MySql : Explain

Let's refine the lesson with MySQL compatibility in mind. MySQL's `EXPLAIN` command is used to analyze query execution plans. We'll include MySQL-specific syntax, behavior, and outputs.

Lesson Plan: Using `EXPLAIN` in MySQL for Query Optimization

---

1. Introduction

The `EXPLAIN` command in MySQL reveals how queries will be executed, including details about table access, join strategies, index usage, and cost. It's a critical tool for performance tuning.

---

2. Schema Design

We will use the following **Employee** and **Project** schema, tailored for MySQL:

```
-- Employee Table
CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    DepartmentID INT,
    Salary DECIMAL(10, 2),
    ManagerID INT,
    JoiningDate DATE,
    INDEX idx_department (DepartmentID),
    INDEX idx_salary (Salary)
);

-- Project Table
CREATE TABLE Project (
    ProjectID INT PRIMARY KEY,
    ProjectName VARCHAR(100),
    DepartmentID INT,
    Budget DECIMAL(15, 2),
    INDEX idx_department_budget (DepartmentID, Budget)
);

-- Sample Data
INSERT INTO Employee (EmployeeID, Name, DepartmentID, Salary, ManagerID, JoiningDate) VALUES
(1, 'Alice', 101, 60000, NULL, '2020-01-15'),
(2, 'Bob', 101, 55000, 1, '2021-06-10'),
(3, 'Charlie', 102, 50000, 1, '2022-03-01');

INSERT INTO Project (ProjectID, ProjectName, DepartmentID, Budget) VALUES
(1, 'AI Development', 101, 200000),
(2, 'Web Redesign', 102, 150000);

```

---

3. Examples with Missed Tuning Opportunities and Fixes

Each example includes:

1. Original query.
2. MySQL `EXPLAIN` output highlighting inefficiencies.
3. Fix with explanation.
4. Optimized `EXPLAIN` output.

Example 1: Full Table Scan on Non-Indexed Column

Query: Find employees in department `101`.

```
EXPLAIN SELECT * FROM Employee WHERE DepartmentID = 101;

```

Output (Expected):

```
| id | select_type | table     | type  | possible_keys | key  | rows | Extra       |
|----|-------------|-----------|-------|---------------|------|------|-------------|
|  1 | SIMPLE      | Employee  | ALL   | NULL          | NULL |    3 | Using where |

```

Issue:

- The `type` is `ALL`, indicating a full table scan, as no index is used.

Fix: Ensure `DepartmentID` is indexed:

```
CREATE INDEX idx_department ON Employee(DepartmentID);
EXPLAIN SELECT * FROM Employee WHERE DepartmentID = 101;

```

Optimized Output:

```
| id | select_type | table     | type   | possible_keys | key          | rows | Extra       |
|----|-------------|-----------|--------|---------------|--------------|------|-------------|
|  1 | SIMPLE      | Employee  | ref    | idx_department| idx_department|    1 | Using where |

```

Explanation:

- Indexing allows MySQL to use the `ref` access type, significantly reducing the number of rows scanned.

Example 2: Suboptimal Join Order

Query: Retrieve employee names and project names.

```
EXPLAIN SELECT E.Name, P.ProjectName
FROM Employee E
JOIN Project P ON E.DepartmentID = P.DepartmentID;

```

Output (Expected):

```
| id | select_type | table     | type  | possible_keys | key  | rows | Extra       |
|----|-------------|-----------|-------|---------------|------|------|-------------|
|  1 | SIMPLE      | Employee  | ALL   | NULL          | NULL |    3 |             |
|  1 | SIMPLE      | Project   | ALL   | NULL          | NULL |    2 | Using where |

```

Issue:

- Both tables perform a full table scan (`ALL` type).

Fix: Add indexes on `DepartmentID` in both tables:

```
CREATE INDEX idx_department_employee ON Employee(DepartmentID);
CREATE INDEX idx_department_project ON Project(DepartmentID);

EXPLAIN SELECT E.Name, P.ProjectName
FROM Employee E
JOIN Project P ON E.DepartmentID = P.DepartmentID;

```

Optimized Output:

```
| id | select_type | table     | type   | possible_keys          | key          | rows | Extra       |
|----|-------------|-----------|--------|------------------------|--------------|------|-------------|
|  1 | SIMPLE      | Employee  | ALL    | idx_department_employee| idx_department_employee | 3 |             |
|  1 | SIMPLE      | Project   | ref    | idx_department_project | idx_department_project  | 1 | Using where |

```

Explanation:

- The join now uses the indexed `DepartmentID`, reducing rows scanned for `Project`.

Example 3: Inefficient Sorting

Query: Get the top 2 highest-paid employees.

```
EXPLAIN SELECT * FROM Employee ORDER BY Salary DESC LIMIT 2;

```

Output (Expected):

```
| id | select_type | table     | type  | possible_keys | key  | rows | Extra             |
|----|-------------|-----------|-------|---------------|------|------|-------------------|
|  1 | SIMPLE      | Employee  | ALL   | NULL          | NULL |    3 | Using filesort    |

```

Issue:

- The `Extra` column shows `Using filesort`, which is costly for large datasets.

Fix: Add an index for `Salary` in descending order:

```
CREATE INDEX idx_salary_desc ON Employee(Salary DESC);
EXPLAIN SELECT * FROM Employee ORDER BY Salary DESC LIMIT 2;

```

Optimized Output:

```
| id | select_type | table     | type   | possible_keys  | key             | rows | Extra       |
|----|-------------|-----------|--------|----------------|-----------------|------|-------------|
|  1 | SIMPLE      | Employee  | index  | idx_salary_desc| idx_salary_desc|    2 |             |

```

Explanation:

- The query uses an index scan, avoiding the costly filesort operation.

Example 4: Suboptimal Filtering in a Subquery

Query: Get employees working on projects with a budget > $150,000.

```
EXPLAIN SELECT Name
FROM Employee
WHERE DepartmentID IN (
    SELECT DepartmentID FROM Project WHERE Budget > 150000
);

```

Output (Expected):

```
| id | select_type    | table     | type  | possible_keys | key  | rows | Extra       |
|----|----------------|-----------|-------|---------------|------|------|-------------|
|  1 | PRIMARY        | Employee  | ALL   | NULL          | NULL |    3 | Using where |
|  2 | DEPENDENT SUBQ | Project   | ALL   | NULL          | NULL |    2 | Using where |

```

Issue:

- The subquery is not optimized, causing a dependent subquery execution for each row.

Fix: Rewrite the query as a join:

```
EXPLAIN SELECT E.Name
FROM Employee E
JOIN Project P ON E.DepartmentID = P.DepartmentID
WHERE P.Budget > 150000;

```

Optimized Output:

```
| id | select_type | table     | type   | possible_keys | key          | rows | Extra       |
|----|-------------|-----------|--------|---------------|--------------|------|-------------|
|  1 | SIMPLE      | Employee  | ALL    | NULL          | idx_department_employee | 3 |             |
|  1 | SIMPLE      | Project   | ref    | idx_department_project | idx_department_project  | 1 | Using where |

```

Explanation:

- Using a join avoids executing the subquery multiple times.

Example 5: Aggregation Without Index

Query: Calculate total salary for department `101`.

```
EXPLAIN SELECT SUM(Salary) FROM Employee WHERE DepartmentID = 101;
```

```
```

Output (Expected):

```
| id | select_type | table     | type  | possible_keys | key  | rows | Extra       |
|----|-------------|-----------|-------|---------------|------|------|-------------|
|  1 | SIMPLE      | Employee  | ALL   | NULL          | NULL |    3 | Using where |
```

Issue:

- A full table scan is required for aggregation.

Fix: Create a composite index for `DepartmentID` and `Salary`:

```
CREATE INDEX idx_department_salary ON Employee(DepartmentID, Salary);
EXPLAIN SELECT SUM(Salary) FROM Employee WHERE DepartmentID = 101;
```

Optimized Output:

```
| id | select_type | table     | type   | possible_keys        | key                   | rows | Extra       |
|----|-------------|-----------|--------|----------------------|-----------------------|------|-------------|
|  1 | SIMPLE      | Employee  | ref    | idx_department_salary| idx_department_salary|    1 | Using index |
```

Explanation:

- The composite index enables MySQL

to efficiently filter and aggregate data.

---

4. Summary

- Use `EXPLAIN` regularly to identify query inefficiencies.
- Optimize joins, filters, and sorting with indexes.
- Rewrite subqueries when needed for better performance.
- Leverage composite indexes for multi-column filtering.

Detailed Notes on `select_type` and `type` in MySQL's EXPLAIN Output

When analyzing query performance using MySQL's `EXPLAIN` statement, two critical columns in the output are `select_type` and `type`. These provide insights into how a query is executed and the level of optimization.

---

1. `select_type`

The `select_type` column indicates the type of SELECT operation being performed. It classifies queries based on their complexity, such as whether the query involves simple SELECTs, subqueries, or joins.

Common Values of `select_type`:

1. **SIMPLE**
   - A straightforward SELECT query with no subqueries or unions.

- **Example:**

```
SELECT * FROM Employee WHERE DepartmentID = 101;

```

- **Explanation:** This is a basic SELECT statement with no nested queries or additional clauses.

---

2. **PRIMARY**
   - The outermost SELECT in a query containing subqueries.
   - **Example:**

```
SELECT Name FROM Employee WHERE DepartmentID IN (
    SELECT DepartmentID FROM Project WHERE Budget > 150000
);

```

   - **Explanation:** The outer SELECT is tagged as PRIMARY, while the inner SELECT is classified as a subquery.

---

3. **SUBQUERY**
   - A SELECT within a WHERE or HAVING clause of another query.
   - **Example:**

```
SELECT Name FROM Employee WHERE DepartmentID IN (
    SELECT DepartmentID FROM Project WHERE Budget > 150000
);

```

   - **Explanation:** The inner query is labeled as SUBQUERY, showing that it runs for each row evaluated by the outer query.

---

4. **DERIVED**
   - A SELECT that generates a temporary table (a derived table) for the outer query.
   - **Example:**

```
SELECT Name FROM (
    SELECT Name, DepartmentID FROM Employee WHERE Salary > 50000
) AS Temp WHERE DepartmentID = 101;

```

   - **Explanation:** The subquery in the FROM clause is classified as DERIVED.

---

5. **UNION / UNION RESULT**
   - **UNION:** Each SELECT in a UNION query is marked as UNION except the first one, which is PRIMARY.
   - **UNION RESULT:** The result of merging the rows of the UNION query.
   - **Example:**

```
SELECT Name FROM Employee WHERE DepartmentID = 101
UNION
SELECT ProjectName FROM Project WHERE Budget > 150000;

```

---

6. **DEPENDENT SUBQUERY**
   - A subquery that depends on columns from the outer query, making it execute repeatedly.
   - **Example:**

```
SELECT Name FROM Employee WHERE EXISTS (
    SELECT 1 FROM Project WHERE Employee.DepartmentID = Project.DepartmentID
);

```

   - **Issue:** These can be slow due to repeated execution.

---

7. **DEPENDENT JOIN**
   - Similar to a dependent subquery, but for joins where the inner query depends on the outer query's result.
   - **Example:**

```
SELECT * FROM Employee E JOIN Project P ON E.DepartmentID = P.DepartmentID WHERE P.Budget > 150000;

```

---

8. **MATERIALIZED**
   - A subquery that MySQL has optimized by storing the result in a temporary table.
   - **Example:**

```
SELECT Name FROM Employee WHERE DepartmentID IN (
    SELECT DISTINCT DepartmentID FROM Project
);

```

---

2. `type`

The `type` column in the `EXPLAIN` output specifies how MySQL accesses tables to retrieve data. It's crucial for evaluating query efficiency. The access type is listed from most to least efficient.

Types of Access Methods:

1. **ALL (Full Table Scan)**
   - MySQL scans the entire table.
   - **Example:**

```
SELECT * FROM Employee WHERE Salary > 50000;

```

   - **Issue:** Indicates no usable index, leading to high cost.

---

2. **INDEX**
   - MySQL scans all rows but reads from the index instead of the table.
   - **Example:**

```
SELECT Name FROM Employee ORDER BY Salary;

```

   - **Use Case:** This occurs when the query uses an index but does not filter rows effectively.

---

3. **RANGE**
   - Uses an index to retrieve rows within a specified range.
   - **Example:**

```
SELECT * FROM Employee WHERE Salary BETWEEN 40000 AND 60000;

```

   - **Efficient:** Limits the number of rows scanned.

---

4. **REF**
   - Retrieves rows based on matching indexed values.
   - **Example:**

```
SELECT * FROM Employee WHERE DepartmentID = 101;

```

- **Use Case:** For equality comparisons with non-unique indexes.

---

5. **EQ_REF**
   - Fetches at most one row per index key using a unique or primary key.
   - **Example:**

```
SELECT * FROM Employee E JOIN Department D ON E.DepartmentID = D.DepartmentID;

```

   - **Efficient:** MySQL knows it only needs one row per key.

---

6. **CONST**
   - The table has one row matching the query, typically used for primary key lookups.
   - **Example:**

```
SELECT * FROM Employee WHERE EmployeeID = 1;

```

   - **Fastest Type:** MySQL treats the table as a constant.

---

7. **SYSTEM**
   - Similar to CONST, but the table contains only one row.
   - **Example:** A single-row table is used in a query.

---

8. **INDEX_MERGE**
   - Combines multiple indexes to filter rows.
   - **Example:**

```
SELECT * FROM Employee WHERE DepartmentID = 101 OR Salary > 50000;

```

   - **Use Case:** When no single index suffices.

---

9. **NULL**
   - No access to any table is needed, such as for queries that return aggregate functions without a `GROUP BY`.
   - **Example:**

```
SELECT COUNT(*) FROM Employee;

```

Comparing `type` Values

Type	Efficiency	Description
SYSTEM	Most Efficient	Table with a single row.
CONST	Highly Efficient	Single-row match (e.g., primary key).
EQ_REF	Very Efficient	Single row per key match.
REF	Efficient	Indexed equality lookup.
RANGE	Moderate Efficiency	Indexed range lookup.
INDEX	Less Efficient	Index scan without filtering.
ALL	Least Efficient	Full table scan.

---

3. How to Use This Information

1. **Optimize Joins:** Use `EQ_REF` or `REF` access by ensuring foreign keys and indexes are present.
2. **Avoid `ALL` or `INDEX`:** Add indexes or rewrite queries to narrow rows scanned.
3. **Check `select_type`:** Rewrite dependent subqueries as joins or derived tables for better performance.

Would you like detailed optimization examples for specific scenarios?