configuring PostgreSQL logging isn't just helpful — **it's essential**.

In the next section, we'll explore exactly **how to tune your logging configuration** to turn PostgreSQL into a fully observable system — without overwhelming your log files.

□ PostgreSQL 17 Logging Best Practices: A Complete Guide to Meaningful Observability

PostgreSQL 17 continues to shine as one of the most powerful and reliable relational databases in the world. But no matter how efficient your queries are or how well your schema is designed, **you can't troubleshoot what you can't see.**

Effective logging is the foundation of:

- Debugging complex issues
- Monitoring live systems
- Auditing schema changes
- Detecting security incidents
- Tuning for performance

In this guide, we'll walk through **10 PostgreSQL 17 logging best practices**, along with the exact `ALTER SYSTEM` commands you can use, and **inline explanations** for what each setting does — and why it matters.

❑ 1. Enable the Logging Collector

```
ALTER SYSTEM SET logging_collector = on;
```

❑ **Inline Explanation:** This activates a PostgreSQL background process that captures log output and writes it to files, instead of just sending it to the terminal or disappearing into `stderr`.

✓ **Why It Matters:**

Without enabling `logging_collector`, logs won't be saved to disk — meaning you can't rotate, store, analyze, or ship them to external tools

like ELK or CloudWatch. This setting is the **first and most essential step** toward persistent, structured logging.

☐ 2. Set the Log Directory and File Name Format

```
ALTER SYSTEM SET log_directory = 'log';  
ALTER SYSTEM SET log_filename = 'postgresql-%Y-%m-%d.log';
```

☐ **Inline Explanation:** Sets the destination folder for log files and formats the filename based on the date (e.g., `postgresql-2025-07-06.log`).

✓ **Why It Matters:**

Organizing logs by date and placing them in a dedicated `log/` directory makes it easy to manage, automate, and troubleshoot. During incidents, it becomes much easier to grab logs from the exact time the issue occurred.

☐ 3. Enable Age-Based Log Rotation

```
ALTER SYSTEM SET log_rotation_age = '1d';  
ALTER SYSTEM SET log_rotation_size = 0;
```

☐ **Inline Explanation:** Tells PostgreSQL to create a new log file every day (`1d`) and ignore file size when rotating logs (`0`).

✓ **Why It Matters:**

Without log rotation, a single file could grow indefinitely, consuming storage and slowing down analysis. Age-based rotation is predictable and works well with automated archiving and cleanup tools.

☐ 4. Use a Structured Log Line Prefix

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

❑ **Inline Explanation:** Adds structured metadata to every log entry — including timestamp, process ID, user, database, client IP, and application name.

✓ **Why It Matters:**

Rich prefixes make your logs **searchable**, **filterable**, and easier to analyze with tools like `pgBadger`, Kibana, or even `grep`. This format helps you trace actions by user, by session, or by client machine.

❑ **5. Log All Connection and Disconnection Events**

```
ALTER SYSTEM SET log_connections = on;  
ALTER SYSTEM SET log_disconnections = on;
```

❑ **Inline Explanation:** Logs every new client connection and every disconnection from the database.

✓ **Why It Matters:**

These logs are vital for tracking session-level activity and detecting issues like:

- Application connection spikes
- Connection leaks
- Unauthorized access attempts

They also help correlate user behavior and load patterns during high-traffic periods.

❑ 6. Log Only DDL (Schema) Changes

```
ALTER SYSTEM SET log_statement = 'ddl';
```

❑ **Inline Explanation:** Tells PostgreSQL to log only Data Definition Language (DDL) statements like `CREATE`, `ALTER`, and `DROP`.

✓Why It Matters:

Logging every SQL statement (`log_statement = 'all'`) floods your logs. By focusing on DDL, you still capture important schema changes for auditing — without the noise of frequent `SELECT` queries.

❑ 7. Log Slow Queries (> 500ms)

```
ALTER SYSTEM SET log_min_duration_statement = 500;
```

❑ **Inline Explanation:** Logs any query that takes more than **500 milliseconds** to complete.

✓Why It Matters:

This helps you **identify slow queries** that may need indexing, refactoring, or caching. You can tune the threshold for your workload:

- High-traffic apps: 100ms–250ms
- OLAP/reporting: 1000ms+
- General apps: 500ms (recommended default)

This is one of the best tools for ongoing performance tuning.

□ 8. Log Checkpoints and Lock Waits

```
ALTER SYSTEM SET log_checkpoints = on;  
ALTER SYSTEM SET log_lock_waits = on;
```

□ **Inline Explanation:** `log_checkpoints` shows when PostgreSQL flushes data to disk; `log_lock_waits` reveals blocking caused by concurrent access or long-running transactions.

✓ Why It Matters:

- **Checkpoints** are performance-critical: too frequent and they hurt I/O, too rare and you risk long recovery times.
- **Lock waits** highlight concurrency issues, especially in transactional or OLTP systems. These logs help uncover deadlocks and contention before users report them.

□ 9. Manage Log File Size and Rotation

```
ALTER SYSTEM SET log_rotation_age = '1d';  
ALTER SYSTEM SET log_rotation_size = 0;
```

□ **Inline Explanation:** Confirms PostgreSQL rotates logs daily, rather than based on size.

✓ Why It Matters:

Age-based rotation works well in environments where log ingestion is time-bound (e.g., daily backups or hourly syncs). It simplifies retention policies and ensures logs don't grow uncontrollably.

□ 10. Optimize for Clarity: Limit Noise and Verbosity

```
ALTER SYSTEM SET log_min_messages = 'warning';
ALTER SYSTEM SET log_duration = 'off';
ALTER SYSTEM SET log_error_verbosity = 'default';
```

□ Inline Explanation:

- `log_min_messages = 'warning'` prevents excessive debug/info logging.
- `log_duration = 'off'` stops logging durations unless analyzing performance.
- `log_error_verbosity = 'default'` balances useful info with readability.

✓ Why It Matters:

You want your logs to be **informative, not overwhelming**. These settings strike a balance between verbosity and clarity — capturing important warnings and errors while filtering out unhelpful noise.

□ Combined Logging Configuration (Ready to Apply)

Here's a copy-paste ready configuration block that applies all best practices:

```
ALTER SYSTEM SET logging_collector = on;
ALTER SYSTEM SET log_directory = 'log';
ALTER SYSTEM SET log_filename = 'postgresql-%Y-%m-%d.log';
ALTER SYSTEM SET log_rotation_age = '1d';
ALTER SYSTEM SET log_rotation_size = 0;
ALTER SYSTEM SET log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h ';
ALTER SYSTEM SET log_statement = 'ddl';
ALTER SYSTEM SET log_min_duration_statement = 500;
ALTER SYSTEM SET log_checkpoints = on;
ALTER SYSTEM SET log_connections = on;
ALTER SYSTEM SET log_disconnections = on;
ALTER SYSTEM SET log_lock_waits = on;
ALTER SYSTEM SET log_min_messages = 'warning';
ALTER SYSTEM SET log_duration = 'off';
ALTER SYSTEM SET log_error_verbosity = 'default';
```

✓ Reload PostgreSQL after making changes:

```
SELECT pg_reload_conf();
```

□ Final Thoughts

Logging in PostgreSQL isn't just about collecting data — it's about making that data **useful and actionable**.

By following these 10 best practices, you'll have:

- □ Targeted logging for performance and auditability
 - □ Structured output for easier analysis
 - □ Security visibility through session tracking
 - □ Tools to troubleshoot problems before users report them
- Whether you're running a startup on Heroku or managing enterprise-scale clusters on AWS, **smart logging makes PostgreSQL easier to operate, tune, and trust**.

11 □ Use pgBadger for PostgreSQL Log Analysis

Once you've properly configured PostgreSQL logging (see points 1–10), the next logical step is to **analyze those logs efficiently**. Raw logs are valuable but difficult to navigate. Manually combing through thousands of lines of log output is both tedious and error-prone.

That's where **pgBadger** becomes a game-changer.

✓ What is pgBadger?

pgBadger is a fast, flexible, and fully open-source PostgreSQL log analyzer written in Perl. It reads your PostgreSQL log files and transforms them into **beautiful, interactive HTML reports** that offer deep insights into:

- ☐ **Slow query analysis**
- ☐ **Error and warning trends**
- ☐ **Autovacuum activity visibility**
- ☐ **Top queries by execution time or frequency**
- ☐ **Session-based activity and client insights**

It supports both standard and CSV log formats and is compatible with PostgreSQL versions from 9.x through 17.

☐ **Sample pgBadger Output**

When you run pgBadger on your PostgreSQL log file, it generates an interactive web report (usually named `report.html`) with graphs, filters, and sortable tables.

☐ **Visuals you get include:**

- Query runtime histograms
- Error breakdown by hour
- Connections per client/IP
- DDL/DML distribution
- Index usage patterns

This visibility allows you to **pinpoint bottlenecks, misbehaving queries, or workloads that need optimization** — without digging through text-based logs.

☐ **How to Use pgBadger**

If you have a PostgreSQL log file ready (like `postgresql-2025-07-06.log`), running pgBadger is straightforward:

```

pgbadger postgresql-2025-07-06.log -o report.html

[postgres@ip-172-31-92-46 ~]$ cd /var/lib/pgsql/17/data/log

[postgres@ip-172-31-92-46 log]$ ls -ltr
total 68
-rw-----. 1 postgres postgres 8959 Jul 10 23:26 postgresql-Thu.log
-rw-----. 1 postgres postgres 54039 Jul 11 01:11 postgresql-Fri.log

[postgres@ip-172-31-92-46 log]$ pgbadger postgresql-Fri.log -o report.html
Attempt to call undefined import method with arguments ("tmpdir") via package
"File::Spec" (Perhaps you forgot to load the package?) at /usr/local/bin/pgbadger
line 46.
[=====>] Parsed 54039 bytes of 54039 (100.00%), queries: 26,
events: 10
LOG: Ok, generating html report...

[postgres@ip-172-31-92-46 log]$ ls -ltr
total 932
-rw-----. 1 postgres postgres 8959 Jul 10 23:26 postgresql-Thu.log
-rw-----. 1 postgres postgres 54039 Jul 11 01:11 postgresql-Fri.log
-rw-r--r--. 1 postgres postgres 884439 Jul 11 01:40 report.html
[postgres@ip-172-31-92-46 log]$

```

❑ **Output:**

An interactive HTML file with charts and summaries you can open in any browser.

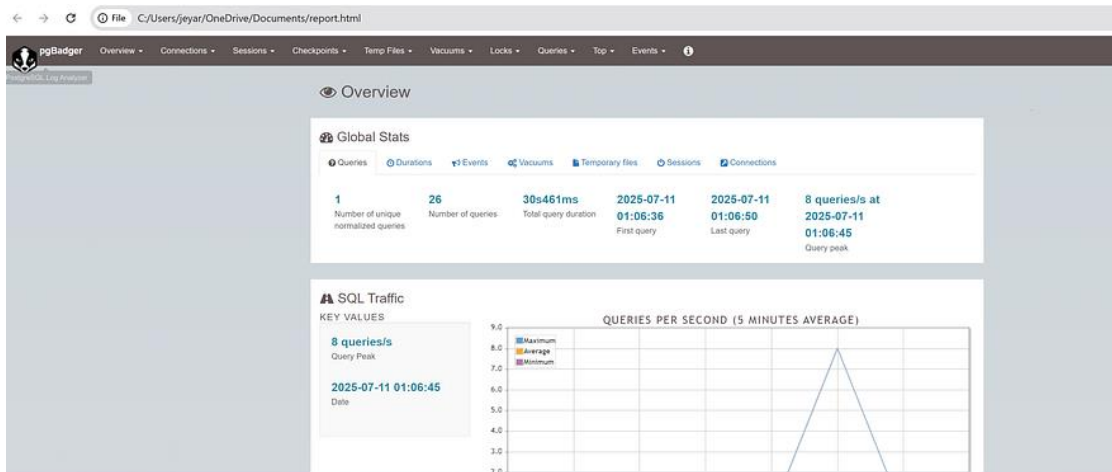
You can also process multiple files at once or automate it via cronjobs or CI pipelines:

```
pgbadger /var/lib/pgsql/data/log/*.log -o /var/www/html/pg_reports/daily_report.html
```

❑ **Understanding pgBadger Report Sections (Based on Screenshot)**

The **pgBadger HTML report** provides deep insights into PostgreSQL performance metrics. Let's explore the sections highlighted in **red boxes** in your screenshot.

Press enter or click to view image in full size



❑ 1. Top Navigation Bar (Modules)

Located at the top of the report, this horizontal menu provides access to various performance-related modules:

Press enter or click to view image in full size

Menu	Purpose
Overview	Summarized view of overall PostgreSQL performance
Connections	Number of active connections and their behavior
Sessions	Session lifecycle and session peaks
Checkpoints	Checkpoint frequency, I/O impact
Temp Files	Temp file usage, which may indicate insufficient work_mem
Vacuums	Autovacuum and manual vacuum operations
Locks	Details about blocking and wait events
Queries	Slowest queries, most frequent queries, and execution times
Top	"Top N" views across queries, clients, databases
Events	SQL or system-related alerts, disconnections, errors

This section (❑ Overview) is the **default landing page**. It gives a bird's-eye view of:

- Total number of queries
- Duration stats
- Query peaks

- First/last query timestamps

This is useful for a quick health snapshot.

❑ 3. Global Stats Panel (Tabs)

This panel provides **clickable tabs** to switch between specific global stats:

Press enter or click to view image in full size

Tab	Description
Queries	Total unique queries and execution count
Durations	Total duration of all queries
Events	Logged error and warning messages
Vacuums	Counts of vacuum and autovacuum runs
Temporary Files	Number of temporary files created
Sessions	Number of sessions tracked
Connections	Active/total connection statistics

These tabs help in **isolating bottlenecks** like frequent temp files or slow sessions.

❑ 4. SQL Traffic Section

This section shows the **Query Rate Over Time**:

- **KEY VALUES:** Peak query throughput (e.g., 8 queries/s)
- **Chart:** A graph showing queries per second in 5-minute averages
- ❑ Maximum, Average, and Minimum lines

This helps you **visualize traffic spikes**, which can be correlated with performance issues or system load.

✔ Use Case

You can **navigate through these sections** to answer performance questions like:

- *What time had the highest query load?*
- *Which queries are consuming most time?*
- *Are there frequent autovacuum operations?*
- *Are there blocking locks or idle sessions?*

□ 5. □ SELECT Traffic

►Section: `SELECT Traffic` (Key Values Panel)

- **0 queries/s** — This indicates there were no SELECT (read-only) operations recorded in the monitored period.
- This section is important for:
- Monitoring dashboard/report usage
- Checking for underutilized read operations

►Graph: `SELECT QUERIES (5 MINUTES PERIOD)`

- Graph displays a **5-minute moving average** of:
- `Maximum`, `Average`, and `Minimum` SELECT queries per second
- In this case, the graph is flat (zero) — indicating **no read load** in the timeframe

□ 6. □ INSERT/UPDATE/DELETE Traffic

►Section: `INSERT/UPDATE/DELETE Traffic` (Key Values Panel)

- **8 queries/s** — This reflects peak write activity (INSERT/UPDATE/DELETE) at a given timestamp.
- **Timestamp:** 2025-07-11 01:06:45 — this helps correlate load with application actions

- High write traffic could indicate:
- Batch jobs
- ETL processes
- App-level writes during business hours

► **Graph:** `WRITE QUERIES (5 MINUTES PERIOD)`

- Tracks 3 types of write operations:
- **INSERT queries** (in red)
- **UPDATE queries** (blue)
- **DELETE queries** (orange)
- Helps determine which type of write operation dominates system activity

✓ **Why It Matters:**

Section Insight **SELECT Traffic** Indicates app reads, dashboard usage, or query performance **WRITE Traffic** Shows if load spikes are due to ETL jobs, application writes, or updates **5-Minute Graphs** Useful for tracking **query bursts**, latency windows, and contention periods

□ **Deep Dive into Query Duration, Prepared Queries, and General Activity in pgBadger**

When analyzing PostgreSQL performance with **pgBadger**, understanding query durations, prepared statement usage, and overall query activity is essential for pinpointing bottlenecks and tuning the database effectively.

□ **1. □ Queries Duration**

► **Key Values Panel**

- **30s461ms** is the **total execution time** of all queries during the observed period.
- This value reflects how busy the PostgreSQL engine was in executing SQL commands.

► **Graph:** Average Queries Duration (5 Minutes Average)

- Shows average latency of:
- **All Queries**
- **SELECT Queries**
- **Write Queries** (INSERT, UPDATE, DELETE)
- Useful to identify:
- Query spikes or slowdowns
- Specific time intervals causing performance degradation
- In this report, a **latency spike** occurred just after **21:00**, possibly due to a batch job or write-heavy transaction.

□ 2. □ Prepared Queries Ratio

► **Key Values Panel**

- **Ratio of bind vs prepare:** 0.00
- **Prepared vs usual statements:** 0.00%
- Indicates **no prepared statements** were used during this period.

Prepared statements help improve performance by reusing execution plans, reducing parse overhead, especially in high-throughput applications.

► **Graph:** Bind vs Prepare Statements (5 Minutes Average)

- Monitors:

- `Prepare/Parse` activity
- `Execute/Bind` operations
- `Bind vs Prepare` effectiveness
- No activity shown — this could mean:
- The application is not using **PREPARE/EXECUTE SQL**
- A potential opportunity for optimization

□ 3. □ **General Activity Table**

This bottom panel summarizes:

- **Daily and Hourly stats**
- Total **query count**: `28`
- **Min/Max/Avg duration** of queries
- **Latency percentiles (90/95/99)** — all `30s461ms`, consistent with a heavy query or small load.

Column Meaning `Min/Max Duration` Fastest and slowest queries `Latency Percentile` Helps identify outliers and tail latency — critical for **SLA analysis** `Avg Duration` Mean time per query execution

✓ **Why These Panels Matter:**

Metric Importance **Query Duration** Spot slow queries and time-based performance issues **Prepared Queries** Check app/database efficiency **General Activity Table** Summary snapshot of load and latency distribution

□ **Minimum Recommended PostgreSQL Settings for pgBadger**

To make pgBadger effective, your logs must contain the right information. Here are the minimum recommended parameters to support rich pgBadger parsing:

```
ALTER SYSTEM SET log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h ';
ALTER SYSTEM SET log_min_duration_statement = 500;
ALTER SYSTEM SET log_checkpoints = on;
ALTER SYSTEM SET log_lock_waits = on;
ALTER SYSTEM SET log_statement = 'ddl';
```

✓ Why these?

- `log_line_prefix`: Adds session metadata to every log line
- `log_min_duration_statement`: Captures slow queries (≥ 500 ms)
- `log_checkpoints`: Shows write activity that affects performance
- `log_lock_waits`: Detects concurrency and contention issues
- `log_statement = 'ddl'`: Audits schema changes for deployment reviews

□ Best Practices for Using pgBadger in Production

Here's how to get the most out of pgBadger in your daily operations:

1. **Schedule daily reports** via cron or CI/CD to track changes over time
2. **Store reports securely** in a location accessible to developers and DBAs
3. **Share findings in team stand-ups** or retrospectives
4. **Track performance regressions** after deployments or schema changes
5. **Use version control** to compare pgBadger reports before and after optimization work

□ Why It Matters

PostgreSQL logs are rich with insights — but they're often unreadable in raw form. **pgBadger bridges that gap** by converting raw logs into a high-level view of your system's health.

- You see which queries dominate performance.
- You discover error patterns before users do.
- You spot growth in vacuum activity or client connection issues.
- You can back up performance claims with visual, historical evidence.

□ Instead of wasting hours scrolling through logs, let pgBadger **show you the story behind the data.**

□ Example Real-World Use Case

□ Imagine a production system starts to slow down after a schema migration.

□ pgBadger can instantly show:

- A spike in `ALTER TABLE` or `CREATE INDEX` usage
- Query runtimes increasing around deployment windows
- Lock wait events during peak traffic
- A decline in autovacuum efficiency post-migration

Instead of debugging blindly, you get an **instant timeline of behavior change** — visualized and filterable.

✓ Final Thoughts

If you're serious about PostgreSQL performance, **pgBadger is not optional — it's essential.**

It provides the observability PostgreSQL deserves, **without requiring a PhD in log parsing**.

- ✓ Easy to install
- ✓ Compatible with standard PostgreSQL logs
- ✓ Generates interactive, browser-friendly reports
- ✓ Open-source and actively maintained

□ If you've followed the 10 logging best practices already, **pgBadger is your 11th step — and your PostgreSQL superpower**.

12 □ Use Centralized Log Management Tools: PostgreSQL Logging Best Practices

By now, you've configured PostgreSQL to generate structured logs and may be using tools like `pgBadger` for visualization. That's great — but what happens when your infrastructure scales? What if you manage **multiple PostgreSQL servers**, or need to correlate database logs with application or system logs?

This is where **centralized log management tools** come in — the final step in building a **production-grade PostgreSQL logging pipeline**.

□ What Is Centralized Log Management?

Centralized log management is the process of **collecting, storing, analyzing, and monitoring logs** from multiple systems in one place — typically through a log aggregation platform.

Instead of manually checking logs on each server (or worse, logging into machines one-by-one), centralized logging lets you:

- Search across all logs instantly
- Visualize patterns and trends

- Trigger alerts on errors or anomalies
- Store logs securely and compliantly
- Correlate logs across services, databases, and environments

This is essential for modern systems, especially when using PostgreSQL with:

- Microservices
- Multi-node clusters (e.g., HA setups)
- Containerized deployments (e.g., Docker, Kubernetes)
- Hybrid or multi-cloud architectures

❑ **Why PostgreSQL Logs Need Centralization**

PostgreSQL logs hold a **treasure trove of insights**, but without centralization, they're trapped in silos.

✗ **The Problem:**

- Logs are stuck on individual servers
- Difficult to correlate database logs with app issues
- No global alerting or pattern detection
- Increased risk of log loss during server failures

✓ **The Solution:**

By shipping PostgreSQL logs to a centralized log platform, you gain:

- ❑ **Real-time visibility** into all PostgreSQL instances
- ❑ **Dashboards and charts** for query timing, lock waits, or error spikes

- ☐ **Correlated context** across your stack (web app + DB + API)
- ☐ **Instant alerts** on failures, performance regressions, or suspicious access
- ☐ **Reliable retention** and archival for audits or compliance

☐ **Recommended Centralized Logging Tools**

Here are some **industry-proven tools** you can use to manage PostgreSQL logs at scale:

☐ **1. ELK Stack (Elasticsearch, Logstash, Kibana)**

- **Elasticsearch** stores logs in a fast, searchable format
- **Logstash** parses and enriches PostgreSQL logs in real-time
- **Kibana** gives you beautiful dashboards to explore query durations, slow queries, error spikes, and more

✓ **Best for:** Open-source lovers, self-managed observability stacks

☐ **Integration:** Use Filebeat or Logstash to collect and ship logs from your PostgreSQL servers

☐ **2. Splunk**

- Enterprise-grade log aggregation and security monitoring
- Supports PostgreSQL via universal forwarders or REST APIs
- Advanced search language and machine learning models for anomaly detection

✓ **Best for:** Regulated industries, security-sensitive environments

☐ **Integration:** Use Splunk Forwarder or Syslog pipelines

☐ **3. Grafana Loki**

- Lightweight, high-performance log aggregation tool

- Designed for Kubernetes environments
- Seamlessly integrates with Grafana dashboards

✓ **Best for:** Cloud-native teams already using Prometheus + Grafana

□ **Integration:** Ship PostgreSQL logs via Promtail or Fluent Bit

□ 4. Cloud-Native Logging Services

AWS CloudWatch Logs

- Ideal for Amazon RDS / Aurora PostgreSQL
- Native support for metrics, dashboards, and alerts
- Easily integrated with CloudTrail and CloudWatch Alarms

Google Cloud Operations (formerly Stackdriver)

- GCP-native PostgreSQL logging with structured querying
- Alerting, retention, and visualization built in

✓ **Best for:** PostgreSQL deployed on managed cloud platforms

□ **Integration:** Use agents or enable native log exports

□ Sample PostgreSQL Configuration for Centralized Logging

To make PostgreSQL logs compatible with external log systems, enable file-based logging:

```
ALTER SYSTEM SET logging_collector = on;  
ALTER SYSTEM SET log_directory = 'log';  
ALTER SYSTEM SET log_filename = 'postgresql-%Y-%m-%d.log';  
ALTER SYSTEM SET log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h ';
```

Then, use a lightweight agent like **Filebeat**, **Fluent Bit**, or **Vector** to forward logs to your target system:

```
# Example Filebeat config snippet
filebeat.inputs:
  - type: log
    enabled: true
    paths:
      - /var/lib/pgsql/data/log/*.log
```

```
output.elasticsearch:
  hosts: ["http://localhost:9200"]
```

❑ **Best Practices for Centralized PostgreSQL Logging**

✓ **Tag logs with metadata**

Include instance ID, environment (prod/stage/dev), or region in log entries or shipping configurations. This makes filtering and alerting easier.

✓ **Avoid logging everything**

Set `log_min_messages = 'warning'` and `log_statement = 'ddl'` to reduce noise and improve relevance.

✓ **Configure slow query logging**

Use `log_min_duration_statement = 500` to capture only problematic queries.

✓ **Enable checkpoints and lock waits**

They reveal deep performance insights not found in standard logs.

✓ **Automate retention and cleanup**

Set data lifecycle rules for log storage — 30, 60, or 90 days depending on compliance and cost.

✓ **Monitor with alerts**

Trigger real-time alerts on query spikes, failed logins, connection floods, or replication errors.

❑ Real-World Scenario

Imagine you're running a SaaS platform with PostgreSQL, and a customer reports slowness around checkout.

With centralized logs:

- You search across all DB nodes for `duration > 1000ms`
- You correlate app logs showing increased load
- You spot a lock wait triggered by an index creation from a deployment
- You trigger a rollback or optimization instantly

Without centralization? You'd be guessing.

✓ Final Thoughts

As your infrastructure grows, so does the need for **holistic observability**. PostgreSQL's built-in logging is powerful, but without centralization, you're **missing the bigger picture**.

By using centralized logging tools like **ELK, Splunk, Loki**, or **CloudWatch**, you gain:

- ❑ Complete visibility across systems
 - ❑ Actionable insights from structured logs
 - ❑ Better compliance, traceability, and performance
 - ❑ Faster incident response and resolution
- ❑ Don't stop at logging — unlock PostgreSQL's full potential by making your logs **searchable, visual, and actionable**.

□ Verifying and Monitoring PostgreSQL Logs: Final Steps for Logging Success

PostgreSQL is a powerful database engine, but its true potential comes alive when you can **observe and understand its behavior in real time**. Throughout this series, we've covered how to configure PostgreSQL logging, rotate log files, detect slow queries, and integrate logs with tools like pgBadger and ELK.

But now it's time for something even more important:

□ How do you know if your logs are actually working?

In this post, we'll cover how to **verify PostgreSQL logging configuration**, where to find logs, how to monitor them in real time, and best practices for **protecting and retaining** log data.

□ Step 1: Verify Your PostgreSQL Logging Configuration

After changing logging settings using `ALTER SYSTEM` or updating `postgresql.conf`, it's crucial to confirm that your database is applying those settings.

You can run the following queries in `psql` to verify your current logging configuration:

```
SELECT current_setting('logging collector');
SELECT current_setting('log destination');
SELECT current_setting('log_filename');
SELECT current_setting('log_line_prefix');
SELECT current_setting('log_connections');
SELECT current_setting('log_statement');
SELECT current_setting('log_min_messages');
SELECT current_setting('log_duration');
SELECT current_setting('log_rotation_age');
SELECT current_setting('log_rotation_size');
```

```

postgres=#
postgres=# SELECT current_setting('logging_collector');
current_setting
-----
on
(1 row)

postgres=# SELECT current_setting('log_destination');
current_setting
-----
stderr
(1 row)

postgres=# SELECT current_setting('log_filename');
current_setting
-----
postgresql-%a.log
(1 row)

postgres=# SELECT current_setting('log_line_prefix');
current_setting
-----
%m [%p]
(1 row)

postgres=# SELECT current_setting('log_connections');
current_setting
-----
off
(1 row)

postgres=# SELECT current_setting('log_statement');
current_setting
-----
none
(1 row)

postgres=# SELECT current_setting('log_min_messages');
current_setting
-----
warning
(1 row)

postgres=# SELECT current_setting('log_duration');
current_setting
-----
off
(1 row)

postgres=# SELECT current_setting('log_rotation_age');
current_setting
-----
1d
(1 row)

postgres=# SELECT current_setting('log_rotation_size');
current_setting
-----
0
(1 row)

postgres=#

```

□ Why It Matters:

- **Misconfiguration is common.** A missing `SELECT` `pg_reload_conf();` or typo in the setting can silently disable logging.

- **Logging destination might be incorrect.** For example, if `log_destination` is set to `stderr` without enabling `logging_collector`, no log files will be written.

✓ **Best Practice:** Always verify each setting after applying changes — especially before a major release or performance audit.

□ **Step 2: Locate Your PostgreSQL Log Files**

Knowing where logs are being saved is the next key step.

Run this query to find the current log directory:

```
SELECT current_setting('log_directory');

postgres=# SELECT current_setting('log_directory');
current_setting
-----
log
(1 row)

postgres=#
```

This returns either a **relative path** (e.g., `log`) or an **absolute path** (e.g., `/var/log/postgresql`), depending on your `postgresql.conf` setup.

□ **Common Default Paths:**

- **Ubuntu/Debian (APT installations):**

```
/var/log/postgresql/
```

- **Red Hat/CentOS (YUM installations):**

```
/var/lib/pgsql/17/data/log
```

- **Custom or containerized setups:**

- Inside Docker volumes (e.g., `/var/lib/postgresql/data/log`)
- Cloud-native directories like `/rdsdbdata/log/` on AWS

✓ **Best Practice:** Always set `log_directory` explicitly in production environments and ensure your log shipping or rotation tools point to the correct path.

☐ **Step 3: Real-Time PostgreSQL Log Monitoring**

Sometimes, you need to **watch logs live**, especially when:

- Debugging a specific SQL query
- Monitoring app/database interaction during deployments
- Investigating login errors or permission issues
- Verifying DDL migrations or schema changes

☐ **Use `tail -f` to follow log files live:**

```
tail -100f /var/lib/pgsql/17/data/log/postgresql-Fri.log
```

This shows the **last 100 lines** of the log and continues to stream new entries as they appear.

☐ **If your logs go to syslog:**

```
tail -f /var/log/messages | grep postgres
```

```
[root@ip-172-31-92-46 log]# tail -f /var/log/messages | grep postgres  
Jul 11 01:58:28 ip-172-31-92-46 su[24271]: (to postgres) root on pts/2
```

You can use filters with `grep` to focus on specific entries:

```
tail -f postgresql-Fri.log | grep "duration"
```

```
[postgres@ip-172-31-92-46 log]$ tail -100f postgresql-Fri.log | grep "duration"  
2025-07-11 01:06:50.062 UTC [4712] LOG:  duration: 1328.619 ms  plan:
```

This would only show entries that include duration information — useful for performance analysis.

✓ **Best Practice:** Create aliases or helper scripts for your SRE/DBA team to instantly tail logs from production or staging servers.

□ **Step 4: Protect and Retain Your Logs**

PostgreSQL logs are not just operationally useful — they may also contain **sensitive data**, performance history, and forensic evidence. That makes it critical to **protect, rotate, and store them correctly**.

Here's how to do it right:

✓ **1. Rotate Logs Automatically**

Use PostgreSQL's built-in rotation settings:

```
ALTER SYSTEM SET log_rotation_age = '1d';  
ALTER SYSTEM SET log_rotation_size = 0;
```


This rotates logs **once per day**, regardless of file size. You can adjust `log_rotation_size` if your logs are unusually large.

☐ Example:

For high-traffic systems, consider `log_rotation_size = '100MB'` to rotate based on volume.

✓2. Retain Logs Based on Compliance or Audit Policies

Different industries have different requirements:

Industry Recommended Log Retention SaaS/Web Apps 14–30 days

Financial Services 90–180 days Healthcare / HIPAA 180+ days

Government / Legal 1+ year

☐ Send logs to a centralized service like:

- Amazon S3 or Glacier (with versioning)
- AWS CloudWatch Logs
- Azure Log Analytics
- On-prem archive (with daily compression)

✓3. Secure Log File Permissions

By default, PostgreSQL logs may be readable by other users on the system — **a security risk**.

Set restrictive file permissions:

```
chmod 600 postgresql-*.log
chown postgres:postgres postgresql-*.log
```

✓ Only the `postgres` system user should have read/write access to logs.

✓4. Archive and Compress Old Logs

Create cron jobs or use system tools to archive logs daily or weekly:

```
mv postgresql-2025-07-*.log /mnt/log archive/
gzip /mnt/log_archive/postgresql-2025-07-*.log

[root@ip-172-31-92-46 log]# mkdir -p /mnt/log_archive/

[root@ip-172-31-92-46 log]#
[root@ip-172-31-92-46 log]# cd /var/lib/pgsql/17/data/log

[root@ip-172-31-92-46 log]# ls -ltr
total 932
-rw-----. 1 postgres postgres 8959 Jul 10 23:26 postgresql-Thu.log
-rw-----. 1 postgres postgres 54039 Jul 11 01:11 postgresql-Fri.log
-rw-r--r--. 1 postgres postgres 884439 Jul 11 01:40 report.html

[root@ip-172-31-92-46 log]# mv postgresql-*.log /mnt/log archive/

[root@ip-172-31-92-46 log]# ls -ltr
total 864
-rw-r--r--. 1 postgres postgres 884439 Jul 11 01:40 report.html

[root@ip-172-31-92-46 log]# cd /mnt/log archive/

[root@ip-172-31-92-46 log_archive]# ls -ltr
total 68
-rw-----. 1 postgres postgres 8959 Jul 10 23:26 postgresql-Thu.log
-rw-----. 1 postgres postgres 54039 Jul 11 01:11 postgresql-Fri.log
[root@ip-172-31-92-46 log_archive]#

[root@ip-172-31-92-46 ~]# gzip /mnt/log_archive/postgresql-*.log
[root@ip-172-31-92-46 ~]#

[root@ip-172-31-92-46 ~]# cd /mnt/log_archive/

[root@ip-172-31-92-46 log_archive]# ls -ltr
total 8
-rw-----. 1 postgres postgres 1732 Jul 10 23:26 postgresql-Thu.log.gz
-rw-----. 1 postgres postgres 2940 Jul 11 01:11 postgresql-Fri.log.gz
[root@ip-172-31-92-46 log_archive]#
```

For enterprise environments, use log shipping tools like:

- **Filebeat / Fluentd** → ELK / Splunk / Loki
- **CloudWatch Agent** → AWS
- **Stackdriver Logging Agent** → GCP

✓ Archived logs are crucial for:

- Incident response
- Legal investigations
- Historical performance tuning

✓ Final Thoughts

Logs are not just output — they're evidence, telemetry, and context. They help you **understand, protect, and improve** your PostgreSQL environments.

By verifying, monitoring, and protecting your logs, you ensure that all previous logging configuration and tuning efforts **actually pay off in real-time operations**.

☐ Your logs are only as useful as they are available, secure, and understandable.

☐ What to Do Next

- ✓ Automate log verification as part of CI/CD DB pipelines
- ✓ Set up a dashboard (e.g., in Grafana or Kibana) to monitor logs visually
- ✓ Periodically test recovery from archived logs during incident simulations



✓ PostgreSQL Logging Best Practices: Summary Cheat Sheet & Final Thoughts

After walking through the what, why, and how of PostgreSQL logging — including configuration, visualization with pgBadger, real-time monitoring, and centralized logging — it's time to wrap things up with a **quick-reference guide** and key takeaways.

Press enter or click to view image in full size

Summary Cheat Sheet: PostgreSQL Logging Essentials

Below is a **condensed cheat sheet** you can bookmark, share with your team, or use in production environments to get PostgreSQL logging configured correctly — quickly and effectively.

 Best Practice	 Command
Enable logging	<code>ALTER SYSTEM SET logging_collector = 'on';</code>
Set log destination	<code>ALTER SYSTEM SET log_destination = 'stderr';</code>
Use readable filename format	<code>ALTER SYSTEM SET log_filename = 'postgresql_%A-%d-%B_%H%M';</code>
Add structured log line prefix	<code>ALTER SYSTEM SET log_line_prefix = 'time=%t, pid=%p, ...';</code>
Log all connections	<code>ALTER SYSTEM SET log_connections = '1';</code>
Focus on DDL/DML changes only	<code>ALTER SYSTEM SET log_statement = 'mod';</code>
Rotate logs daily	<code>ALTER SYSTEM SET log_rotation_age = '1d';</code>

✓ **Pro Tip:** Don't forget to apply your changes by running:

```
SELECT pg_reload_conf();
```

This reloads your PostgreSQL configuration without needing a full database restart.

☐ Final Thoughts & Conclusion

☐ PostgreSQL logs are your window into the engine.

When configured thoughtfully, they reveal what queries are slowing you down, when users are connecting, what changes are being made to your schema — and where your performance or security posture might be slipping.

Too often, logs are overlooked until something breaks.

But with the right logging strategy, you can:

- ✓ **Troubleshoot faster** — get to root causes without guesswork
- ✓ **Detect problems before users do** — with slow query logging, connection tracking, and alerts
- ✓ **Improve performance** — by identifying expensive queries and locking issues
- ✓ **Support compliance** — by retaining logs that show who did what, and when

□ Logging isn't just a DBA task — it's a foundational observability strategy for any production-grade PostgreSQL setup.