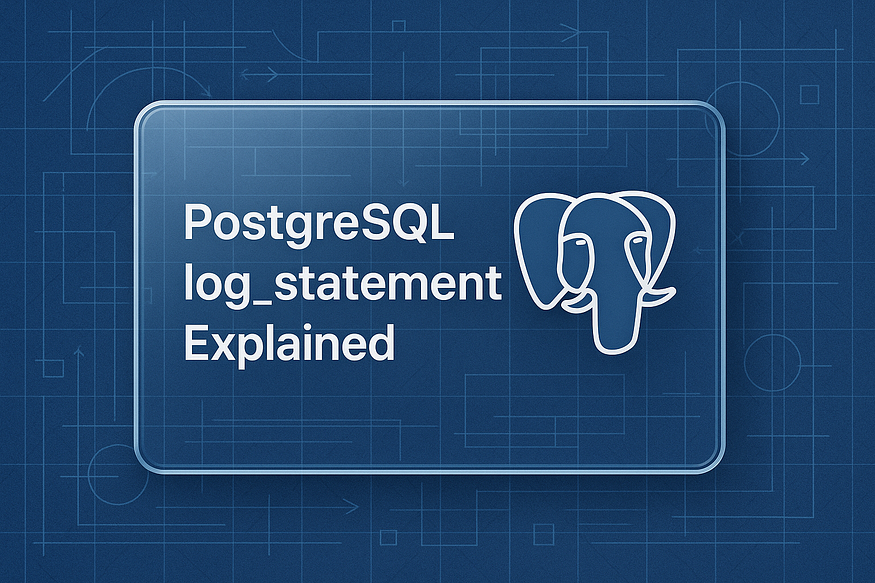
# **PostgreSQL**log\_statement**Explained: Complete Guide With Real-Time Examples**



## **📘 Understanding**log\_statement**in PostgreSQL: Your Key to Query Visibility**

PostgreSQL is celebrated not only for its reliability and performance — but also for its ****robust logging system****. Whether you’re troubleshooting performance issues, tracking user activity, or performing compliance audits, ****query-level logging**** is often your first and most powerful line of insight.

One of the ****most critical logging parameters**** in PostgreSQL — yet surprisingly underutilized — is log\_statement.

In this article, we’ll explore:

* What log\_statement does
* How to configure it in PostgreSQL 13+
* Its different logging levels and when to use them
* Real-world examples that show how this parameter can make or break your observability strategy

Let’s dive in.

## **🔍 What is**log\_statement**?**

log\_statement is a PostgreSQL configuration parameter that ****controls which SQL statements get logged**** to the PostgreSQL log file.

Think of it as a filter that determines ****how verbose**** the query logs should be. It’s incredibly helpful for:

* Monitoring application behavior
* Debugging complex interactions
* Auditing query patterns for compliance
* Detecting unexpected or malicious SQL activity

But with great power comes great responsibility — setting it too aggressively can flood your logs with noise, while setting it too narrowly may cause you to miss important insights.

## **⚙️ How to Configure**log\_statement

You can set log\_statement in:

* The postgresql.conf file (permanent)
* Via SQL using ALTER SYSTEM (requires reload)
* Temporarily per session (for debugging)

## **Example (in**postgresql.conf**):**

log\_statement = 'all'

Then reload the server for changes to take effect:

SELECT pg\_reload\_conf();

## **🧰 The Four Logging Levels**

PostgreSQL offers ****four options**** for log\_statement, each increasing in verbosity:

## **1.**none**(default)**

* No SQL statements are logged.
* Best for production systems with separate query logging tools.

## **2.**ddl

* Logs only DDL statements such as CREATE, ALTER, DROP.
* Useful for tracking ****schema changes**** without the noise of everyday queries.

## **3.**mod

* Logs all DDL plus ****INSERT****, ****UPDATE****, ****DELETE****, and ****TRUNCATE****.
* Great for tracking ****data mutations**** without logging every SELECT.

## **4.**all

* Logs ****everything****, including every SELECT.
* Ideal for ****debugging**** or during ****security reviews****, but can generate large logs quickly.

🧠 ****Tip****: Pair log\_statement with log\_duration or log\_min\_duration\_statement to capture slow queries only.

## **💡 Real-World Example**

Let’s say you set:

log\_statement = 'mod'

Then run the following:

SELECT \* FROM customers; -- not logged  
UPDATE customers SET active = false WHERE last\_login < now() - interval '1 year'; -- logged

Only the UPDATE statement appears in your logs, because log\_statement = 'mod' includes data-modifying queries, but ****excludes read-only SELECTs****.

Now change it to:

log\_statement = 'all'

Rerun the SELECT—now it’s logged too.

This gives you full traceability of what’s happening in your database at the query level.

## **🔐 Security & Compliance Use Cases**

If you’re in a ****regulated environment**** (e.g., healthcare, fintech), log\_statement can play a key role in:

* ****Auditing who did what****
* ****Tracking schema changes****
* ****Detecting unauthorized access patterns****

You can even combine it with ****log\_line\_prefix**** for detailed, timestamped logs with user info, session ID, and application name.

## **🚀 Final Thoughts**

The log\_statement parameter might seem like a minor setting, but it’s a ****powerful gateway to understanding your PostgreSQL workload****.

By setting the right logging level, you can:

* Debug faster
* Audit smarter
* Monitor more effectively

Whether you’re chasing a runaway query or analyzing long-term behavior trends, log\_statement gives you the ****raw query data**** you need to stay in control.

## **🔎 What Is**log\_statement**in PostgreSQL?**

When it comes to understanding what’s happening inside your PostgreSQL database, logging is your best friend. One of the most useful — but often overlooked — logging features is the log\_statement parameter.

## **💡 So, what is it?**

The log\_statement parameter in PostgreSQL ****controls which SQL statements are written to the server log****. It acts as a filter to determine how much query-related information should be recorded.

This setting is especially valuable for:

* ✅ ****Debugging****: Want to trace what a client or app is doing behind the scenes? This logs the actual SQL.
* ✅ ****Auditing****: Need to track who changed what and when? Enable detailed logs.
* ✅ ****Compliance****: In regulated industries (e.g., finance, healthcare), logging query history is a requirement.

With log\_statement, you decide ****how much detail**** gets logged—from nothing at all to every single SQL command.

## **⚙ Default Behavior of**log\_statement

Out of the box, PostgreSQL is tuned for ****performance****, not verbosity. That’s why the default setting is:

log\_statement = 'none'

## **Here’s what this means:**

* ✅ ****No queries are logged**** to the PostgreSQL log file
* ✅ ****Better performance****, since there’s minimal I/O overhead from logging
* 🚫 ****Not ideal for troubleshooting****, since you can’t see what queries were run
* 🚫 ****Useless for audit trails****, as there’s no visibility into user actions

This default makes sense for high-throughput environments where performance is critical — but for development, staging, or security-sensitive environments, you’ll likely want to ****increase the logging level****.

## **📌 When Should You Change It?**

If you’re:

* Debugging application behavior
* Monitoring data changes
* Investigating performance issues
* Auditing database usage

Then it’s time to consider a more verbose setting like 'mod' or 'all'.

We’ll cover the available options (none, ddl, mod, all) and how to apply them in the next section.

## **🔧**log\_statement**Options in PostgreSQL (With Use Cases)**

PostgreSQL gives you granular control over how much SQL logging you want by offering multiple options for the log\_statement parameter. Each level captures a different category of SQL statements, allowing you to strike the right balance between visibility and performance.

Let’s explore each option in detail — with real-world use cases to help you choose the right setting for your environment.

## **1️⃣**none**— No SQL statements are logged**

log\_statement = 'none' -- Default setting

* 🧘 ****Description****: Completely disables query-level logging.
* ⚙️ ****Performance****: Very lightweight and optimal for high-throughput systems.
* 🚫 ****Drawback****: You won’t see any SQL activity in your logs, which makes debugging or auditing extremely difficult.

### **✅ Use Case:**

* Ideal for ****production environments**** where performance is a top priority and other monitoring tools (like pgBadger or APMs) are already in place.
* Not suitable when troubleshooting or when visibility into query behavior is required.

## **2️⃣**ddl**— Logs only DDL (schema) statements**

log\_statement = 'ddl'

* 🧱 ****Description****: Logs ****Data Definition Language**** commands like CREATE, ALTER, DROP, and so on.
* 🔍 ****Visibility****: Tracks structural changes without overwhelming the logs with everyday queries.

### **✅ Use Case:**

* Great for ****schema change auditing****, especially in teams where multiple developers work on the same database.
* Helps during ****migrations or upgrades**** to ensure no unintended schema modifications occur.

## **3️⃣**mod**— Logs DDL + modifying DML statements**

log\_statement = 'mod'

* ✏️ ****Description****: Logs all DDL commands *plus* ****data-modifying**** operations like INSERT, UPDATE, DELETE, and TRUNCATE.
* 🔄 ****Tracking Changes****: Gives you insight into any command that changes the state of your data.

### **✅ Use Case:**

* Perfect for ****auditing data changes**** in staging or QA environments.
* Useful for ****security reviews**** where tracking who changed what data (and when) is essential.
* Strikes a good balance between visibility and log size for many production environments.

## **4️⃣**all**— Logs everything, including SELECT**

log\_statement = 'all'

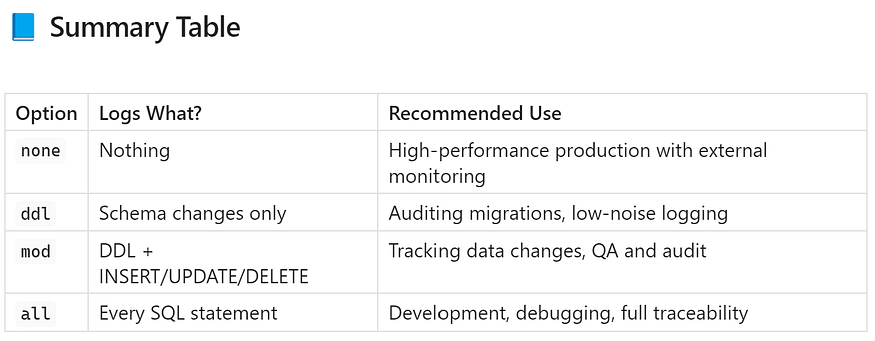
* 🧠 ****Description****: The most verbose option — logs ****every single SQL statement****, including read-only operations like SELECT.
* 🔎 ****Full Transparency****: Allows complete visibility into query behavior, execution patterns, and user actions.

### **✅ Use Case:**

* Best for ****debugging**** application behavior or performance issues in development environments.
* Valuable during ****code testing, profiling, or training environments**** where full query traceability is needed.

⚠️ ****Warning****: This setting can generate ****very large log files****, especially in read-heavy workloads. Use with caution in production environments.

Press enter or click to view image in full size



## **✅ Final Thoughts**

Choosing the right log\_statement level is all about ****your environment’s needs****. Use none when performance matters most, mod for balanced auditing, and all when you want to see everything during development or debugging.

PostgreSQL gives you the flexibility — ****you just need to tune it wisely****.

## **💻 How**log\_statement**Works in PostgreSQL**

Every query executed in PostgreSQL passes through its internal ****logging pipeline****. However, ****only queries that match the log\_statement level**** get written to the PostgreSQL server logs.

This filtering mechanism helps database administrators and developers fine-tune how much visibility they need — without overwhelming the system with unnecessary log volume.

## **⚙️ How to Set**log\_statement

There are two main ways to configure the log\_statement parameter:

1. ****Via postgresql.conf file (permanent setting)****  
   Located typically at:

/var/lib/pgsql/{version}/data/postgresql.conf

1. You’ll need to edit the file and restart or reload PostgreSQL.
2. ****Dynamically with SQL (no restart needed)****  
   Use the following SQL command for immediate effect:

ALTER SYSTEM SET log\_statement = 'mod'; -- Or 'ddl', 'all', etc.   
SELECT pg\_reload\_conf(); -- Apply the change

## **📁 Where Are PostgreSQL Logs Stored?**

By default, PostgreSQL writes its log files to:

/var/lib/pgsql/{version}/data/log/postgresql.log

You can monitor logs in real time using:

[postgres@ip-172-31-20-155 ~]$ cd /var/lib/pgsql/17/data/log/  
[postgres@ip-172-31-20-155 log]$  
[postgres@ip-172-31-20-155 log]$ ls -ltr  
total 8  
-rw-------. 1 postgres postgres 2519 Jun 21 16:23 postgresql-Sat.log  
-rw-------. 1 postgres postgres 0 Jun 22 00:00 postgresql-Sun.log  
-rw-------. 1 postgres postgres 817 Jun 23 23:30 postgresql-Mon.log  
-rw-------. 1 postgres postgres 0 Jun 24 00:00 postgresql-Tue.log  
[postgres@ip-172-31-20-155 log]$  
  
[postgres@ip-172-31-20-155 log]$ cat postgresql-Mon.log  
2025-06-23 23:28:57.501 UTC [22503] ERROR: invalid input syntax for type inet: "your\_ip\_here" at character 75  
2025-06-23 23:28:57.501 UTC [22503] STATEMENT: SELECT COUNT(\*) AS client\_conns  
 FROM pg\_stat\_activity  
 WHERE client\_addr = 'your\_ip\_here';  
2025-06-23 23:29:37.317 UTC [22503] ERROR: column "query\_time" does not exist at character 24  
2025-06-23 23:29:37.317 UTC [22503] HINT: Perhaps you meant to reference the column "pg\_stat\_activity.query\_id".  
2025-06-23 23:29:37.317 UTC [22503] STATEMENT: SELECT  
 pid, usename, query\_time, query  
 FROM pg\_stat\_activity  
 WHERE state = 'active'  
 AND (now() - query\_start) > interval '5 minutes';  
2025-06-23 23:30:58.439 UTC [22503] ERROR: column "pid" does not exist at character 26  
2025-06-23 23:30:58.439 UTC [22503] STATEMENT: SELECT pg\_cancel\_backend(pid);  
[postgres@ip-172-31-20-155 log]$

This is incredibly useful during development, testing, or incident investigations.

## **🔬 Hands-On Examples: How Different**log\_statement**Levels Behave**

Now let’s see how PostgreSQL behaves under different log\_statement settings. These examples will help you ****visualize exactly what does and doesn’t get logged****.

## **📌 Example 1:**log\_statement = 'none'**(Default)**

Check the current setting:

postgres=# SHOW log\_statement;  
 log\_statement  
---------------  
 none

Run some SQL commands:

CREATE TABLE course (course\_no integer, name text, price numeric);  
INSERT INTO course (course\_no, name, price) VALUES (1, 'postgres', 150);  
SELECT \* FROM course;

✅ ****Result:****  
None of these statements will appear in the logs. This is expected because log\_statement = 'none' logs ****nothing****.

Check the logs to confirm:

tail -30f /var/lib/pgsql/17/data/log/postgresql-Tue.log

You won’t see any entries related to the above queries.

## **📌 Example 2:**log\_statement = 'ddl'**(Schema Changes Only)**

Now let’s enable logging for DDL statements (schema changes):

ALTER SYSTEM SET log\_statement = 'ddl';  
SELECT pg\_reload\_conf(); -- Reload without restart

postgres=# ALTER SYSTEM SET log\_statement = 'ddl';  
ALTER SYSTEM  
postgres=#  
postgres=# SELECT pg\_reload\_conf();  
 pg\_reload\_conf  
----------------  
 t  
(1 row)  
  
postgres=#

Run the same set of queries:

CREATE TABLE course (course\_no integer, name text, price numeric);  
INSERT INTO course (course\_no, name, price) VALUES (1, 'databricks', 2000);  
SELECT \* FROM course;

postgres=# CREATE TABLE course (course\_no integer, name text, price numeric);  
CREATE TABLE  
postgres=#  
postgres=#  
postgres=# INSERT INTO course (course\_no, name, price) VALUES (1, 'databricks', 2000);  
INSERT 0 1  
postgres=#  
postgres=# SELECT \* FROM course;  
 course\_no | name | price  
-----------+------------+-------  
 1 | databricks | 2000  
(1 row)  
  
postgres=#

✅ ****Result:****  
Only the CREATE TABLE command will be logged. The INSERT and SELECT statements will not.

Confirm by checking the logs:

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

You should see a log entry similar to:

[postgres@ip-172-31-20-155 log]$ tail -30f /var/lib/pgsql/17/data/log/postgresql-Tue.log  
2025-06-24 00:39:22.566 UTC [22935] ERROR: syntax error at or near "]" at character 1  
2025-06-24 00:39:22.566 UTC [22935] STATEMENT: ]  
 ALTER SYSTEM SET log\_statement = 'ddl';  
2025-06-24 00:39:33.477 UTC [1823] LOG: received SIGHUP, reloading configuration files  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_vacuum\_scale\_factor" changed to "0.05"  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_analyze\_scale\_factor" changed to "0.02"  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_max\_workers" cannot be changed without restarting the server  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_naptime" changed to "30s"  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "log\_statement" changed to "ddl"  
2025-06-24 00:39:33.478 UTC [1823] LOG: configuration file "/var/lib/pgsql/17/data/postgresql.auto.conf" contains errors; unaffected changes were applied  
2025-06-24 00:40:16.029 UTC [22935] LOG: statement: CREATE TABLE course (course\_no integer, name text, price numeric);

But no entries for the INSERT or SELECT.

## **🧠 Why This Matters**

* Setting the right log\_statement level gives you ****precise control**** over your query visibility.
* It enables ****auditing and troubleshooting**** without flooding your logs.
* Knowing what gets logged (and what doesn’t) helps you ****interpret logs more accurately**** during real-world operations.

In the next part, we’ll look at more examples using mod and all to see how you can log data-changing and read-only queries too.

## **📌 Example 3:**log\_statement = 'mod'**— Logging DDL and Modifying DML**

If you want to capture ****data-changing activity**** in your PostgreSQL logs without logging every SELECT, the mod setting is your go-to option.

Let’s enable it:

ALTER SYSTEM SET log\_statement = 'mod';  
SELECT pg\_reload\_conf(); -- Applies the change without restart

Now, run the following SQL commands:

INSERT INTO course (course\_no, name, price) VALUES (2, 'rds mysql', 5000);  
DELETE FROM course WHERE course\_no = 2;

You should see a log entry similar to:

postgres=# ALTER SYSTEM SET log\_statement = 'mod';  
ALTER SYSTEM  
postgres=#  
postgres=# INSERT INTO course (course\_no, name, price) VALUES (2, 'rds mysql', 5000);  
INSERT 0 1  
postgres=#  
postgres=#  
postgres=# DELETE FROM course WHERE course\_no = 2;  
DELETE 1  
postgres=#  
postgres=#

## **✅ Result:**

Both the INSERT and DELETE operations will be logged, because they modify data.

However, if you run a SELECT query like:

SELECT \* FROM course;

postgres=# SELECT \* FROM course;  
course\_no | name | price  
-----------+------------+-------  
 1 | databricks | 2000  
(1 row)  
  
postgres=#  
postgres=#

It ****will not**** be logged — because mod doesn’t log read-only operations.

## **📝 Summary:**

* ✔️ ****Logs****: INSERT, UPDATE, DELETE, TRUNCATE, and all DDL (e.g., CREATE, ALTER)
* ❌ ****Does not log****: SELECT (unless it triggers a data-modifying function)

Use this setting if you want visibility into ****data mutations**** without the performance hit of logging every single query.

## **📌 Example 4:**log\_statement = 'all'**— Logging Every Query**

Need full visibility into every query hitting your PostgreSQL instance? Set log\_statement to all:

ALTER SYSTEM SET log\_statement = 'all';  
SELECT pg\_reload\_conf(); -- Reload the config

postgres=# SELECT \* FROM course;  
course\_no | name | price  
-----------+------------+-------  
 1 | databricks | 2000  
(1 row)  
  
postgres=#  
postgres=# ALTER SYSTEM SET log\_statement = 'all';  
ALTER SYSTEM  
postgres=#  
postgres=# SELECT pg\_reload\_conf();  
 pg\_reload\_conf  
----------------  
 t  
(1 row)  
  
postgres=#

Now, run a few sample queries:

INSERT INTO course (course\_no, name, price) VALUES (3, 'oracle', 10000);  
SELECT \* FROM course;

postgres=# INSERT INTO course (course\_no, name, price) VALUES (3, 'oracle', 10000);  
INSERT 0 1  
postgres=#  
postgres=# SELECT \* FROM course;  
course\_no | name | price  
-----------+------------+-------  
 1 | databricks | 2000  
 1 | oracle | 10000  
(2 rows)  
  
postgres=#  
postgres=#

## **✅ Result:**

Every query — yes, even simple SELECT statements—will be written to the PostgreSQL logs.

Check it with:

tail -30f /var/lib/pgsql/17/data/log/postgresql-Tue.log

You’ll see log entries like:

2025-06-24 00:39:22.566 UTC [22935] ERROR: syntax error at or near "]" at character 1  
2025-06-24 00:39:22.566 UTC [22935] STATEMENT: ]  
 ALTER SYSTEM SET log\_statement = 'ddl';  
2025-06-24 00:39:33.477 UTC [1823] LOG: received SIGHUP, reloading configuration files  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_vacuum\_scale\_factor" changed to "0.05"  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_analyze\_scale\_factor" changed to "0.02"  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_max\_workers" cannot be changed without restarting the server  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "autovacuum\_naptime" changed to "30s"  
2025-06-24 00:39:33.478 UTC [1823] LOG: parameter "log\_statement" changed to "ddl"  
2025-06-24 00:39:33.478 UTC [1823] LOG: configuration file "/var/lib/pgsql/17/data/postgresql.auto.conf" contains errors; unaffected changes were applied  
2025-06-24 00:40:16.029 UTC [22935] LOG: statement: CREATE TABLE course (course\_no integer, name text, price numeric);  
2025-06-24 00:43:21.637 UTC [22992] LOG: statement: ALTER SYSTEM SET log\_statement = 'mod';  
2025-06-24 00:44:29.489 UTC [1825] LOG: checkpoint starting: time  
2025-06-24 00:44:34.711 UTC [1825] LOG: checkpoint complete: wrote 53 buffers (0.3%); 0 WAL file(s) added, 0 removed, 0 recycled; write=5.216 s, sync=0.002 s, total=5.223 s; sync files=34, longest=0.001 s, average=0.001 s; distance=222 kB, estimate=272 kB; lsn=0/156D1D0, redo lsn=0/156D178  
2025-06-24 00:44:55.863 UTC [22992] LOG: statement: ALTER SYSTEM SET log\_statement = 'all';  
2025-06-24 00:45:01.315 UTC [1823] LOG: received SIGHUP, reloading configuration files  
2025-06-24 00:45:01.316 UTC [1823] LOG: parameter "autovacuum\_max\_workers" cannot be changed without restarting the server  
2025-06-24 00:45:01.316 UTC [1823] LOG: parameter "log\_statement" changed to "all"  
2025-06-24 00:45:01.316 UTC [1823] LOG: configuration file "/var/lib/pgsql/17/data/postgresql.auto.conf" contains errors; unaffected changes were applied  
2025-06-24 00:45:19.968 UTC [22992] LOG: statement: INSERT INTO course (course\_no, name, price) VALUES (3, 'mongodb', 170);  
2025-06-24 00:45:28.648 UTC [22992] LOG: statement: SELECT \* FROM course;  
2025-06-24 00:49:29.417 UTC [1825] LOG: checkpoint starting: time  
2025-06-24 00:49:29.524 UTC [1825] LOG: checkpoint complete: wrote 2 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.101 s, sync=0.001 s, total=0.108 s; sync files=2, longest=0.001 s, average=0.001 s; distance=0 kB, estimate=245 kB; lsn=0/156D420, redo lsn=0/156D3C8

## **⚠️ Warning:**

Using log\_statement = 'all' provides ****maximum visibility****, which is great for debugging, audits, and development environments. ****But in production, be careful****:

* 🚨 Can ****quickly bloat log files**** in read-heavy applications
* 💥 May ****impact performance**** due to disk I/O overhead

## **🧠 Best Practice:**

Use log\_statement = 'all':

* In ****development****, for deep debugging
* In ****test environments****, for auditing or query pattern analysis
* With ****short-term time limits****, to avoid excessive log generation

## **🔚 Wrapping Up This Section**

With these hands-on examples, you’ve now seen how PostgreSQL reacts to each log\_statement level:

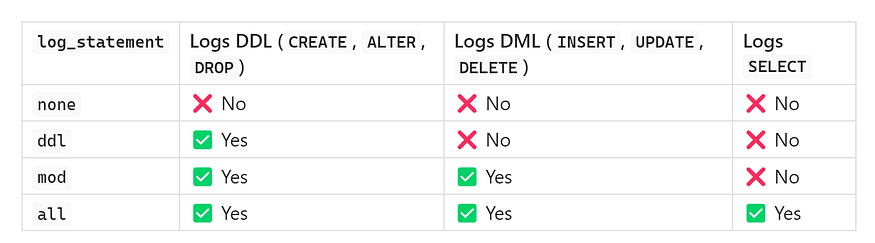
Setting Logs What? Ideal For none Nothing High-performance prod environments ddl Schema changes Change tracking & auditing mod DDL + data-modifying DML Data mutation monitoring all Every SQL statement Debugging, full traceability

## **🧮 Quick Comparison: What Does Each**log\_statement**Level Capture?**

When choosing the right log\_statement level, it helps to see a ****side-by-side comparison**** of what each setting actually logs.

Here’s a quick reference table:

Press enter or click to view image in full size



This table gives you a clear understanding of how much query activity each level captures — so you can choose one that matches your observability needs without compromising performance.

## **🔐 Where Should You Use**log\_statement**?**

The ideal setting for log\_statement depends on ****your environment**** and ****your goals****. Here’s a breakdown of common environments and the recommended setting for each:

****Environment**** ****Recommended Setting**** ****Why?**** ****Development**** all Full visibility into all queries for debugging and learning ****UAT / QA**** mod Capture schema and data changes, skip noisy SELECTs ****Production**** none or ddl Prioritize performance, optionally log schema changes for traceability ****Compliance / Auditing**** mod Ensure all DDL and DML operations are tracked for audit purposes

By aligning the log\_statement level with the purpose of your environment, you avoid unnecessary noise while still capturing the information that matters.

## **📝 Conclusion: Fine-Tuning PostgreSQL Logging for Visibility and Performance**

The log\_statement parameter is one of PostgreSQL’s most ****powerful logging tools****. It gives you full control over how much query activity you want to observe, track, or audit—without requiring any external plugins.

Whether you’re debugging, auditing, or protecting performance, log\_statement helps you balance:

* 🔎 ****Query visibility**** — know what’s running and when
* 📊 ****Auditability**** — trace who changed what and why
* ⚙️ ****Performance**** — avoid log bloat in high-throughput environments

## **🎯 Key Takeaway:**

There’s no one-size-fits-all setting. Use log\_statement ****strategically****, adjusting it based on your current workload, environment, and operational needs.