My SQL

Gnanavel Durairaj

Content

On Day – 2

1. Aggregation Function
2. Grouping Data
3. Sub Queries and Nested Queries
4. Indexes and Optimization
5. Understanding Query Execution Plan
6. Basic Query Optimization
7. Data Manipulation
8. Working with NULL

Aggregation Functions

Aggregation functions perform calculations on a set of values and return a single value. They are often used with the GROUP BY clause to summarize data.

Common Aggregation Functions:

1. **COUNT:**
   - Counts the number of rows that match a condition.
   - Example:

```
SELECT COUNT(*) FROM employees;
```

   - Returns the total number of rows in the employees table.

Aggregation Functions

2. SUM:

- Calculates the sum of a numeric column.

Example:

```
SELECT SUM(salary) FROM employees;
```

- Returns the total salary of all employees.

3. AVG:

- Calculates the average value of a numeric column.

Example:

```
SELECT AVG(salary) FROM employees;
```

- Returns the average salary of all employees.

Aggregation Functions

4. MIN:

- Finds the minimum value in a column.

Example:

```
SELECT MIN(salary) FROM employees;
```

- Returns the lowest salary in the employees table.

5. MAX:

- Finds the maximum value in a column.

Example:

```
SELECT MAX(salary) FROM employees;
```

- Returns the highest salary in the employees table.

Grouping Data Using GROUP BY

The GROUP BY clause groups rows that have the same values in specified columns into summary rows. It is often used with aggregation functions to perform calculations on each group.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example:

```
SELECT department, COUNT(*) AS
employee_count
FROM employees
GROUP BY department;
```

This query groups employees by department and counts the number of employees in each department.

Filtering Grouped Data Using HAVING

The HAVING clause is used to filter groups based on a condition. It is similar to the WHERE clause but is used with aggregated data.

Syntax:

```
SELECT column1, aggregate_function(column2) FROM table_name
GROUP BY column1
HAVING condition;
```

Example:

```
SELECT department, AVG(salary) AS avg_salary FROM employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

This query groups employees by department, calculates the average salary for each department, and returns only those departments where the average salary is greater than 50,000.

Subqueries and Nested Queries

In MySQL, subqueries and nested queries refer to queries that are embedded within other queries. These can be used in `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements to perform more complex operations.

1. Subqueries:

A subquery is a query inside another query. It is typically used to retrieve data that will be used by the outer query.

Types of Subqueries:

- Single-Row Subquery: Returns only one row and one column.
- Multi-Row Subquery: Returns multiple rows but only one column.
- Multi-Column Subquery: Returns multiple columns and multiple rows.
- Correlated Subquery: A subquery that references columns from the outer query.

2. Nested Queries:

A nested query refers to any query inside another query, which can be a subquery in a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

Subqueries and Nested Queries

A subquery (or nested query) is a query inside another query. It can be used in SELECT, INSERT, UPDATE, DELETE, or WHERE clauses.

Types of Subqueries:

1. **Scalar Subquery:**
   - Returns a single value.

   Example:

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

   - This query retrieves employees whose salary is greater than the average salary.

Subqueries and Nested Queries

2. **Row Subquery:**
   - Returns a single row with multiple columns.

   Example :

```
SELECT name, department
FROM employees
WHERE (department, salary) = (SELECT department, MAX(salary) FROM
employees GROUP BY department);
```

   - This query retrieves employees who have the highest salary in their respective departments.

Subqueries and Nested Queries

3. **Column Subquery:**
   - Returns a single column with multiple rows.

   Example :

```
SELECT name
FROM employees
WHERE department IN (SELECT department FROM departments
WHERE location = 'New York');
```

   - This query retrieves employees who work in departments located in New York.

Subqueries and Nested Queries

4. **Table Subquery:**
   - Returns a table (multiple rows and columns).

   Example :

```
SELECT department, AVG(salary) AS avg_salary
FROM (SELECT * FROM employees WHERE hire_date > '2020-01-01')
AS recent_employees
GROUP BY department;
```

   - This query calculates the average salary for employees hired after January 1, 2020, grouped by department.

Subqueries and Nested Queries

Example 1: Subquery in a SELECT Statemnt

You can use a subquery to retrieve a value that is then used in the outer query.

```
SELECT name, salary
FROM employees
WHERE department_id = (SELECT department_id
                       FROM departments
                       WHERE department_name = 'HR');
```

In this example, the subquery `(SELECT department_id FROM departments WHERE department_name = 'HR')` is used to fetch the department_id for the HR department. The outer query then retrieves employees working in that department.

Subqueries and Nested Queries

Example 2: Subquery in a WHERE clause

You can also use a subquery inside a `WHERE` clause to filter results.

```
SELECT name FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

This query retrieves the names of employees whose salary is greater than the average salary of all employees. The subquery calculates the average salary, and the outer query uses it to filter the employees.

Example 3: Correlated Subquery:

A correlated subquery is one where the subquery depends on the outer query for its values. It references columns from the outer query.

```
SELECT name, salary FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees
                WHERE department_id = e.department_id);
```

In this example, the subquery uses the `department_id` from the outer query (aliased as `e`) to calculate the average salary for each department. The outer query then selects employees who earn more than the average salary in their department.

Subqueries and Nested Queries

Example 4: Subquery in an UPDATE statement:

Subqueries can also be used in an `UPDATE` statement to modify data.

```
UPDATE employees SET salary = salary * 1.1
WHERE department_id = (SELECT department_id FROM departments
                       WHERE department_name = 'IT');
```

In this case, the salary of all employees in the "IT" department is increased by 10%. The subquery finds the `department_id` for the IT department.

Example 5: Subquery in an INSERT statement:

You can insert data based on the result of a subquery.

```
INSERT INTO new_employees (name, salary)
SELECT name, salary FROM employees
WHERE department_id = (SELECT department_id FROM departments
                       WHERE department_name = 'Marketing');
```

This query inserts all employees from the Marketing department into the `new_employees` table.

Subqueries and Nested Queries

Example 6: Subquery in a DELETE statement:

A subquery can be used to identify records to delete.

```
DELETE FROM employees WHERE department_id = (SELECT department_id
                    FROM departments WHERE department_name = 'Sales');
```

This will delete all employees working in the "Sales" department.

Types of Subqueries:

- Single-Row Subquery: - Returns only one row.
  - Used with comparison operators like `=`, `<`, `>`, `<=`, etc.

```
SELECT name  FROM employees WHERE department_id = (SELECT department_id
                    FROM departments WHERE department_name = 'IT');
```

- Multi-Row Subquery: - Returns multiple rows.
  - Used with operators like `IN`, `ANY`, or `ALL`.

```
SELECT name   FROM employees   WHERE department_id IN (SELECT department_id
                    FROM departments WHERE department_name IN ('IT', 'HR'));
```

Subqueries and Nested Queries

Types of Subqueries:

- Multi-Column Subquery: - Returns more than one column.
  - Useful in conjunction with `IN` or `EXISTS`.

```
SELECT name
FROM employees
WHERE (department_id, salary) IN (SELECT department_id, salary FROM employees
                                  WHERE salary > 50000);
```

- Correlated Subquery: - A subquery that references columns from the outer query.
  - Executes once for each row processed by the outer query.

```
SELECT name, salary
FROM employees e
WHERE salary > (SELECT AVG(salary)
                FROM employees
                WHERE department_id = e.department_id);
```

Subqueries and Nested Queries

Types of Subqueries:

- Using `EXISTS` with Subqueries:

The `EXISTS` operator checks if the subquery returns any results. It is commonly used in correlated subqueries.

```
SELECT name
FROM employees e
WHERE EXISTS (SELECT 1
              FROM departments d
              WHERE d.department_id = e.department_id
                AND d.department_name = 'HR');
```

This query will return the names of employees who belong to the HR department.

Indexes and Optimization in MySQL

1. What Are Indexes and Why Are They Important?

What Is an Index?

An index is a database object that improves query performance by allowing the database to find data faster, similar to a book's table of contents.

Why Are Indexes Important?

- Speeds Up Queries: Instead of scanning the entire table, MySQL quickly finds rows using an index.
- Improves Sorting and Filtering: Queries with ORDER BY or WHERE conditions run faster.
- Reduces Disk I/O: Helps the database fetch data more efficiently.

Indexes and Optimization in MySQL

Indexes are a crucial aspect of optimizing database queries in MySQL. They help speed up data retrieval by reducing the number of rows MySQL needs to scan. Without indexes, MySQL would need to scan every row in a table for each query, which can be slow, especially for large datasets.

Types of Indexes in MySQL:

1. Primary Index (or Primary Key):
   - Every table in MySQL should ideally have a primary key. A primary key is unique and ensures that no two rows have the same value in the primary key column.
   - The primary key automatically creates a unique index on the table.
2. Unique Index:
   - A unique index ensures that all values in the indexed column are distinct.
   - This is commonly used for columns like email addresses or usernames where duplicates are not allowed.
3. Regular (Non-Unique) Index:
   - A regular index speeds up the search queries but does not require uniqueness. It can be used on columns that are frequently searched or involved in joins.
4. Full-Text Index:
   - A full-text index is used for searching large text fields (like `TEXT` or `VARCHAR`) efficiently. It allows you to perform complex searches like finding words, phrases, or matches within large blocks of text.
5. Spatial Index:
   - Used for spatial data types (like `POINT`, `LINESTRING`, and `POLYGON`). This index type allows efficient queries on geographical data.
6. Composite Index:
   - A composite index is an index on multiple columns. It can help queries that filter or sort by more than one column.
   - MySQL uses the left-most prefix of composite indexes for matching queries.

Indexes and Optimization in MySQL

How Indexes Work:
- When you create an index, MySQL builds a separate data structure (typically a B-tree or hash table) that allows for fast searching of indexed columns.
- The index essentially acts like a lookup table to avoid scanning the entire dataset for the search key.

For example, consider the following query:

```
SELECT * FROM users WHERE email = 'test@example.com';
```

Without an index on the `email` column, MySQL would scan the entire `users` table. But with an index on `email`, MySQL can quickly find the row by searching the index instead.

How Indexes Improve Query Performance:
- Faster Lookups: Searching or filtering data with indexed columns is much faster.
- Faster Joins: When joining tables, indexes can significantly improve performance.
- Faster Sorting: Queries with `ORDER BY` clauses can be optimized if the sorting column is indexed.

Downsides of Indexes:
1. Increased Storage: Indexes consume additional disk space.
2. Slower Writes: Insert, update, and delete operations can be slower because the indexes need to be updated when the data changes.
3. Index Maintenance: As data changes, indexes need to be maintained, which can add overhead.

Indexes and Optimization in MySQL

Indexing Strategies for Optimization:

1. Use Indexes on Columns Used in WHERE, JOIN, and ORDER BY Clauses:
   - If you frequently filter, join, or sort by certain columns, those are good candidates for indexing.
2. Use Composite Indexes:
   - If you have queries that filter by multiple columns, consider creating a composite index for those columns.
   Example: CREATE INDEX idx_name ON table_name (column1, column2);
3. Avoid Over-Indexing:
   - While indexing speeds up SELECT queries, too many indexes can negatively affect write operations (INSERT, UPDATE, DELETE).
   - Focus on the most important queries that need optimization.
4. Use EXPLAIN to Analyze Queries:
   - MySQL's `EXPLAIN` command provides insights into how MySQL executes a query and whether indexes are being used effectively.
   - It shows which indexes MySQL uses for each query and the type of operation (e.g., full table scan, index scan).
   Example:EXPLAIN SELECT * FROM users WHERE email = 'test@example.com';
5. Optimize Index Size:
   - For larger tables, you can use covering indexes—indexes that contain all the columns needed for a query—so that MySQL can fetch the results entirely from the index without needing to access the table data.
   - Example: If a query selects `column1` and `column2`, you can create a composite index on `(column1, column2)` to optimize the query.

Indexes and Optimization in MySQL

When to Use Indexes?

✅ Columns used in WHERE, JOIN, ORDER BY, and GROUP BY.

✅ Primary and foreign keys should always be indexed.

❌ Avoid indexing small tables (it may slow down inserts/updates).

Trade-offs:

- **Storage Overhead:** Indexes consume additional disk space.
- **Write Performance:** Indexes slow down INSERT, UPDATE, and DELETE operations because the index must be updated whenever the data changes.

Creating and Dropping Indexes

Creating an Index

Use the CREATE INDEX statement to create an index on one or more columns.

Syntax:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

Example:

```
CREATE INDEX idx_employee_name
ON employees (name);
```

This creates an index on the name column of the employees table.

Creating and Dropping Indexes

Creating a Unique Index:

Ensures that all values in the indexed column(s) are unique.

Syntax:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

Example:

```
CREATE UNIQUE INDEX idx_employee_email
ON employees (email);
```

Creating and Dropping Indexes

Dropping an Index

Use the DROP INDEX statement to remove an index.

Syntax:

```
DROP INDEX index_name
ON table_name;
```

Example:

```
DROP INDEX idx_employee_name
ON employees;
```

Understanding Query Execution Plans (EXPLAIN)

Why Use EXPLAIN?

The EXPLAIN statement helps analyze how MySQL executes a query and whether indexes are being used efficiently.

Syntax:

```
EXPLAIN SELECT ...;
```

Using EXPLAIN

```
EXPLAIN SELECT * FROM employees WHERE name = 'John';
```

Understanding Query Execution Plans (EXPLAIN)

Example Output

id	select_type	table	type	possible_keys	key	rows	Extra
1	SIMPLE	employees	ref	idx_name	idx_name	1	Using index

Key Details from EXPLAIN:

- **type:** Shows query performance. (Best: const, ref; Worst: ALL)
- **possible_keys:** Lists potential indexes.
- **key:** The actual index used.
- **rows:** Number of rows MySQL will scan.

If type = ALL, MySQL is doing a full table scan (slow). You may need to create an index.

Basics of Query Optimization

1. **Use Indexes Wisely**

   ✅ **Do:**

   ```
   SELECT * FROM employees WHERE name = 'Alice';
   ```

   - Uses an index on name.

   ❌ **Avoid:**

   ```
   SELECT * FROM employees WHERE LOWER(name) = 'alice';
   ```

   - Indexes do not work if a function (LOWER()) is used.

Basics of Query Optimization

2. Optimize ORDER BY Queries

✅ **Do:**

```
SELECT * FROM employees ORDER BY salary ASC;
```

- Uses an index on salary.

❌ **Avoid:**

```
SELECT * FROM employees ORDER BY salary + 1000;
```

- Indexes will not be used if you perform calculations inside ORDER BY.

Basics of Query Optimization

3. Avoid SELECT * (Only Select Necessary Columns)

- ✅ **Do:**

```
SELECT name, salary FROM employees WHERE department_id = 1;
```

- Fetches only required columns.

❌ **Avoid:**

```
SELECT * FROM employees WHERE department_id = 1;
```

- Fetching all columns increases memory usage and slows performance.

Basics of Query Optimization

4. Use Joins Efficiently

✅ **Do:**

```
SELECT e.name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

- Uses an indexed JOIN for better performance.

❌ **Avoid:**

```
SELECT e.name, d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

- Older JOIN syntax can cause performance issues.

Basics of Query Optimization

5. Use LIMIT to Reduce Load

✅ **Do:**

```
SELECT * FROM employees ORDER BY salary DESC LIMIT 10;
```

- Fetches only top 10 employees..

❌ **Avoid:**

```
SELECT * FROM employees ORDER BY salary DESC;
```

- Retrieving all records may slow down performance.

Data Manipulation in MySQL

Data manipulation allows you to filter, modify, and format data effectively. Let's explore CASE statements, handling NULL values, and pattern matching with LIKE and wildcards.

Using CASE Statements

The CASE statement works like an if-else condition inside SQL queries. It helps in conditional output formatting.

Syntax

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE default_result
END
```

Data Manipulation in MySQL

Example: Assign Salary Levels

```
SELECT name, salary,
    CASE
        WHEN salary > 80000 THEN 'High'
        WHEN salary BETWEEN 50000 AND 80000 THEN 'Medium'
        ELSE 'Low'
    END AS salary_level
FROM employees;
```

Categorizes employees into High, Medium, or Low salary groups.

Data Manipulation in MySQL

Example: Status Mapping in a CASE Statement

```
SELECT order_id, status,
    CASE status
        WHEN 'P' THEN 'Pending'
        WHEN 'C' THEN 'Completed'
        WHEN 'X' THEN 'Cancelled'
        ELSE 'Unknown'
    END AS order_status
FROM orders;
```

Converts status codes (P, C, X) into meaningful descriptions.

Working with NULL Values (IS NULL, IS NOT NULL)

In SQL, NULL represents missing or unknown values.

Checking for NULL Values

Example: Find Employees Without Email

```
SELECT name FROM employees WHERE email IS NULL;
```

- Retrieves employees without an email.

Example: Find Employees Who Have an Email

```
SELECT name FROM employees WHERE email IS NOT NULL;
```

- Retrieves employees who have an email.

Handling NULL Using COALESCE()

The COALESCE() function returns the first non-null value.

Example: Display Default Email if NULL

```
SELECT name, COALESCE(email, 'Not Provided') AS email
FROM employees;
```

- If email is NULL, it will display "Not Provided".

Handling NULL Using IFNULL()

IFNULL(value, replacement) replaces NULL with a default value.

Example: Replace NULL Salary with 0

```
SELECT name, IFNULL(salary, 0) AS salary FROM employees;
```

- If salary is NULL, it returns 0.

Handling NULL Using IFNULL()

IFNULL(value, replacement) replaces NULL with a default value.

Example: Replace NULL Salary with 0

```
SELECT name, IFNULL(salary, 0) AS salary FROM employees;
```

- If salary is NULL, it returns 0.

Exercise and Quiz

Thanks – End of Day 02

Edited by Gnanavel Durairaj

10-Mar-2025