

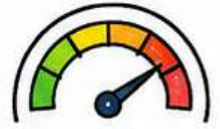
Chapter 29: MySQL Query Optimization

29.1 Query Optimization

- * Query optimization is the process of **improving the performance** of SQL queries to **reduce execution time** and **resource usage**.
- * Efficient queries help in **faster data retrieval** and **better application performance**.
- * Optimization involves writing **efficient queries**, using **indexes**, avoiding **unnecessary operations** and understanding how MySQL executes queries.

Key Points :

- ✓ Optimization improves query speed and reduces server load.
- ✓ Good query design leads to efficient applications.
- ✓ Proper indexing and avoiding redundant work are key to optimization.



Performance



Optimize



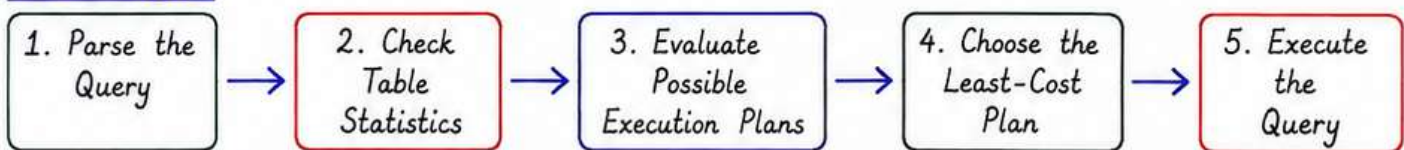
29.2 MySQL Query Optimizer

- * MySQL Query Optimizer is a component that determines the **most efficient** way to execute a query.
- * It analyzes the query, available indexes, table statistics and chooses the **best execution plan**.
- * The optimizer aims to **reduce the overall cost** of query execution.



Query Optimizer

How it Works :



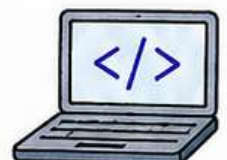
Optimizer Considers :

Factor	Description
Indexes	Availability and selectivity of indexes on columns.
Table Statistics	Number of rows, data distribution and value uniqueness.
Query Structure	Joins, where conditions, group by, order by, etc.
System Variables	Configuration settings that affect query execution.
Cost Estimation	Optimizer calculates cost of each plan and chooses the least-cost plan.



Key Points :

- ✓ The optimizer does not guarantee the best possible plan always.
- ✓ Keeping statistics up-to-date helps the optimizer make better choices.
- ✓ Use **EXPLAIN** statement to see how the optimizer will execute a query.



29.3 EXPLAIN

- * **EXPLAIN** statement is used to obtain information about how MySQL executes a **SELECT** query.
- * It shows the **execution plan** chosen by the optimizer without actually executing the query.
- * It helps in understanding **performance bottlenecks** and improving queries.

Syntax :

```
EXPLAIN SELECT column_list FROM table_name WHERE condition;
```

Output Columns :

Column	Description
id	Query block number.
select_type	Type of SELECT (SIMPLE, PRIMARY, SUBQUERY, etc.).
table	Table used in the query.
type	Join type (ALL, index, range, ref, eq_ref, const, system).
possible_keys	Indexes that MySQL may use.
key	Index actually used.
key_len	Length of the key used.
ref	Columns or constants compared to key.
rows	Estimated number of rows examined.
Extra	Additional information (Using where, Using index, Using filesort, etc.).

Key Points :

- ✓ EXPLAIN does not execute the query, it only shows the plan.
- ✓ Focus on **type**, **key**, **rows** and **Extra** columns for analysis.
- ✓ Aim for **type** as **ref**, **eq_ref** or **const** for better performance.



Execution Plan



Understand Query Performance



29.4 EXPLAIN ANALYZE

- * **EXPLAIN ANALYZE** executes the query and provides **actual execution statistics** along with the execution plan.
- * It shows **actual time taken**, **number of rows processed** and **loops**.
- * It helps in identifying **real performance issues**.

Syntax :

```
EXPLAIN ANALYZE SELECT column_list FROM table_name WHERE condition;
```

Example :

```
EXPLAIN ANALYZE SELECT * FROM employees  
WHERE department_id = 10;
```

Output Shows :

- Actual time spent in each step
- Actual rows read and returned
- Number of loops for each operation

Key Points :

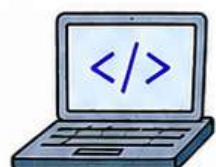
- ✓ EXPLAIN shows estimated plan; **EXPLAIN ANALYZE** shows actual execution.
- ✓ Very useful for complex queries and tuning.
- ✓ Helps verify if the optimizer's plan is efficient in real scenarios.



Actual Execution Statistics



Real Performance Insight



29.5 Execution Plans

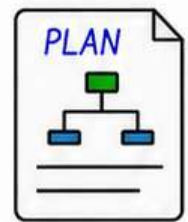
- * Execution plan shows how MySQL will execute a query.
- * It helps to identify **performance issues** and **bottlenecks**.
- * By analyzing the execution plan, we can **optimize queries** for better performance.

Syntax :

```
EXPLAIN SELECT column_list FROM table_name WHERE condition;
```

Common Plan Types (type column) :

Type	Description
system	Table has only one row (const table).
const	Table has at most one matching row, accessed by primary key.
eq_ref	Primary key or unique index lookup for each row from previous table.
ref	Non-unique index lookup.
range	Rows are retrieved from a given range using an index.
index	Full index scan.
ALL	Full table scan (worst performance).



Execution Plan



Analyze Plan



Better Performance

Key Points :

- ✓ Focus on **type**, **key**, **rows** and **Extra** columns in the plan.
- ✓ Use indexes to change bad plans (ALL) into good plans (ref, range, etc.).
- ✓ A good execution plan means **less cost** and **faster** query execution.



29.6 Full Table Scans

- * A full table scan (type = ALL) means MySQL reads every row in a table to find the result.
- * It is the **least efficient** way to execute a query.
- * It can be very **slow** for large tables.

Causes :

- No index on the columns used in WHERE clause.
- Using functions on indexed columns.
- Leading wildcard (%) in LIKE condition.
- OR conditions on different columns without proper indexes.
- Selecting * (all columns) when not needed.

How to Avoid :

- Create indexes on columns used in WHERE, JOIN, ORDER BY, GROUP BY.
- Avoid functions on indexed columns.
- Avoid leading wildcard in LIKE searches.
- Select only required columns.
- Use proper query design.



Full Table Scan



Optimize Queries

Key Points :

- ✓ Full table scans are expensive and slow.
- ✓ Use indexes to limit the number of rows scanned.
- ✓ Always analyze queries using **EXPLAIN** and optimize them.
- ✓ Good indexing and query design can eliminate full table scans.



29.7 Index Optimization

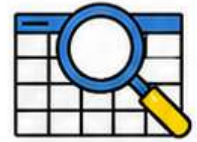
- * Indexes improve the **speed** of data retrieval operations.
- * Proper use of indexes can greatly improve query performance.
- * However, too many indexes can **slow** down INSERT, UPDATE and DELETE operations.

Best Practices :

Practice	Description
Index on WHERE Columns	Create indexes on columns frequently used in WHERE clause.
Index on JOIN Columns	Index columns used in JOIN conditions.
Composite Indexes	Use composite indexes for queries filtering on multiple columns.
Avoid Over-Indexing	Avoid creating indexes on columns with low selectivity (e.g., gender, status).
Covering Indexes	Include all required columns in an index to avoid table lookups.
Regular Maintenance	Rebuild or optimize indexes periodically.



Index Structure



Faster Search



Better Performance

Key Points :

- ✓ Use indexes wisely for better query performance.
- ✓ Index columns used in **WHERE**, **JOIN**, **ORDER BY** and **GROUP BY**.
- ✓ Avoid indexing columns with low cardinality.
- ✓ Maintain indexes for optimal performance.



29.8 JOIN Optimization

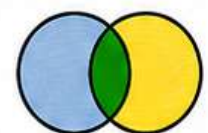
- * **JOINS** combine rows from two or more tables based on **related columns**.
- * Optimizing **JOINS** can significantly improve query performance.
- * The **order of tables** and **indexes** on JOIN columns are very important.

Tips for JOIN Optimization :

- Ensure **JOIN** columns are indexed.
- Use **INNER JOIN** instead of **OUTER JOIN** when possible.
- Filter data as early as possible (use **WHERE** first).
- Avoid SELECT *; select only required columns.
- Join smaller tables first.

Example :

```
SELECT e.name, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.id
WHERE e.status = 'Active';
```



JOIN Tables



Optimized JOIN

Key Points :

- ✓ Always index columns used in **JOIN** conditions.
- ✓ Use **INNER JOIN** and filter early for better performance.
- ✓ Avoid unnecessary columns and large intermediate result sets.
- ✓ Optimize **JOIN** order and query structure for efficiency.



Efficient Results

29.9 Subquery Optimization

- * Subqueries are queries nested inside another query.
- * Improperly written subqueries can cause **poor performance**.
- * Optimizing subqueries can reduce **execution time** and improve efficiency.

Best Practices :

Practice	Description
Use JOINS Instead of Subqueries	JOINS are often faster than subqueries.
Use EXISTS Instead of IN	EXISTS stops scanning once a match is found, IN may scan the entire result set.
Limit Subquery Results	Ensure subquery returns only necessary rows (use LIMIT or proper filters).
Avoid Correlated Subqueries	Correlated subqueries run once for each row and can be very slow .
Use Indexes in Subqueries	Index the columns used in WHERE , JOIN or comparison in subqueries.

Key Points :

- ✓ Prefer **JOINS** over subqueries for better performance.
- ✓ Use **EXISTS** instead of **IN** in most cases.
- ✓ Keep subqueries simple and return only required data.
- ✓ Proper indexing is crucial for subquery optimization.



Subquery



Analyze



Better Performance



29.10 GROUP BY Optimization

- * **GROUP BY** is used to group rows that have the same values.
- * Optimizing **GROUP BY** can reduce sorting and temporary table usage.
- * It improves **aggregation** performance.

Best Practices :

- Use indexes on **GROUP BY** columns.
- Avoid selecting unnecessary columns.
- Filter data early using **WHERE** clause.
- Avoid using functions on **GROUP BY** columns.
- Ensure only required aggregates are calculated.

Example :

```
SELECT department_id, COUNT(*)  
FROM employees  
WHERE status = 'Active'  
GROUP BY department_id;
```



Aggregation



Optimize



Smarter Queries



Efficient Aggregation

Common Issues and Solutions :

Issue	Solution
Using functions on GROUP BY columns	Avoid functions on columns, use raw columns.
Huge data before filtering	Apply WHERE clause before GROUP BY .
No index on GROUP BY columns	Create index on GROUP BY columns.
Selecting unnecessary columns	Select only GROUP BY columns and aggregates.

Key Points :

- ✓ Index columns used in **GROUP BY** for faster grouping.
- ✓ Filter data before grouping to reduce data size.
- ✓ Avoid functions and unnecessary columns in **GROUP BY**.
- ✓ Proper **GROUP BY** usage leads to less sorting and faster results.

29.11 ORDER BY Optimization

- * **ORDER BY** is used to sort the result set.
- * Sorting large result sets can be **expensive**.
- * Proper **indexing** and query design can optimize **ORDER BY** performance.

Best Practices :

Practice	Description
Use Indexes for Sorting	Create index on columns used in ORDER BY to avoid filesort (extra sorting).
Match Index Order	Index order should match ORDER BY (ASC/DESC).
Avoid Sorting Unnecessary Rows	Use WHERE clause to filter data before ORDER BY .
Avoid Sorting Large Text/Blob Columns	Sorting on large text fields is slow. Sort on indexed, smaller columns.
Use LIMIT with ORDER BY	When using LIMIT , MySQL can stop sorting after retrieving required rows.

Key Points :

- ✓ Use indexes on **ORDER BY** columns.
- ✓ Ensure index order matches the sort order.
- ✓ Filter data early and sort less.
- ✓ Use **LIMIT** with **ORDER BY** to reduce sorting workload.



Sorting Results



Indexed Sorting



Faster Sorting



Better Performance

29.12 Pagination Optimization

- * Pagination is used to **split large result sets** into smaller pages.
- * Inefficient pagination can cause **slow performance** for high offset values.
- * Proper pagination techniques can **improve query speed** and user experience.

Common Approach :

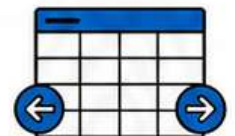
Example (Using LIMIT & OFFSET) :

- Use **LIMIT** and **OFFSET**.
- Avoid large **OFFSET** values.
- Use indexed columns in **ORDER BY**.
- Use keyset pagination for large datasets.

```
SELECT id, name, email
FROM users
ORDER BY id
LIMIT 20 OFFSET 40; -- Page 3 (skip first 40 rows)
```



Pagination



Page Navigation

Better Approach - Keyset (Seek) Pagination :

- Instead of **OFFSET**, use the last seen value.
- Works faster for large datasets.
- Requires indexed columns.



```
SELECT id, name, email
FROM users
WHERE id > 40 -- Last seen id
ORDER BY id
LIMIT 20;
```



Better User Experience

Key Points :

- ✓ Use **LIMIT** with caution; large **OFFSET** values slow down queries.
- ✓ Use indexed columns in **ORDER BY**.
- ✓ For large datasets, prefer keyset pagination.
- ✓ Pagination improves performance and user experience.



Efficient Queries

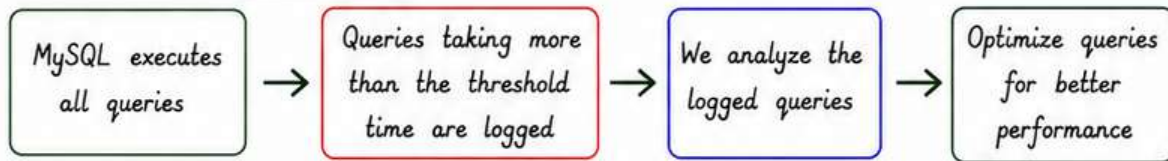
29.13 Slow Query Log

- * Slow Query Log records queries that take **more time** than a specified threshold.
- * It helps in **identifying slow queries** that degrade performance.
- * Analyzing slow queries helps in **optimization**.



Slow Query Log

How Slow Query Log Works :



Find Slow Queries

Enable Slow Query Log :

```
SET GLOBAL slow_query_log = 'ON';  
SET GLOBAL long_query_time = 2; -- Time in seconds  
SET GLOBAL slow_query_log_file = '/var/lib/mysql/slow.log';
```

View Slow Queries :

```
SHOW VARIABLES LIKE 'slow_query_log%';  
SHOW VARIABLES LIKE 'long_query_time';  
SHOW GLOBAL STATUS LIKE 'Slow_queries';
```



Improve Performance

Key Points :

- ✓ Helps find queries that are taking more time.
- ✓ Set appropriate **long_query_time** to avoid logging too many queries.
- ✓ Regularly review slow log and optimize queries.
- ✓ Use indexes, query rewriting and proper schema design.



Optimize Queries

29.14 Performance Schema

- * Performance Schema is a built-in feature in MySQL for **monitoring server performance**.
- * It provides detailed **statistics** about queries, locks, waits, I/O and more.
- * It helps in **troubleshooting** and **optimizing** database performance.



Performance Schema

Key Features :

Feature	Description
Statement Performance	Tracks execution time and resource usage of SQL statements.
Wait Events	Shows where the server is waiting (I/O, locks, CPU, etc.).
Memory Instrumentation	Provides memory usage statistics.
Table I/O Statistics	Tracks read/write operations on tables.
User and Thread Statistics	Monitors connection, threads and user activity.



Monitor & Analyze

Example Queries :

```
-- Top 10 slowest queries  
SELECT DIGEST_TEXT, COUNT_STAR, SUM_TIMER_WAIT  
FROM performance_schema.events_statements_summary_by_digest  
ORDER BY SUM_TIMER_WAIT DESC  
LIMIT 10;  
  
-- Top wait events  
SELECT EVENT_NAME, COUNT_STAR, SUM_TIMER_WAIT  
FROM performance_schema.events_waits_summary_global_by_event_name  
ORDER BY SUM_TIMER_WAIT DESC  
LIMIT 10;
```

Enable Performance Schema :

```
-- Check if Performance Schema is enabled  
SHOW VARIABLES LIKE 'performance_schema';  
  
-- It is enabled by default in  
modern MySQL versions.
```

Key Points :

- ✓ Performance Schema helps in deep analysis.
- ✓ Use it to find bottlenecks and slow components.
- ✓ Monitor queries, waits, locks and I/O.
- ✓ Helps in tuning and capacity planning.



Better Insights



Optimize Performance