# Week-2 NOTES

## ▼ Subqueries

Types of Subqueries

- **Single-Row Subqueries**
  - A subquery that returns exactly one row and one column, used with simple operators like `=`.
- **Multi-Row Subqueries**
  - A subquery that returns multiple rows, used with operators like `IN`, `ANY`, and `ALL`.
- **Correlated Subqueries**
  - An advanced type where the inner query depends on the outer query for its values.
- **Subqueries in Different Clauses**
  - **a) In the `FROM` Clause** (known as Derived Tables).
  - **b) In the `SELECT` Clause** (known as Scalar Subqueries).

### ▼ Single-Row Subqueries

The database always executes the inner query (the subquery) first, and then uses its result to run the outer query.

**Goal**: Find all books that have the same price as 'The God of Small Things'.

#### Step 1: The Inner Query Runs First

The database needs to find the price of 'The God of Small Things'.

```
SELECT
	price
FROM
	books
WHERE
	title = 'The God of Small Things';
```

This query returns a single value: `399.00`.

#### Step 2: The Outer Query Uses the Result

The database now effectively substitutes that result into the main query.

```
SELECT
	title,
	price
FROM
	books
WHERE
	price = 399.00;
```

#### Step 3: The Complete Subquery

Here is how you write it as a single statement. The part in parentheses is the single-row subquery.

```
SELECT
    title,
    price
FROM
    books
WHERE
    price = (
      SELECT
          price
      FROM
          books
       WHERE
          title = 'The God of Small Things'
        );
```

## Common Use Cases

Single-row subqueries are most often used in the `WHERE` and `HAVING` clauses.

Use in HAVING Clause

Goal: Find all authors who have published more books than R.K. Narayan.

```
SELECT
    a.author_name,
    COUNT(b.book_id) AS book_count
FROM
    authors a
    JOIN books b
    ON
    a.author_id = b.author_id
GROUP BY
    a.author_name
HAVING
    book_count > (
  -- This subquery returns a single value: the number of books by R.K. Narayan
  SELECT
      COUNT(book_id)
  FROM
      books
  WHERE
      author_id = (
            SELECT
                author_id
              FROM
                authors
              WHERE
                author_name = 'R.K. Narayan')

);
```

## The Critical Rule and Common Error

A single-row subquery **must** return only one row. If it returns more than one, your query will fail.

**Incorrect Example**: This will cause an error because multiple books have the genre 'Classic', so the subquery returns a list of prices, not a single price.

```
Error Code: 1242. Subquery returns more than 1 row
```

## ▼ Multi-Row Subqueries

A **multi-row subquery** is a nested `SELECT` statement that can return **multiple rows** of data. Because it returns a list of values rather than a single value, you cannot use standard operators like `=`. Instead, you must use special operators designed to work with lists: `IN`, `NOT IN`, `ANY`, and `ALL`.

### The `IN` Operator

This is the most common and intuitive multi-row operator. It checks if a value from the outer query matches **any** value in the list returned by the subquery.

- **Analogy**: "Is this item on the shopping list?" ✅
- **Example**: Find the names of all authors who have written a book in the 'Classic' genre.

```
SELECT
  author_name
FROM
  authors
WHERE
  author_id IN (
    SELECT
      author_id
    FROM
      books
    WHERE
      genre = 'Classic'
  );
```

### The `NOT IN` Operator

This is the opposite of `IN`. It checks if a value from the outer query does **not** match any value in the list returned by the subquery.

- **Analogy**: "Is this item specifically *not* on the shopping list?" ❌
- **Example**: Find the names of all authors who have **not** written any books in the 'Classic' genre.

```
SELECT
  author_name
FROM
  authors
WHERE
  author_id NOT IN (
    SELECT
      author_id
    FROM
      books
    WHERE
      genre = 'Classic'
  );
```

### The `ANY` Operator

The `ANY` operator compares a value to **each** value in the list returned by the subquery. It returns `TRUE` if the comparison is true for **at least one** of the values. It must be used with a standard comparison operator (`=`, `>`, `<`).

- `> ANY` means "greater than the minimum value in the list".
- `< ANY` means "less than the maximum value in the list".
- `= ANY` is identical to `IN`.
- **Example**: Find any book that is more expensive than the **cheapest** 'Classic' book.

```
SELECT
  title,
  price
FROM
  books
WHERE
  price > ANY (
    -- Subquery returns a list of prices for classic books: (199.00, 249.00)
    SELECT
      price
    FROM
      books
    WHERE
      genre = 'Classic'
  );
```

*In this case, `> ANY` means "greater than 199.00", so any book with a price over ₹199.00 will be returned.*

## The `ALL` Operator

The `ALL` operator compares a value to **every** value in the list returned by the subquery. It returns `TRUE` only if the comparison is true for **all** of the values.

- `> ALL` means "greater than the maximum value in the list".
- `< ALL` means "less than the minimum value in the list".
- `<> ALL` is identical to `NOT IN`.
- **Example**: Find the book(s) that are more expensive than **every single** 'Classic' book.

```
SELECT
  title,
  price
FROM
  books
WHERE
  price > ALL (
    -- Subquery returns the same list of prices: (199.00, 249.00)
    SELECT
      price
    FROM
      books
    WHERE
      genre = 'Classic'
  );
```

*In this case, `> ALL` means "greater than 249.00", so only books with a price over ₹249.00 will be returned.*

# ▼ Correlated Subqueries

A **correlated subquery** is a nested query that depends on the outer query for its values. Unlike a simple subquery that runs once, a correlated subquery is executed **repeatedly**, once for each row being processed by the outer query.

This creates a row-by-row processing loop, similar to a nested loop in programming. The inner query cannot be run independently because it needs data from the outer query to work.

## How It Works: A Step-by-Step Example

The best way to understand this is to trace the execution.

**Goal**: Find all books that are more expensive than the average price *for their own genre*.

```
SELECT
    title,
    genre,
    price
FROM
    books AS b1
WHERE
    price > (
        SELECT
            AVG(price)
        FROM
            books AS b2
        WHERE
            b2.genre = b1.genre -- This is the correlation link
    );
```

Here’s the process:

1. The outer query starts processing its first row. Let's say `b1` is `('Five Point Someone', 'Contemporary', 299.00)`.
2. The inner query now executes using the genre from b1 ('Contemporary'):

   SELECT AVG(price) FROM books WHERE genre = 'Contemporary'.

   (Let's assume the result is 299.00).
3. The outer query's `WHERE` clause is now evaluated for this row: `WHERE 299.00 > 299.00`. This is `FALSE`, so the book is discarded.
4. The outer query moves to the next row. Now `b1` is `('The Room on the Roof', 'Classic', 199.00)`.
5. The inner query executes again, this time with the new genre:

   SELECT AVG(price) FROM books WHERE genre = 'Classic'.

   (Let's assume the result is 224.00).
6. The `WHERE` clause is evaluated for this second row: `WHERE 199.00 > 224.00`. This is `FALSE`.

This process repeats for every single row in the `books` table.

## Common Use Case with `EXISTS`

Correlated subqueries are frequently used with the `EXISTS` operator to check for the presence of related data.

**Goal**: Find all authors who have written at least one book.

```
SELECT
    author_name
FROM
    authors AS a
```

```
WHERE
  EXISTS (
    -- For each author, the subquery checks if a book exists with their ID.
    SELECT
      1 -- The '1' is a placeholder; what's selected doesn't matter
    FROM
      books AS b
    WHERE
      b.author_id = a.author_id -- The correlation link
  );
```

This is often more efficient than a `JOIN` because the subquery can stop searching as soon as it finds the first matching book for an author.

## Performance Considerations

- **Potential for Slowness**: Because the inner query runs for each row of the outer query, correlated subqueries can be slow on very large tables.
- **Optimization**: Modern database optimizers are very good at converting many correlated subqueries into more efficient `JOIN`s behind the scenes. However, if you have a slow query, checking if it can be rewritten as a standard `JOIN` is a common optimization technique.

# ▼ Subqueries in Different Clauses

### a) In the `FROM` Clause (Derived Tables)

When a subquery is used in the `FROM` clause, its result set is treated as a temporary, virtual table for the outer query to use. This is called a **derived table**.

- **Key Rule**: A derived table **must** be given an alias.
- **Analogy**: Think of it as building a custom, temporary table on the fly that exists only for the duration of your query.

Example: Find the average number of books written per author.

This is a two-step question. First, we need to count the books for each author. Then, we can find the average of those counts.

1. **Inner Query (The Derived Table)**: This part creates a temporary table named `author_counts` with two columns: `author_id` and `book_count`.
2. **Outer Query**: This part treats `author_counts` like a real table and calculates the average of its `book_count` column.

```
SELECT
  AVG(book_count) AS avg_books_per_author
FROM
  (
    SELECT
      a.author_id,
      COUNT(b.book_id) AS book_count
    FROM
      authors AS a
    LEFT JOIN
      books AS b ON a.author_id = b.author_id
    GROUP BY
      a.author_id
  ) AS author_counts; -- The derived table MUST have an alias
```

### b) In the `SELECT` Clause (Scalar Subqueries)

A subquery used in the `SELECT` list must be a **scalar subquery**, meaning it returns exactly one row and one column (a single value). This value is then displayed as a new column in the final result.

- **Key Rule**: The subquery must return only one value.
- **Analogy**: Think of it as looking up a specific fact and adding it as a new column to every row of your result.

**Example**: Display each author's name and, in another column, show the total count of all books in the library.

```
SELECT
  author_name,
  (
    SELECT COUNT(*) FROM books
  ) AS total_library_books
FROM
  authors;
```

*In this case, the subquery `(SELECT COUNT(*) FROM books)` runs once, and its single result is simply appended to every row.*

**More Advanced (Correlated) Example**: List each author and the year their most recent book was published.

```
SELECT
  author_name,
  (
    SELECT
      MAX(publish_year)
    FROM
      books AS b
    WHERE
      b.author_id = a.author_id -- Correlated to the outer query
  ) AS latest_publication
FROM
  authors AS a;
```

*Here, the subquery is correlated and runs once for each author to find their specific latest publication year.*

# ▼ Set Operators

In MySQL, set operations are used to combine the result sets of two or more `SELECT` statements into a single result set.

The primary set operators supported by MySQL are `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT`.

To illustrate these operations, let's create two simple tables, `employees_usa` and `employees_india`, with some overlapping data.

**Sample Data Setup**

```

CREATE TABLE employees_usa ( id INT PRIMARY KEY, name VARCHAR(50), department VARCHAR(50)
);

CREATE TABLE employees_india ( id INT PRIMARY KEY, name VARCHAR(50), department VARCHAR(50)
);

INSERT INTO employees_usa VALUES
(1, 'John Doe', 'Sales'),
```

```
(2, 'Jane Smith', 'Marketing'),
(3, 'Peter Jones', 'Sales'); -- This employee also exists in the India office

INSERT INTO employees_india VALUES
(3, 'Peter Jones', 'Sales'), -- This is the common employee
(4, 'Priya Sharma', 'HR'),
(5, 'Rohan Singh', 'Marketing');
```

## UNION

The `UNION` operator combines the result sets of two or more `SELECT` statements and **removes duplicate rows**. It merges the lists and ensures every row in the final result is unique.

### Rules for `UNION`

- Each `SELECT` statement within the `UNION` must have the **same number of columns**.
- The columns must also have **compatible data types**.
- The column names in the final result set are taken from the **first** `SELECT` statement.

### Example

Get a single list of all unique employees from both offices.

```
SELECT id, name, department FROM employees_usa
UNION
SELECT id, name, department FROM employees_india;
```

### Result

Notice that "Peter Jones" appears only once, even though he is in both tables. `UNION` has removed the duplicate entry.

| id | name | department |
|---|---|---|
| 1 | John Doe | Sales |
| 2 | Jane Smith | Marketing |
| 3 | Peter Jones | Sales |
| 4 | Priya Sharma | HR |
| 5 | Rohan Singh | Marketing |

## UNION ALL

The `UNION ALL` operator also combines the result sets of two or more `SELECT` statements, but it **does not remove duplicate rows**. It simply appends the results of the second query to the first one.

Because it doesn't perform the extra step of checking for duplicates, `UNION ALL` is generally **faster** than `UNION`.

### Example

Get a combined list of all employees, including duplicates.

```
SELECT id, name, department FROM employees_usa
UNION ALL
SELECT id, name, department FROM employees_india;
```

### Result

Here, "Peter Jones" appears twice because `UNION ALL` includes all rows from both tables regardless of duplication.

| id | name | department |
| --- | --- | --- |
| 1 | John Doe | Sales |
| 2 | Jane Smith | Marketing |
| 3 | Peter Jones | Sales |
| 3 | Peter Jones | Sales |
| 4 | Priya Sharma | HR |
| 5 | Rohan Singh | Marketing |

### `INTERSECT`

**Availability:** This operator is available from **MySQL 8.0.31** onwards.

The `INTERSECT` operator returns only the rows that are **common to both** `SELECT` statements. It gives you the intersection, or overlap, between the two result sets.

#### Example

Find the employees who work in both the USA and India offices.

```
SELECT id, name, department FROM employees_usa
INTERSECT
SELECT id, name, department FROM employees_india;
```

#### Result

The query returns only "Peter Jones" because his record is the only one that exists in both tables.

| id | name | department |
| --- | --- | --- |
| 3 | Peter Jones | Sales |

### `EXCEPT`

**Availability:** This operator is also available from **MySQL 8.0.31** onwards.

The `EXCEPT` operator returns all the unique rows from the **first** `SELECT` statement that are **not present** in the second `SELECT` statement. It essentially subtracts the second result set from the first.

#### Example

Find the employees who work exclusively in the USA office and not in the India office.

```
SELECT id, name, department FROM employees_usa
EXCEPT
SELECT id, name, department FROM employees_india;
```

#### Result

The query returns "John Doe" and "Jane Smith". "Peter Jones" is excluded because his record is present in the `employees_india` table.

| id | name | department |
| --- | --- | --- |
| 1 | John Doe | Sales |
| 2 | Jane Smith | Marketing |

# ▼ Stored Procedure

A **stored procedure** is a pre-compiled set of one or more SQL statements stored in the database. Think of it as a **reusable script or a mini-program** that lives on your database server. You call it by name, pass it some arguments, and it executes its logic

Basic Syntax and Structure

```
-- 1. Change the delimiter
DELIMITER $$

-- 2. Define the procedure
CREATE PROCEDURE procedure_name(parameter1, parameter2, ...)
BEGIN
  -- Your SQL statements go here...
  SELECT 'Hello World!';
END$$

-- 3. Change the delimiter back
DELIMITER ;

-- 4. How to run it
CALL procedure_name();
```

## Parameters: `IN`, `OUT`, and `INOUT`

Parameters are how you pass data into and get data out of a procedure.

### 1. `IN` (Input - The Default)

This parameter passes a value **into** the procedure. The procedure can use it, but it cannot change the original variable that was passed in.

**Example**: Get a user by their ID.

```
DELIMITER $$
CREATE PROCEDURE GetUserByID(IN user_id INT)
BEGIN
  SELECT * FROM users WHERE id = user_id;
END$$
DELIMITER ;

-- Usage
CALL GetUserByID(123);
```

### 2. `OUT` (Output)

This parameter is used to return a value **from** the procedure back to the caller. The procedure sets its value.

**Example**: Count users and return the total.

```
DELIMITER $$
CREATE PROCEDURE CountUsers(OUT total_users INT)
BEGIN
  SELECT COUNT(*) INTO total_users FROM users;
END$$
DELIMITER ;

-- Usage
CALL CountUsers(@user_count); -- The result is stored in the session variable @user_count
SELECT @user_count; -- Now you can see the value
```

The `SELECT ... INTO ...` syntax is key here—it's how you assign a query result to a variable.

### 3. `INOUT` (Input and Output)

This parameter passes a value **in**, allows the procedure to **modify it**, and passes the new value **out**.

**Example**: A counter that you pass in and get back incremented.

```
DELIMITER $$
CREATE PROCEDURE IncrementCounter(INOUT counter INT)
BEGIN
  SET counter = counter + 1;
END$$
DELIMITER ;

-- Usage
SET @my_counter = 5;
CALL IncrementCounter(@my_counter);
SELECT @my_counter; -- The result will be 6
```

### Declaring Variables

Use `DECLARE` to create local variables within the procedure.

```
DELIMITER $$
CREATE PROCEDURE ProcessOrder(IN order_id INT)
BEGIN
  -- Declare local variables
  DECLARE order_total DECIMAL(10, 2);
  DECLARE customer_email VARCHAR(100);

  -- Assign values using SELECT ... INTO
  SELECT total_amount, customer_email INTO order_total, customer_email
  FROM orders WHERE id = order_id;
END$$
DELIMITER ;
```

Conditional Logic with `IF`

```
IF user_status = 'active' THEN
-- Do something
ELSEIF user_status = 'pending' THEN
-- Do something else
ELSE
-- Do a default action
END IF;
```

Looping with `WHILE`

```
DELIMITER $$
CREATE PROCEDURE InsertDummyData()
BEGIN
  DECLARE i INT DEFAULT 1;
  WHILE i <= 10 DO
    INSERT INTO test_table (name) VALUES (CONCAT('Test Name ', i));
    SET i = i + 1;
  END WHILE;
```

```
END$$
DELIMITER ;
```

Error Handling with `DECLARE HANDLER`

By default, if a statement inside a procedure fails, the procedure aborts. You can gracefully catch and handle specific errors (like "no rows found" or "duplicate key").

**Example**: Handle the case where `SELECT...INTO` finds no user.

SQL

```
DELIMITER $$
CREATE PROCEDURE GetUserName(IN user_id INT, OUT user_name VARCHAR(100))
BEGIN
    -- Declare a handler for the 'NOT FOUND' error
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    BEGIN
        SET user_name = 'User Not Found';
    END;

    SELECT name INTO user_name FROM users WHERE id = user_id;
    -- If no user is found, the handler above will trigger and execution will continue
END$$
DELIMITER ;

-- Usage
CALL GetUserName(999, @name); -- 999 is a non-existent ID
SELECT @name; -- The result will be 'User Not Found' instead of an error
```

- `CONTINUE`: The procedure continues after the handler runs.
- `EXIT`: The procedure terminates after the handler runs.

## Dynamic SQL with `PREPARE` and `EXECUTE`

Sometimes you need to build and run SQL queries dynamically (e.g., the table name or column name is a parameter).

**Warning:** This can open you up to SQL injection if not handled carefully.

**Example**: A generic search procedure.

```
DELIMITER $$
CREATE PROCEDURE GenericSearch(
    IN table_name VARCHAR(64),
    IN column_name VARCHAR(64),
    IN search_value VARCHAR(100)
)
BEGIN
    -- Build the query string
    SET @query = CONCAT('SELECT * FROM ', table_name, ' WHERE ', column_name, ' = ?');

    PREPARE stmt FROM @query;
    SET @search_val = search_value;
    EXECUTE stmt USING @search_val; -- The '?' is safely replaced by the variable
    DEALLOCATE PREPARE stmt;
```

```
END$$
DELIMITER ;

-- Usage
CALL GenericSearch('users', 'email', 'test@example.com');
```

Using the `?` placeholder is the **safe way** to pass values into a prepared statement. Avoid concatenating user-provided values directly into the query string.

### Transactions

Procedures are perfect for wrapping a series of operations into a single transaction, ensuring that either all statements succeed or none do (atomicity).

**Example**: A bank transfer.

```
DELIMITER $$
CREATE PROCEDURE TransferFunds(
    IN from_account INT,
    IN to_account INT,
    IN amount DECIMAL(10, 2)
)
BEGIN
    -- On any error, roll back the transaction
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
    END;

    START TRANSACTION;
        UPDATE accounts SET balance = balance - amount WHERE id = from_account;
        UPDATE accounts SET balance = balance + amount WHERE id = to_account;
    COMMIT;
END$$
DELIMITER ;
```

- **View a Procedure's Code**:
  `SHOW CREATE PROCEDURE procedure_name;`
- **Delete a Procedure**:
  `DROP PROCEDURE [IF EXISTS] procedure_name;`

## What is a Cursor?

A **cursor** is a temporary, named area in memory that holds the result of a `SELECT` query. Its main purpose is to let you **process the result set one row at a time**. It acts like a pointer or a bookmark that moves from one row to the next.

This row-by-row processing happens inside a `LOOP` within a stored procedure.

### When Should We Use a Cursor?

You should use a cursor **only when you absolutely have to**. Standard SQL queries that operate on the entire set of data at once (set-based operations) are far more efficient.

However, cursors are necessary for certain complex tasks:

- **Sequential Processing:** When you need to perform a series of complex actions for each individual row that can't be done in a single SQL statement.
- **Complex Calculations:** When the calculation for one row depends on the result from a previous row.
- **Calling Other Procedures:** When you need to run another stored procedure for each row in your result set.

## The Cursor Lifecycle (The Process)

Using a cursor involves a clear, four-step process. The `LOOP` is part of this lifecycle.

### 1. `DECLARE` the Cursor

First, you define the cursor and associate it with a `SELECT` statement. This step names your cursor and tells it what data to work with.

```
DECLARE sales_cursor CURSOR FOR
  SELECT totalamount FROM orders WHERE status = 'Completed';
```

### 2. `OPEN` the Cursor

Next, you open the cursor. This executes the `SELECT` query, and the result set is stored in the cursor's memory area, ready to be read.

```
OPEN sales_cursor;
```

### 3. `FETCH` Data Inside a `LOOP`

This is where the row-by-row action happens. You create a `LOOP` that repeatedly calls the `FETCH` command. `FETCH` retrieves the **next available row** from the cursor and stores its data into local variables you've declared.

```
-- Variable to hold the data from each row
DECLARE current_amount DECIMAL(10, 2);
read_loop: LOOP -- Get the next row FETCH sales_cursor INTO current_amount;
END LOOP;
```

### 4. `CLOSE` the Cursor

After the loop finishes, you must `CLOSE` the cursor. This releases the memory and resources that the cursor was using.

```
CLOSE sales_cursor;
```

### `HANDLER`

How does the loop know when to stop? If you `FETCH` past the last row, it causes an error. To prevent this, you use a **`HANDLER`**. A `DECLARE HANDLER FOR NOT FOUND` is a rule that tells the procedure what to do when the cursor runs out of rows, which is usually to set a flag that ends the loop.

**Example with a handler:**

```
DELIMITER $$
CREATE PROCEDURE CalculateTotalSales()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE current_amount DECIMAL(10, 2);
  DECLARE total_sales DECIMAL(12, 2) DEFAULT 0.00;
```

```
    -- 1. DECLARE Cursor
    DECLARE sales_cursor CURSOR FOR
        SELECT totalamount FROM orders WHERE status = 'Completed';

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- 2. OPEN Cursor
    OPEN sales_cursor;

    read_loop: LOOP
        -- 3. FETCH inside the LOOP
        FETCH sales_cursor INTO current_amount;

        -- Exit condition for the loop
        IF done THEN
            LEAVE read_loop;
        END IF;

        SET total_sales = total_sales + current_amount;
    END LOOP;

    -- 4. CLOSE Cursor
    CLOSE sales_cursor;

    SELECT total_sales;
END$$
DELIMITER ;
```

### Advanced Error Handling with `SIGNAL`

Besides using a generic handler for errors, you can create and throw your own custom errors using the `SIGNAL` statement. This is extremely useful for enforcing business rules.

#### Example: Prevent Ordering a Non-Existent Item

Let's modify the `AddItemToOrderWithTransaction` procedure to throw a specific error if the `menuitemid` doesn't exist.

```
DELIMITER $$
CREATE PROCEDURE AddItemSafely(IN p_orderid INT, IN p_menuitemid INT)
BEGIN
    DECLARE item_count INT;

    SELECT COUNT(*) INTO item_count FROM menuitems WHERE menuitemid = p_menuitemid;

    IF item_count = 0 THEN
        -- Throws a custom error and stops execution
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Menu item does not exist.';
    ELSE
        SELECT 'Item added' AS message;
    END IF;
END$$
DELIMITER ;

-- This call will fail with the custom error message:
CALL AddItemSafely(3, 999);
```

## ▼ Stored Functions

A **Stored Function** is a reusable set of SQL statements stored in the database. It performs operations and returns a **single value**. It is different from a Stored Procedure because:

- A **function must return a value** using the `RETURN` statement.
- A function is usually called within SQL queries.

## Syntax of Stored Function

```
DELIMITER $$

CREATE FUNCTION function_name(parameter1 datatype, parameter2 datatype, ...)
RETURNS return_datatype
DETERMINISTIC
BEGIN
    -- Function logic
    RETURN some_value;
END $$

DELIMITER ;
```

- `DELIMITER $$` : Changes the statement delimiter to allow using `;` inside the function body.
- `DETERMINISTIC` : Indicates the function always returns the same output for the same input.
- `RETURN` : The mandatory statement to return a value.

When you define a **stored function**, you can specify a **characteristic** like:

- `DETERMINISTIC`
- `NOT DETERMINISTIC`
- `CONTAINS SQL`
- `NO SQL`
- `READS SQL DATA`
- `MODIFIES SQL DATA //not allowed` — ERROR 1415 (0A000): Not allowed to modify SQL data in stored function or trigger

## Example 1 – Simple Function to Calculate Tax

### Requirement:

We want to calculate a tax (say 5%) on any given amount.

```
DELIMITER $$

CREATE FUNCTION calculate_tax(amount DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    RETURN amount * 0.05;
END $$

DELIMITER ;
```

### How to Use:

```
SELECT calculate_tax(100);  -- Returns 5.00
```

## Example 2 – Function to Get Customer Full Name

### Requirement:

Return a customer's full name by passing `customerid`.

```
DELIMITER $$

CREATE FUNCTION get_customer_fullname(cust_id INT)
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
    DECLARE full_name VARCHAR(100);

    SELECT CONCAT(firstname, ' ', lastname) INTO full_name
    FROM customers
    WHERE customerid = cust_id;

    RETURN full_name;
END $$

DELIMITER ;
```

### How to Use:

```
SELECT get_customer_fullname(1);  -- Returns 'Jignesh Patel'
```

## Example 3 – Calculate Total Amount for an Order

### Requirement:

We want a function that calculates total amount of an order (sum of quantity × item price).

```
DELIMITER $$

CREATE FUNCTION calculate_order_total(o_id INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE total DECIMAL(10,2);

    SELECT SUM(quantity * itempriceatorder)
    INTO total
    FROM orderdetails
    WHERE orderid = o_id;

    IF total IS NULL THEN
        SET total = 0;
    END IF;

    RETURN total;
END $$

DELIMITER ;
```

### How to Use:

```
SELECT calculate_order_total(1);  -- Returns total amount for order 1
```

### Advanced Concept: Handling NULLs, Conditions, and Loops

You can use control flow statements inside functions:

- `IF`, `CASE`
- Loops: `WHILE`, `REPEAT`, `LOOP`

#### Example – Check if a Menu Item Is Available

```
DELIMITER $$

CREATE FUNCTION is_item_available(menu_id INT)
RETURNS BOOLEAN
DETERMINISTIC
BEGIN
    DECLARE availability BOOLEAN;

    SELECT isavailable INTO availability
    FROM menuitems
    WHERE menuitemid = menu_id;

    IF availability IS NULL THEN
        RETURN FALSE;
    END IF;

    RETURN availability;
END $$

DELIMITER ;
```

Usage:

```
SELECT is_item_available(1);  -- Returns 1 (TRUE) or 0 (FALSE)
```

## Using Stored Functions in Queries

You can use your stored functions directly in `SELECT`, `WHERE`, `ORDER BY`, etc.

#### Example – List Orders with Calculated Tax

```
SELECT orderid, totalamount, calculate_tax(totalamount) AS tax_amount
FROM orders;
```

This will list each order with a calculated tax.

## DETERMINISTIC vs NON-DETERMINISTIC

- `DETERMINISTIC`: The function always returns the same result for same inputs.
- `NOT DETERMINISTIC`: The function may return different results (e.g., using `NOW()`, `RAND()`).

**Example** (Non-deterministic function):

```
SET GLOBAL log_bin_trust_function_creators = 1; -- This tells MySQL to trust function creators and allow creation of non-deterministic functions without explicit characteristics.
-- This is less safe for replication environments because non-deterministic behavior may cause inconsistencies
```

```
across master and slave servers.
-- Non- Deterministic Function
```

```
DELIMITER $$

CREATE FUNCTION get_current_time()
RETURNS DATETIME
NOT DETERMINISTIC

BEGIN
  RETURN NOW();
END $$

DELIMITER ;
```

### Limitations of Stored Functions

- Cannot use `COMMIT`, `ROLLBACK`, or transactions.
- Cannot modify tables directly (use Stored Procedures for that).
- Must return a value.
- Should not have side effects (good practice).

### Best Practices

- Use functions only when they return a computed value.
- Keep them small and focused.
- Use `DETERMINISTIC` if possible for performance benefits.
- Avoid making functions depend on global state or variables outside their scope.

## ▼ Triggers

A **Trigger** is a **special kind of stored program** that is automatically executed (or fired) when a specific event occurs on a table.

- It is triggered by events such as `INSERT`, `UPDATE`, or `DELETE`.
- It helps enforce business rules, do automatic logging, validation, or audit.

### Trigger Syntax

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON table_name
FOR EACH ROW
BEGIN
  -- SQL statements
END;
```

- `BEFORE` or `AFTER`: Defines when the trigger fires.
- `INSERT`, `UPDATE`, or `DELETE`: The event that triggers the action.
- `FOR EACH ROW`: The trigger executes for each row affected.

### Key Concepts

| Term | Description |
| --- | --- |
| `NEW` | Holds the new row data (for `INSERT` and `UPDATE`). |

| Term | Description |
|---|---|
| `OLD` | Holds the old row data (for `UPDATE` and `DELETE`). |
| `BEFORE` Trigger | Runs **before** the event happens (good for validation or modifying input). |
| `AFTER` Trigger | Runs **after** the event happens (good for logging, updating related tables). |

## Example 1 – Simple BEFORE INSERT Trigger

### Goal:

Automatically set `totalamount = 0` when a new order is inserted, if not provided.

```
DELIMITER $$

CREATE TRIGGER before_order_insert
BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
  IF NEW.totalamount IS NULL THEN
    SET NEW.totalamount = 0;
  END IF;
END$$

DELIMITER ;
```

```
INSERT INTO orders (customerid, tablenumber, status) VALUES (1, 2, 'Pending');
```

- `totalamount` will automatically be set to `0`.

## Example 2 – AFTER INSERT Trigger for Audit Logging

Let's say we want to log every time a new customer is added.

### Step 1: Create Audit Table

```
CREATE TABLE customer_audit (
  auditid INT AUTO_INCREMENT PRIMARY KEY,
  customerid INT,
  action VARCHAR(20),
  actiontime DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

### Step 2: Trigger

```
DELIMITER $$

CREATE TRIGGER after_customer_insert
AFTER INSERT ON customers
FOR EACH ROW
BEGIN
  INSERT INTO customer_audit (customerid, action)
  VALUES (NEW.customerid, 'INSERT');
END$$

DELIMITER ;
```

### Example 3 – BEFORE UPDATE Trigger for Data Validation

Ensure that the price of a menu item is never negative.

```
DELIMITER $$

CREATE TRIGGER before_menuitem_update
BEFORE UPDATE ON menuitems
FOR EACH ROW
BEGIN
  IF NEW.price < 0 THEN
    SET NEW.price = OLD.price;
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Price must be a positive number.';
  END IF;
END$$

DELIMITER ;
```

### Example 4 – AFTER DELETE Trigger to Log Deletes

Log when an order is deleted.

```
CREATE TABLE order_deletes_audit (
  auditid INT AUTO_INCREMENT PRIMARY KEY,
  orderid INT,
  deleted_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Trigger:

```
DELIMITER $$

CREATE TRIGGER after_order_delete
AFTER DELETE ON orders
FOR EACH ROW
BEGIN
  INSERT INTO order_deletes_audit (orderid)
  VALUES (OLD.orderid);
END$$

DELIMITER ;
```

### Important Rules & Limitations

| Rule | Explanation |
|---|---|
| Only one trigger per event per table (in MySQL) | You cannot create multiple triggers of the same type (e.g., two `BEFORE INSERT` on same table). |
| Triggers run automatically | No need to call them explicitly. |
| No transaction control | You cannot use `COMMIT` or `ROLLBACK` inside triggers. |
| Cannot return data | Triggers do not return values. |

### Viewing Triggers

```
SHOW TRIGGERS;
```

Or

```
SELECT * FROM information_schema.triggers WHERE trigger_schema = 'hoteldb';
```

## Dropping a Trigger

```
DROP TRIGGER IF EXISTS before_order_insert;
```

## Maintaining Order Total Automatically

Instead of manually updating `orders.totalamount`, we can automatically update it when order details change.

```
DELIMITER $$

CREATE TRIGGER after_orderdetails_insert
AFTER INSERT ON orderdetails
FOR EACH ROW
BEGIN
    UPDATE orders
    SET totalamount = totalamount + (NEW.quantity * NEW.itempriceatorder)
    WHERE orderid = NEW.orderid;
END$$

DELIMITER ;
```

Similarly, create `AFTER UPDATE` and `AFTER DELETE` triggers to keep the `orders.totalamount` in sync.

# ▼ Views

A **View** is a **virtual table** based on the result of a SELECT query.

- It doesn’t store data by itself.
- Acts like a table when queried, but the data comes from underlying tables.
- Useful for abstraction, simplifying complex queries, and enforcing security.

### Simple Analogy:

- **Table** → Actual data storage (physical table).
- **View** → A window that shows data in a pre-defined way.

## Basic Syntax of View Creation

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

Create a simple view that lists customer full names and phone numbers.

```
CREATE VIEW CustomerInfo AS
SELECT customerid, CONCAT(firstname, ' ', lastname) AS fullname, phonenumber
FROM customers;
```

## Using a View

Once created, you can query the view just like a normal table:

```
SELECT * FROM CustomerInfo;
```

It returns:

| customerid | fullname | phonenumber |
|---|---|---|
| 1 | Jignesh Patel | 9876543210 |
| 2 | Meera Shah | 9876543211 |
| 3 | Hitesh Desai | 9876543212 |

## Example – View with JOIN

### Requirement:

Create a view that shows each order along with customer name.

```
CREATE VIEW OrderSummary AS
SELECT o.orderid,
       CONCAT(c.firstname, ' ', c.lastname) AS customer_name,
       o.tablenumber,
       o.totalamount,
       o.status
FROM orders o
JOIN customers c ON o.customerid = c.customerid;
```

```
SELECT * FROM OrderSummary;
```

## Updating Data Through Views

By default, a view is **read-only**, unless it is **updatable**.

### Updatable View Conditions (simplified):

- Must reference a single table.
- Cannot use aggregate functions (`SUM()`, `COUNT()`, etc.).
- No GROUP BY, DISTINCT, UNION, or subqueries in SELECT.

### Example of Updatable View (Allowed):

```
CREATE VIEW SimpleCustomer AS
SELECT customerid, firstname, lastname, phonenumber
FROM customers;
```

Now you can do:

```
UPDATE SimpleCustomer SET firstname = 'Nisharg' WHERE customerid = 1;
```

It updates the **underlying `customers` table**.

### Non-Updatable View Example:

```
CREATE VIEW OrderTotals AS
SELECT customerid, SUM(totalamount) AS total_spent
```

```
FROM orders
GROUP BY customerid;
```

```
UPDATE OrderTotals SET total_spent = 1000 WHERE customerid = 1;
```

Will result in error:

```
ERROR 1417 (HY000): This version of MySQL doesn't yet support updateable views with aggregates
```

### WITH CHECK OPTION

The `WITH CHECK OPTION` is a powerful way to enforce **data integrity rules when updating or inserting into views**.

#### Purpose:

It prevents a row from being updated (or inserted) such that it would no longer be visible through the view.

#### Example

#### Step 1 – Create Filtered View

```
CREATE VIEW AvailableMenuItems AS
SELECT menuitemid, itemname, price
FROM menuitems
WHERE isavailable = TRUE
WITH CHECK OPTION;
```

#### Step 2 – Attempted Update

```
UPDATE AvailableMenuItems SET isavailable = FALSE WHERE menuitemid = 1;
```

Error:

```
ERROR 1369 (HY000): CHECK OPTION failed 'AvailableMenuItems'
```

Why?

- Because after the update, the row no longer satisfies `isavailable = TRUE`, and the row would disappear from the view.
- `WITH CHECK OPTION` prevents this to maintain consistency.

#### Dropping a View

```
DROP VIEW IF EXISTS CustomerInfo;
```

▼ Views vs CTE

## Views vs CTE (Common Table Expressions)

### What Is a View?

- A **View** is a **virtual table** based on the result of a stored `SELECT` query.
- Defined using `CREATE VIEW`.
- Acts like a table when queried.

- Data is not stored separately – always reflects the underlying tables.
- Example:

```
CREATE VIEW CustomerInfo AS
SELECT customerid, CONCAT(firstname, ' ', lastname) AS fullname, phonenumber
FROM customers;
```

## What Is a CTE (Common Table Expression)?

- A **CTE** is a **temporary result set** used in a single query.
- Defined using `WITH cte_name AS (SELECT ...)`.
- Exists only during the execution of that query.
- Example:

```
WITH HighValueOrders AS (
    SELECT orderid, customerid, totalamount
    FROM orders
    WHERE totalamount > 300
)
SELECT * FROM HighValueOrders;
```

## Key Differences

| Feature | View | CTE |
|---|---|---|
| Lifetime | Permanent (until explicitly dropped) | Temporary (for one query only) |
| Reusability | Yes – reusable in multiple queries | No – used in the query where defined |
| Recursion | ❌ Not supported | ✅ Supported with `WITH RECURSIVE` |
| Parameters | ❌ Not supported | ❌ Not directly supported, use WHERE clause |
| Performance | May be cached by DB | Re-executed every time |
| Security | Can restrict user access | Depends on query access control |
| Ideal Use Case | Data abstraction, security layer | Organizing complex queries and recursion |

## 4. Example Comparison

### View Example:

```
CREATE VIEW OrderSummary AS
SELECT orderid, customerid, totalamount
FROM orders
WHERE status = 'Completed';
```

Query:

```
SELECT * FROM OrderSummary WHERE totalamount > 500;
```

### CTE Example:

```
WITH CustomerOrderCounts AS (
    SELECT customerid, COUNT(orderid) AS orders_count
    FROM orders
    GROUP BY customerid
```

```
)
SELECT customerid FROM CustomerOrderCounts WHERE orders_count > 1;
```

### 5. Advanced Feature of CTE – Recursion Example

```
WITH RECURSIVE Numbers AS (
  SELECT 1 AS num
  UNION ALL
  SELECT num + 1 FROM Numbers WHERE num < 5
)
SELECT * FROM Numbers;
```

# Backup & Restore

## Types of Backups

| Type | Description | Pros | Cons |
| --- | --- | --- | --- |
| Logical Backup | Data is exported as SQL scripts (e.g., `mysqldump`). | Easy to restore, portable. | Slower for large datasets. |
| Physical Backup | Copies raw data files (e.g., `.ibd`, `.frm`). | Faster, precise. | Less flexible, hard to migrate. |
| Incremental Backup | Only backs up data changed since last backup. | Saves space and time. | Restore process is complex. |
| Full Backup | Entire database is backed up. | Simple restore. | Requires large storage. |

## Logical Backup

A logical backup is a file that contains a series of SQL statements which can be executed to rebuild a database's structure and data from scratch.

Think of it as a detailed **blueprint or a recipe** for your database. It doesn't copy the physical database files on the disk; instead, it generates the instructions (`CREATE TABLE`, `INSERT INTO`, etc.) needed to recreate everything. The most common tool for creating logical backups in MySQL is the `mysqldump` command-line utility.

## How it Works: The `mysqldump` Process

Let's perform a basic backup of your `hoteldb` and see what happens.

### Creating the Backup

You run this command in your system's terminal (not the MySQL client):

```
mysqldump -u root -p hoteldb > hoteldb_backup.sql
```

This command connects to the database and "dumps" its structure and data into a text file named `hoteldb_backup.sql`.

### Restoring the Backup

To restore, you feed this script back into the `mysql` client:

```
mysql -u root -p new_database < hoteldb_backup.sql
```

This executes every command in the file, effectively recreating your database in `new_database`.

## Advanced `mysqldump` Options

The real power of `mysqldump` comes from its many options that allow you to customize your backup.

### Consistency on Live Databases

- -single-transaction

  This is arguably the most important option for any live database using the InnoDB storage engine. It starts a transaction before the backup begins, ensuring that you get a consistent "snapshot" of the data at a single moment in time. It achieves this without locking the tables, meaning your application can continue to read and write data while the backup is running.

  ```
  mysqldump -u root -p --single-transaction hoteldb > consistent_backup.sql
  ```

### Including All Database Objects

- -routines and --triggers

  By default, mysqldump does not back up your Stored Procedures, Functions, and Triggers. You must explicitly include them. For a complete backup, always use these flags.

  ```
  mysqldump -u root -p --single-transaction --routines --triggers hoteldb > complete_backup.sql
  ```

### Partial Backups (Structure or Data Only)

- -no-data (or -d)

  This dumps only the database schema (the CREATE TABLE statements) without any of the data. This is perfect for creating an empty copy of your database for a development or testing environment.

  ```
  mysqldump -u root -p --no-data hoteldb > schema_only.sql
  ```

- -no-create-info (or -t)

  This dumps only the data (the INSERT statements) without the table structure. This is useful if you want to refresh the data in a database where the tables already exist.

  ```
  mysqldump -u root -p --no-create-info hoteldb > data_only.sql
  ```

### Filtering Data

- -where='condition' (or -w 'condition')

  This extremely powerful option allows you to back up only the rows that match a specific condition.

  **Example**: Back up only the completed orders and their details.

  ```
  mysqldump -u root -p hoteldb orders --where="status='Completed'" > completed_orders.sql
  ```

Similar To this command line we can perform same using workbench

## Step 1: Open the Data Export Tool

1. Connect to your database in MySQL Workbench.
2. In the left-hand **Navigator** panel, under the **Management** section, click on **Data Export**.1

## Step 2: Select the Database to Back Up

A new "Data Export" tab will open.

1. In the main panel, you'll see a list of all the databases (schemas) on your server.
2. Check the box next to the database you want to back up. In your case, select hoteldb.

## Step 3: Configure Your Backup Options

This is where you control what gets backed up, similar to using the advanced options in `mysqldump`.

1. **Objects to Export**: For each schema you select, you can choose to dump its **Structure and Data**, **Data Only**, or **Structure Only**. The default "Dump Structure and Data" is what you usually want for a full backup.
2. **Export Options**:
   - Select **Export to Self-Contained File**. This will create a single `.sql` file, just like the `mysqldump` command.2
   - Specify the file path where you want to save your backup (e.g., `C:\backups\hotel.sql`).
   - Check the box **Include Create Schema** if you want the `CREATE DATABASE ...;` statement to be included in your backup file. This is very useful.

### Including Stored Procedures, Functions, and Triggers

To include your stored code, which is essential for a complete backup:

1. Click on the **Advanced Options...** button on the top right.
2. In the new window that appears, ensure that the boxes for **Routines** (for procedures and functions) and **Triggers** are checked.
3. Click **OK**.

## Step 4: Start the Export

1. After configuring everything, click the **Start Export** button in the bottom right corner.
2. A progress window will appear, and you'll see the export process running.
3. Once it's finished, you'll see a message "Export completed." Your `.sql` backup file is now saved in the location you specified.

## How to Restore Using Workbench

The process to restore is just as simple, using the **Data Import/Restore** tool.

1. In the Navigator, click on **Data Import/Restore**.
2. Select **Import from Self-Contained File** and choose the `.sql` backup file you created.
3. Under **Default Schema to be Imported**, you can select the target database you want to restore into.
4. Click **Start Import** to begin the restore process.

## Physical Backup

**MySQL Enterprise Backup (MEB)** is Oracle's commercial, high-performance tool for creating **hot, online, physical backups** of MySQL databases. It's part of the MySQL Enterprise Edition subscription and is designed for mission-critical systems where performance, reliability, and low downtime are essential.

Think of it as the official, industrial-strength version of a physical backup utility, directly supported by Oracle.

- Instead of exporting SQL statements, you copy the **MySQL data directory** (like `/var/lib/mysql`).

### Steps:

1. Stop MySQL Service:

```
sudo systemctl stop mysql
```

2. Copy Data Directory:

```
cp -r /var/lib/mysql /backup/mysql_data/
```

3. Restart MySQL:

```
sudo systemctl start mysql
```

### Restoration

1. Stop MySQL.
2. Copy backed-up data files back to `/var/lib/mysql`.
3. Ensure permissions are correct:

```
sudo chown -R mysql:mysql /var/lib/mysql
```

4. Start MySQL.

## Export / Import Using `SELECT INTO OUTFILE`

### Export Data to File (e.g., CSV)

```
SELECT * FROM customers
INTO OUTFILE '/tmp/customers.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

### Import Data Back

```
LOAD DATA INFILE '/tmp/customers.csv'
INTO TABLE customers
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

# ▼ Query Optimization

## What is an Index in Database?

An **Index** is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and overhead when inserting or updating data.

It works similarly to the index of a book. Instead of searching the entire book page by page to find a particular topic, you can go directly to the index and find the page number quickly.

## Why Are Indexes Used?

- **Speed Up Queries**: Indexes make SELECT queries faster by reducing the number of rows to scan.
- **Efficient Data Retrieval**: Especially useful for large tables where full table scans are slow.
- **Sorting and Searching**: Helps in ORDER BY and WHERE conditions.
- **Avoid Repeated Full Table Scans**: Without indexes, MySQL does a full table scan for every query.

## How Does an Index Work Internally?

- Data in a table is stored unordered.
- An index is often implemented as a **B-Tree** or **Hash Table**.

### B-Tree (Balanced Tree)

- Organizes data hierarchically.
- Allows logarithmic time complexity (O(log n)) for search operations.

Example:

Suppose a table `students(id, name)` has 1 million rows.

Without index → Every query scans 1 million rows.

With index on `id` → The query finds the record in ≈ $\log_2(1{,}000{,}000)$ ≈ 20 steps.

## Types of Indexes

1. **Primary Key Index**

   Automatically created when you define a PRIMARY KEY.

2. **Unique Index**

   Ensures all indexed values are unique (e.g., UNIQUE(email)).

3. **Normal (Non-Unique) Index**

   Speeds up searches but allows duplicate values.

4. **Full-Text Index**

   Used for searching text within long text columns.

5. **Composite Index**

   Index on multiple columns (e.g., INDEX (col1, col2)).

   - Useful when WHERE clause uses multiple columns together.

## When to Use Indexes

### Use Index When:

- The column is frequently used in `WHERE` clauses.
- The column is used in `JOIN` conditions.
- The column is part of `ORDER BY` or `GROUP BY`.
- The table is large and performance is an issue.

### Avoid Indexing:

- On small tables (because full table scan is fast).
- On columns with very low cardinality (few unique values like boolean flags).
- On columns that are frequently updated, as indexes need updating too, adding overhead.

## How to Create and Use Index in SQL

### Create an Index:

```
CREATE INDEX idx_customerid ON orders(customerid);
```

### Composite Index:

```
CREATE INDEX idx_customer_table ON orders(customerid, tablenumber);
```

### Drop an Index:

```
DROP INDEX idx_customerid ON orders;
```

### Example Query Using Index:

```
SELECT * FROM orders WHERE customerid = 1;
```

## Checking Index Usage

Use EXPLAIN to check whether MySQL uses an index:

```
EXPLAIN SELECT * FROM orders WHERE customerid = 1;
```

Look for the `key` column in the result — if it shows the index name, the index is used.

## Covering Index

A **Covering Index** holds all columns a query needs.

Example Query:

```
SELECT customerid, totalamount FROM orders WHERE customerid = 1;
```

Create covering index:

```
CREATE INDEX idx_covering ON orders(customerid, totalamount);
```

## What Is EXPLAIN in MySQL?

The `EXPLAIN` **command** helps you understand how MySQL executes a query — i.e., the execution plan.

- It shows how tables are scanned.
- Which indexes are used.
- The join order.
- How many rows are estimated to be scanned.

This helps us **optimize slow queries**.

## Basic Syntax

```
EXPLAIN SELECT * FROM orders WHERE customerid = 1;
```

Or more advanced:

```
EXPLAIN FORMAT=JSON SELECT * FROM orders WHERE customerid = 1;
```

## Output Columns of EXPLAIN

| Column | Meaning |
|---|---|
| `id` | Query execution order identifier. |
| `select_type` | Type of SELECT: `SIMPLE`, `PRIMARY`, `SUBQUERY`, etc. |
| `table` | The table being accessed. |
| `type` | Access type (important): `const`, `eq_ref`, `ref`, `range`, `index`, `ALL`. |
| `possible_keys` | Indexes MySQL could use. |
| `key` | Index MySQL actually used. |
| `key_len` | Length of the index used (in bytes). |
| `ref` | Column used in key lookup. |
| `rows` | Estimated rows to be scanned. |
| `Extra` | Additional info (e.g., `Using where`, `Using filesort`, `Using temporary`). |

### Type Column – Understanding Access Types

| Type | Description | Example |
|---|---|---|
| const | Single row read by PRIMARY KEY or UNIQUE index. | `WHERE customerid = 1` |
| eq_ref | Best for JOIN on unique columns. | `JOIN customers ON orders.customerid = customers.customerid` |
| ref | Non-unique index lookup. | `WHERE status = 'Completed'` |
| range | Range scan (e.g., WHERE ordertime BETWEEN ...). | `WHERE ordertime BETWEEN '2023-01-01' AND '2023-12-31'` |
| index | Full index scan (faster than table scan). | Ordering by indexed column. |
| ALL | Full table scan → Bad unless small table. | `SELECT * FROM orders WHERE orderid > 0;` |

| Value | Meaning |
|---|---|
| Using where | MySQL applies WHERE condition after index lookup. |
| Using index | Index contains all columns required → Avoids table row access. |
| Using filesort | MySQL performs an extra sort (could be optimized by indexes). |
| Using temporary | A temporary table is used (e.g., for GROUP BY). |
| Using join buffer | For non-indexed joins → Bad practice for large datasets. |

## EXAMPLES

### Example 1 – Simple Query

```
EXPLAIN SELECT * FROM orders WHERE customerid = 1;
```

Suppose we have index:

```
CREATE INDEX idx_customerid ON orders(customerid);
```

Output:

| id | select_type | table | type | possible_keys | key | key_len | r |
|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | orders | ref | idx_customerid | idx_customerid | 4 | c |

Interpretation:

- `type = ref`: MySQL uses index lookup (good).
- `key = idx_customerid`: The correct index is used.
- `rows = 10`: Estimated number of matching rows.

### Example 2 – Join Query

```
EXPLAIN SELECT o.orderid, c.firstname
FROM orders o
JOIN customers c ON o.customerid = c.customerid
WHERE c.phonenumber = '9876543210';
```

Assuming `customers.phonenumber` is UNIQUE and indexed.

Expected EXPLAIN Output:

| id | select_type | table | type | possible_keys | key | key_len | r |
|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | customers | const | phonenumber | phonenumber | 102 | c |
| 1 | SIMPLE | orders | ref | idx_customerid | idx_customerid | 4 | c |

Interpretation:

- `customers` is accessed by UNIQUE index → Very fast (`type = const`).
- Then `orders` is filtered by `customerid` index.

### Example 3 – ORDER BY Without Index (Bad)

```
EXPLAIN SELECT * FROM orders ORDER BY ordertime;
```

Output:

| id | select_type | table | type | possible_keys | key | key_len | ref |
|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | orders | ALL | NULL | NULL | NULL | NULI |

Warning:

- `type = ALL`: Full Table Scan.
- `Using filesort`: Inefficient sorting.

Fix → Create index on `ordertime`:

```
CREATE INDEX idx_ordertime ON orders(ordertime);
```

After that, running EXPLAIN again should show:

| id | select_type | table | type | possible_keys | key | key_len | ref |
|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | orders | index | idx_ordertime | idx_ordertime | 8 | NUL |

### EXPLAIN ANALYZE (MySQL 8.0+)

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE customerid = 1;
```

🛠️ Difference from EXPLAIN:

- Provides **actual execution time per step**.
- Shows real rows scanned vs estimated.

Example Output:

| id | select_type | table | type | key | rows | filtered | Ext |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | SIMPLE | orders | ref | idx_customerid | 10 | 100.00 | Usi |