Performance Tuning & Monitoring

Query Optimization, Indexing, and Observability in MySQL

Why Performance Tuning Matters

Database performance directly impacts application latency, user experience, and infrastructure cost. Inefficient queries and poor indexing strategies can cause exponential load growth, leading to bottlenecks even on powerful hardware.

Scope of This Module

This session covers query optimization with EXPLAIN, indexing strategies, slow query detection, Performance Schema monitoring, buffer pool tuning, partitioning, and modern monitoring tools used in production systems.

Target Audience

Designed for database engineers, backend developers, and SREs responsible for maintaining high-performance MySQL systems at scale, including OLTP and analytical workloads.

Query Optimization with EXPLAIN

Understanding How MySQL Executes Queries

Purpose of EXPLAIN

EXPLAIN reveals how MySQL plans to execute a query, including join order, index usage, access methods, and estimated rows scanned. It is the primary diagnostic tool for identifying inefficient queries.

Key Output Columns

Important fields include select_type, table, type, possible_keys, key, rows, filtered, and Extra. Together, these describe whether indexes are used, how data is accessed, and where bottlenecks may exist.

Interpreting Execution Cost

The rows and filtered columns indicate how much data MySQL expects to scan and discard. High row counts or type = ALL often signal missing or ineffective indexes.

Index Access Types in EXPLAIN

From Constant-Time Lookups to Full Table Scans

- **Best-Case Access Types:** system and const represent constant-time access where MySQL reads at most one row, typically using PRIMARY KEY or UNIQUE indexes. These access types scale extremely well even under high concurrency.
- **Indexed Row Lookups:** eq_ref and ref indicate indexed access returning one or multiple matching rows. These are optimal for joins and filtered queries when foreign keys and frequently queried columns are properly indexed.
- **Range to Full Scan:** range performs index range scans for BETWEEN or inequality predicates, while index and ALL indicate full index or table scans. type = ALL is the worst case and usually signals missing or misordered indexes.

Creating Effective Indexes

Single-Column and Composite Index Strategies

Single-Column Indexes

Single-column indexes are effective when queries filter, join, or sort primarily on one column. They should be created on frequently used WHERE, JOIN, ORDER BY, and GROUP BY columns to minimize scanned rows.

Composite Index Design

Composite indexes index multiple columns in a defined order. MySQL can only use the leftmost prefix, making column order critical. Equality conditions should appear before range or sorting columns.

Common Index Pitfalls

Indexes are not used when leading columns are skipped, functions are applied to indexed columns, or predicates do not match index order. Poorly designed indexes increase write overhead without improving reads.

Advanced Indexing Techniques

Full-Text and Prefix Index Optimization

- **Full-Text Indexes:** Full-text indexes enable efficient natural language search on large text columns. They are optimized for MATCH ... AGAINST queries and significantly outperform LIKE '%keyword%' patterns on large datasets.
- **Boolean Mode Searches:** Using BOOLEAN MODE allows advanced query control with operators such as +, -, and *. This supports relevance-based searches while maintaining performance at scale.
- **Prefix Indexes:** Prefix indexes store only the first N characters of large string columns. They reduce index size and memory usage while remaining effective for high-cardinality prefixes such as URLs or email domains.

Identifying Slow Queries

Using Slow Query Logs for Performance Diagnosis

Slow Query Log

The slow query log captures SQL statements that exceed a defined execution time threshold. Enabling it provides direct visibility into real performance issues occurring in production workloads.

Configuring Thresholds

The long_query_time parameter controls which queries are logged. Setting it too high hides issues, while setting it too low generates noise. A common starting point is 1 second for OLTP systems.

Analyzing Slow Queries

Tools like mysqldumpslow aggregate slow query logs to identify the most frequent and time-consuming queries, enabling prioritization of optimization efforts.

Query Optimization Best Practices

Reducing I/O, CPU, and Execution Time

- **Avoid SELECT *:** Selecting unnecessary columns increases I/O, memory usage, and network transfer. Queries should explicitly request only required columns to reduce execution cost and improve cache efficiency.
- **LIMIT and OFFSET Pitfalls:** Large OFFSET values force MySQL to scan and discard rows. Rewriting queries to use indexed WHERE conditions with LIMIT significantly improves performance on large tables.
- **Predicate and Join Optimization:** Avoid functions on indexed columns in WHERE clauses, prefer UNION ALL when duplicates are acceptable, and ensure join columns are indexed to minimize nested loop costs.

Performance Schema Overview

Built-in Observability for MySQL Internals

- **What Is Performance Schema:** Performance Schema is a low-level instrumentation framework that collects detailed metrics about query execution, waits, I/O, locks, and memory usage directly inside the MySQL server.
- **Instrumentation Model:** Metrics are captured via instruments and exposed through consumers. This design allows fine-grained control over what is monitored, balancing observability depth against runtime overhead.
- **Why It Matters:** Unlike logs or external monitoring, Performance Schema provides real-time, structured insight into internal behavior, making it essential for diagnosing complex performance and concurrency issues.

Table & Index I/O Statistics

Identifying Hot Tables and Unused Indexes

Table I/O Metrics
Performance Schema tracks read, write, insert, update, and delete operations per table. High total access counts highlight hot tables that are critical candidates for optimization.

Index Usage Analysis
Index-level statistics reveal how often each index is read or written. Indexes with zero reads and writes are strong candidates for removal to reduce memory and write overhead.

Optimization Impact
Focusing tuning efforts on the most frequently accessed tables and indexes yields disproportionate performance gains compared to optimizing rarely used objects.

Query & Lock Monitoring

Detecting Bottlenecks and Concurrency Issues

Query Execution Statistics
Performance Schema aggregates queries by execution count and total wait time. High-frequency or high-latency statements are primary optimization targets due to their cumulative impact on system load.

Digest-Based Analysis
Query digests normalize similar SQL statements, enabling identification of problematic query patterns even when literals differ. This is critical for tuning ORM-generated workloads.

Lock Contention Visibility
Lock wait summaries expose tables and transactions involved in blocking scenarios. Persistent lock contention often indicates missing indexes, long transactions, or suboptimal isolation levels.

InnoDB Buffer Pool Configuration

Memory Allocation for High-Performance Workloads

Role of the Buffer Pool
The InnoDB buffer pool caches data pages and indexes in memory, reducing disk I/O. Its size is the single most important configuration parameter for MySQL performance.

Sizing Guidelines
A common rule of thumb is to allocate 60–80% of available system RAM to the buffer pool on dedicated database servers, balancing cache efficiency against OS and connection overhead.

Concurrency Optimization
Multiple buffer pool instances reduce contention on large memory pools. Chunk size controls incremental memory allocation and should align with instance count for predictable scaling.

Monitoring the InnoDB Buffer Pool

Measuring Cache Efficiency and Memory Health

- **Buffer Pool Statistics:** INFORMATION_SCHEMA and SHOW STATUS expose metrics such as free buffers, dirty pages, and database pages. These values indicate memory pressure and write-back behavior.
- **Hit Ratio Calculation:** The buffer pool hit ratio measures how often data is served from memory instead of disk. Values above 99% indicate healthy caching for most OLTP workloads.
- **Operational Signals:** Low hit ratios, rapidly growing dirty pages, or consistently low free buffers suggest insufficient memory allocation or suboptimal query patterns.

Table Partitioning Strategies

Managing Large Tables at Scale

- **Range Partitioning:** Range partitioning splits data based on value ranges such as dates or numeric intervals. It is especially effective for time-series data where queries target recent periods and benefit from partition pruning.
- **Hash Partitioning:** Hash partitioning distributes rows evenly across partitions using a hash function. This improves parallelism and avoids hotspots when access patterns are evenly distributed by key.
- **Operational Benefits:** Partitioning enables faster queries through pruning, simpler maintenance by dropping or archiving old partitions, and improved cache efficiency due to smaller working sets.

Monitoring Tools Overview

Operational Visibility Beyond the Database Engine

- **MySQL Workbench:** MySQL Workbench provides a graphical interface for connection management, query execution, schema design, and basic performance monitoring, making it suitable for development and troubleshooting workflows.
- **Prometheus and Grafana:** Prometheus collects time-series metrics from MySQL exporters, while Grafana visualizes them through dashboards. This stack is widely used in cloud-native and containerized environments.
- **ELK Stack Integration:** The ELK stack centralizes MySQL logs such as slow query logs in Elasticsearch, enabling advanced search, correlation, and alerting across distributed systems.

Key Takeaways

MySQL Performance Tuning & Monitoring Summary

- **Query Optimization:** Use EXPLAIN to understand execution plans, eliminate full table scans, avoid SELECT *, rewrite inefficient pagination, and ensure queries are written to leverage indexes effectively.
- **Indexing Strategy:** Design single and composite indexes based on real query patterns, respect column order, remove unused indexes, and apply advanced techniques such as full-text and prefix indexing where appropriate.
- **Monitoring and Architecture:** Leverage Performance Schema for deep visibility, size and monitor the InnoDB buffer pool correctly, apply partitioning for large tables, and integrate external monitoring tools for continuous observability.