ARTICLE

MySQL Slow Query Log Tuning for Performance Troubleshooting

The slow query log is one of the most practical tools for troubleshooting MySQL performance, but only when it is tuned to surface useful evidence instead of noise. This article explains when to enable it, which settings matter, how to interpret the output, and what to verify before using it in production.

Databases - July 24, 2026

Key takeaways

The MySQL slow query log is most useful when you need evidence, not guesses. Tuned correctly, it helps you identify queries that exceed a latency threshold, correlate them with load or lock pressure, and separate true bottlenecks from harmless background traffic.

The main tuning decisions are about signal quality and operational overhead. You need to decide which sessions to capture, what threshold counts as ?slow? in your environment, whether to record queries that miss indexes, and how long to keep the log enabled before reviewing the data.

The log is not a fix by itself. It is a diagnostic tool that works best when paired with execution plan review, schema checks, and workload context. If you use it without a validation workflow, it can create a large amount of data without leading to a useful conclusion.

Why this matters during performance troubleshooting

When a MySQL system becomes slow, the first symptom is often broad: application timeouts, queue buildup, rising CPU, or increased lock waits. Those symptoms do not tell you which SQL statements are responsible. The slow query log fills that gap by recording queries that cross a defined threshold, letting you see actual workload behavior instead of relying on application assumptions.

This matters operationally because the most expensive query is not always the most frequent one. A statement that runs rarely may still stall user-facing transactions if it holds locks, scans large tables, or forces repeated disk reads. For teams that already use MySQL query performance tuning for high-load databases, the slow query log is often the first evidence source that confirms where the tuning effort should start.

Used correctly, it helps answer three practical questions: which SQL statements are consuming time, whether the delay is due to execution time or lock time, and whether the slowdown is systemic or isolated to a narrow workload pattern.

How the slow query log works

The slow query log records statements that satisfy one or more thresholds, most commonly execution time above long_query_time. It can also capture statements that examine too many rows, or those that do not use an index when configured to do so. In MySQL, the details you collect depend on the server variables you choose and the logging format you enable.

At a minimum, the log gives you the query text, timing information, and contextual metadata such as timestamp and user. In more verbose configurations, it can also show rows examined, lock time, and query properties that help you decide whether the problem is the SQL itself, the index design, or contention elsewhere in the system.

That said, the slow query log is only as useful as the threshold you choose. If the threshold is too low, the log becomes noisy and can hide the real outliers. If it is too high, you miss the queries that are hurting user experience but do not look severe in aggregate.

Tuning the log for useful signal

The most important tuning variable is long_query_time. This should reflect the latency level that is operationally meaningful in your environment, not an arbitrary industry number. For a low-latency OLTP system, even a query taking 200 ms may be worth investigating. For a reporting workload, the threshold may need to be higher to avoid drowning in expected analytical activity.

A second useful control is whether to include statements that do not use indexes through log_queries_not_using_indexes. This can surface missing-index problems quickly, but it can also generate excessive noise if you have legitimate small-table scans or short-lived maintenance activity. Enable it only when you are actively investigating access path issues and be prepared to disable it after the diagnostic window.

The log output format also matters. A structured format is easier to parse, aggregate, and compare across time windows, especially if you feed the data into analysis tools. An unstructured format may still be enough for manual inspection during an incident, but it becomes harder to trend or automate.

Finally, consider whether to log all sessions or only selected workloads. In busy systems, capturing everything can be expensive and may produce many statements that are not relevant to the issue you are chasing. Narrowing the scope, when possible, reduces noise and helps you preserve attention for the queries that actually affect service latency.

Practical workflow for using the slow query log

This workflow is intentionally compact because the goal is not long-term logging. It is to create a controlled evidence window that shows which statements deserve deeper analysis. If you are troubleshooting a high-load environment, the same approach complements query-level review and plan validation rather than replacing them.

What this means in practice

A common production scenario is an application that reports occasional response-time spikes while database CPU remains moderate. The slow query log shows a small number of statements with high rows examined and lock time, but no single statement dominates total runtime. That pattern usually means the issue is not one catastrophic query, but a mix of inefficient access paths, lock contention, and workload timing.

In another environment, you may see a single statement appearing repeatedly with modest execution time but high cumulative impact because it runs very often. In that case, the right action is not to raise the threshold and ignore it. Instead, treat frequency as part of the cost model. Small inefficiencies can matter more than large rare queries when they sit on the critical path of a busy application.

A third pattern is a log full of administrative or batch statements that are slow by design. That does not mean the log is wrong; it means the threshold and capture window do not match the operational question. In those cases, narrow the logging window to user traffic or a known incident period so you do not confuse expected batch work with production degradation.

Decision guidance: when to enable, narrow, or skip it

Use the slow query log when you need a time-bounded record of expensive statements and you do not yet know which queries are responsible. It is especially useful during incidents, after a release, or when a small number of workload changes may have altered query patterns.

Narrow the logging scope when the system is busy enough that full capture would create too much noise or overhead. If your issue is isolated to one service, tenant, or maintenance window, logging everything is usually less useful than focusing on the affected slice of traffic.

Skip or minimize the slow query log when you already have direct evidence from application tracing, plan regression analysis, or a clear lock-wait diagnosis. In those cases, the log can still confirm the finding, but it should not replace a more specific investigative path.

If you need to go beyond logging and analyze why a query is slow, execution plan review is the next layer of evidence. That is where you confirm whether the query is scanning too many rows, using the wrong join order, or being blocked by schema design. The log identifies the candidate; the plan explains the cause.

Common mistakes that reduce the value of the log

The most common mistake is leaving the threshold too low for too long. A very low long_query_time may seem comprehensive, but it often creates a log that is too large to interpret and may obscure the important statements.

Another mistake is assuming that a slow query log entry automatically means the SQL text is the root cause. A query can be slow because of lock waits, stale statistics, or an access path change after a deployment. Always separate symptom from cause before making a change.

Teams also often collect logs without defining the validation window. If you do not know when the issue started, whether the workload was representative, or which business function was affected, the log can be difficult to interpret in context.

A final mistake is leaving verbose capture settings enabled after the incident is over. That can create unnecessary overhead and long-term noise. The slow query log is most effective as a focused troubleshooting instrument, not as a permanent substitute for performance observability.

Implementation trade-offs to consider

Tuning the slow query log is always a balance between visibility and overhead. More capture gives you more diagnostic detail, but it also increases storage use and the time required to analyze the data. Less capture lowers operational cost but increases the risk of missing the statement that triggered the issue.

There is also a trade-off between simplicity and precision. A single threshold is easy to reason about, but it does not capture all forms of pain equally well. A statement that is slightly above the threshold and frequent may be more important than a statement that is far above the threshold but rare. That is why total impact matters as much as per-query latency.

Another important trade-off is how aggressively you treat non-indexed statements. Logging them can expose schema problems quickly, but it may also produce a large volume of queries that are intentionally scanning small tables or temporary data sets. Use that setting as a short-lived investigative aid, not as a permanent always-on detector.

For broader performance investigations, it is often useful to combine the slow query log with application-side timing and server metrics. The log tells you what SQL was slow; metrics tell you whether the broader environment was also constrained by I/O, CPU, or contention.

Production readiness checklist

Before enabling the slow query log in production, verify the following:

- The threshold matches the operational objective, not a generic default.
- You know whether the issue is latency, lock time, or total query cost.
- The capture window is short and representative.
- Log volume is acceptable for your storage and analysis process.
- Any settings that increase verbosity are deliberately temporary.
- You have a plan to review the log against execution plans and schema metadata.
- You know how to disable or narrow capture after collecting enough evidence.
- The team understands which workload slice is under investigation.

If you need to validate the execution side of a suspected query issue after log collection, plan analysis is often the fastest way to confirm whether the database is scanning, joining, or sorting inefficiently. That is where the evidence becomes actionable rather than merely descriptive.

Final takeaway

MySQL slow query log tuning is about collecting enough evidence to identify the statements that matter without burying yourself in noise. The best results come from a clear threshold, a short and representative capture window, and a disciplined review process that checks execution plans, lock time, and workload context before changing production behavior. When used this way, the slow query log becomes a precise troubleshooting tool rather than a passive archive of slow SQL.

Use this guidance together with Always On Availability Groups troubleshooting to connect the workflow with related operational context already available on the site.

Use this guidance together with Transparent Data Encryption to connect the workflow with related operational context already available on the site.