**DDL, DML, DQL, and TCL in MySQL**

In MySQL, SQL (Structured Query Language) is divided into several categories based on the operations they perform on a database. These categories are:

1. **DDL (Data Definition Language)**
2. **DML (Data Manipulation Language)**
3. **DQL (Data Query Language)**
4. **TCL (Transaction Control Language)**

Each category handles specific aspects of database management. Let’s go over each one in detail:

| **Category** | **Description** | **Example Commands** |
| --- | --- | --- |
| **DDL** | Defines and modifies the structure of the database. | CREATE, ALTER, DROP, TRUNCATE |
| **DML** | Manipulates data in the database (insert, update, delete). | INSERT, UPDATE, DELETE |
| **DQL** | Queries the database for retrieving data. | SELECT, JOIN, GROUP BY, ORDER BY |
| **TCL** | Controls transactions (commit, rollback, savepoints). | COMMIT, ROLLBACK, SAVEPOINT |

**1. DDL (Data Definition Language)**

**DDL (Data Definition Language)** refers to a set of SQL commands used to define, manage, and modify the structure of a database, tables, and other database objects. These commands do not deal with the manipulation of data but instead focus on defining and altering the database schema, ensuring the right data structure to store and organize the information.

In MySQL, DDL commands are used to perform tasks like creating databases, tables, altering their structure, or removing them altogether. Once a DDL operation is executed, it is immediately applied, and the changes are permanent, meaning they cannot be rolled back unless specific backup mechanisms are in place.

Here’s a breakdown of the main **DDL commands** in MySQL:

**1. CREATE**

The CREATE command is used to create new database objects, such as databases, tables, views, and indexes.

**CREATE DATABASE:**

The CREATE DATABASE command is used to create a new database.

Example:

CREATE DATABASE my\_database;

**CREATE TABLE:**

The CREATE TABLE command is used to create a new table within a database. You define the structure of the table by specifying the columns, their datatypes, and constraints like primary keys.

Example:

CREATE TABLE employees (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

age INT,

hire\_date DATE

);

**CREATE INDEX:**

An index is created to improve the performance of data retrieval operations. It can be applied to one or more columns of a table.

Example:

CREATE INDEX idx\_age ON employees (age);

**CONSTRAINTS:**

**Constraints** are rules or restrictions applied to the columns in a database table to ensure the integrity, validity, and accuracy of the data. They help enforce data quality by preventing invalid data from being inserted into the database. Constraints can be applied to columns when creating or altering a table, and they govern how the database accepts or rejects data.

We will create a table named orders where:

* **PRIMARY KEY** on order\_id ensures each order is uniquely identified.
* **UNIQUE** on email ensures no two orders can have the same email.
* **NOT NULL** on customer\_name ensures that every order has a name associated with it.
* **DEFAULT** on order\_amount provides a default value of 100 if no value is given.
* **CHECK** ensures the order\_date cannot be in the future.
* **FOREIGN KEY** ensures that each order is linked to a valid customer from the customers table.
* **INDEX** improves the performance of queries that filter or sort by the order\_date.

**SQL Example:**

CREATE TABLE customers (

id INT AUTO\_INCREMENT PRIMARY KEY, -- Primary key for customers table

name VARCHAR(100) NOT NULL -- Name cannot be NULL

);

CREATE TABLE orders (

order\_id INT AUTO\_INCREMENT PRIMARY KEY, -- Primary key for orders table

email VARCHAR(255) UNIQUE NOT NULL, -- Email must be unique and not NULL

customer\_name VARCHAR(100) NOT NULL, -- Customer name cannot be NULL

order\_amount DECIMAL(10, 2) DEFAULT 100.00, -- Default value for order\_amount is 100

order\_date DATE CHECK (order\_date <= CURDATE()), -- Ensures order\_date is not in the future

customer\_id INT, -- Foreign key to customers table

FOREIGN KEY (customer\_id) REFERENCES customers(id), -- Foreign key constraint

INDEX idx\_order\_date (order\_date) -- Index to optimize queries on order\_date

);

**2. ALTER**

The ALTER command is used to modify the structure of an existing database object, like adding or removing columns, changing datatypes, or renaming objects.

**ALTER TABLE:**

* **Add a Column**: Adds a new column to an existing table.

ALTER TABLE employees ADD COLUMN salary DECIMAL(10, 2);

* **Modify a Column**: Changes the datatype or properties of an existing column.

ALTER TABLE employees MODIFY COLUMN age SMALLINT;

* **Rename a Column**: Renames an existing column.

ALTER TABLE employees RENAME COLUMN hire\_date TO joining\_date;

* **Drop a Column**: Removes an existing column from the table.

ALTER TABLE employees DROP COLUMN salary;

**3. DROP**

The DROP command is used to delete a database object, such as a database, table, or index. This command permanently removes the object and all its data (for tables) or structure.

**DROP DATABASE:**

Deletes an entire database and all of its contents (tables, views, etc.).

Example:

DROP DATABASE my\_database;

**DROP TABLE:**

Deletes a table and all its rows and structure from the database.

Example:

DROP TABLE employees;

**DROP INDEX:**

Deletes an index from a table.

Example:

DROP INDEX idx\_age ON employees;

**4. TRUNCATE**

The TRUNCATE command removes all rows from a table, but unlike DROP, it keeps the table structure intact, allowing you to insert data into it later. It is faster than DELETE because it doesn't generate individual row deletion logs.

Example:

TRUNCATE TABLE employees;

Note: TRUNCATE is a DDL operation and cannot be rolled back (in most database systems), and it doesn’t activate triggers (unlike DELETE).

**5. RENAME**

The RENAME command is used to change the name of a database object, such as a table. It is often used when you need to reorganize the structure of a database.

Example:

RENAME TABLE employees TO staff\_members;

**2. DML (Data Manipulation Language)**

**DML (Data Manipulation Language)** refers to the subset of SQL commands used for managing and manipulating data stored in database tables. Unlike DDL (Data Definition Language), which deals with the structure of the database, DML is focused on the data itself — allowing you to insert, update, delete, and retrieve records.

The most common DML operations include **INSERT**, **UPDATE** and **DELETE**. Let’s explore these in detail.

**1. INSERT**

The INSERT command is used to add new records (rows) into a table. You specify the table into which data should be inserted and the values for each column.

**Syntax**:

INSERT INTO table\_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);

**Examples:**

* Inserting a single row:

INSERT INTO employees (id, name, age, hire\_date)

VALUES (1, 'John Doe', 30, '2025-02-16');

* Inserting multiple rows:

INSERT INTO employees (id, name, age, hire\_date)

VALUES

(2, 'Jane Smith', 28, '2024-06-10'),

(3, 'Michael Brown', 35, '2023-08-20');

* Inserting data from another table (subquery):

INSERT INTO archived\_employees (id, name, age)

SELECT id, name, age FROM employees WHERE age > 60;

**2. UPDATE**

The UPDATE command is used to modify existing records in a table. You can specify one or more columns whose values should be updated, and you can filter the records to update using the WHERE clause.

**Syntax**:

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

**Examples:**

* Updating a single column:

UPDATE employees

SET age = 31

WHERE id = 1;

* Updating multiple columns:

UPDATE employees

SET age = 32, hire\_date = '2025-03-01'

WHERE id = 1;

* Using conditions:

UPDATE employees

SET salary = salary \* 1.1

WHERE age > 30;

**Important**: Always use the WHERE clause in an UPDATE statement unless you intend to update all records. Without WHERE, **all rows in the table** will be updated, which can lead to unintended data changes.

**3. DELETE**

The DELETE command is used to remove records from a table. You can use the WHERE clause to specify which rows should be deleted. If no condition is provided, **all rows** in the table will be deleted, but the structure of the table remains intact.

**Syntax:**

DELETE FROM table\_name

WHERE condition;

**Examples:**

* Deleting a specific record:

DELETE FROM employees

WHERE id = 1;

* Deleting all records (without dropping the table):

DELETE FROM employees;

* Deleting using a condition:

DELETE FROM employees

WHERE age < 30;

**Important**: As with UPDATE, always be cautious when using DELETE without a WHERE clause. It will remove **all records** in the table.

**3. DQL (Data Query Language)**

**DQL (Data Query Language)** is a subset of SQL that is used to query the database and retrieve data from one or more tables. Unlike DDL (Data Definition Language) or DML (Data Manipulation Language), DQL commands are designed to retrieve information rather than modify it. The primary DQL command is **SELECT**, but it also includes related clauses and operations that help you filter, sort, and organize the retrieved data.

The primary focus of DQL is to allow you to **retrieve**, **organize**, and **filter** data as needed. Here's a detailed breakdown of DQL concepts and commands:

**1. SELECT**

The SELECT statement is the most commonly used DQL command, used to retrieve data from one or more tables. It can be used to retrieve all or specific columns, with various filters and ordering options.

**Syntax**:

SELECT column1, column2, ...

FROM table\_name

WHERE condition

ORDER BY column

LIMIT number;

**Example:**

* **Basic SELECT**: Retrieve all columns from a table.

SELECT \* FROM employees;

* **Select Specific Columns**: Retrieve specific columns (e.g., name and age from the employees table).

SELECT name, age FROM employees;

* **Using WHERE Clause**: Retrieve records with conditions (e.g., employees older than 30).

SELECT \* FROM employees WHERE age > 30;

* **Using ORDER BY**: Sort the result set by one or more columns.

SELECT \* FROM employees ORDER BY age DESC;

* **Using LIMIT**: Limit the number of rows returned (e.g., retrieve only 5 records).

SELECT \* FROM employees LIMIT 5;

**2. WHERE Clause**

The WHERE clause is used to filter records based on specified conditions. It can be used with comparison operators, logical operators, and pattern matching.

**Common Comparison Operators:**

* =: Equal to
* <> or !=: Not equal to
* >: Greater than
* <: Less than
* >=: Greater than or equal to
* <=: Less than or equal to
* BETWEEN: Range between two values
* IN: Matches a list of values
* LIKE: Pattern matching (with wildcards)
* IS NULL: Checks for NULL values

**Example:**

* **Filtering records**: Retrieve employees with age greater than 30.

SELECT \* FROM employees WHERE age > 30;

* **Using BETWEEN**: Retrieve employees whose age is between 25 and 35.

SELECT \* FROM employees WHERE age BETWEEN 25 AND 35;

* **Using IN**: Retrieve employees who belong to specific departments.

SELECT \* FROM employees WHERE department IN ('HR', 'IT');

* **Using LIKE**: Search for employees whose names start with 'J'.

SELECT \* FROM employees WHERE name LIKE 'J%';

* **Checking NULL values**: Retrieve employees whose hire\_date is not set.

SELECT \* FROM employees WHERE hire\_date IS NULL;

**Subqueries in SQL**

A **subquery** (also known as a **nested query**) is a query that is embedded within another query. It is used to perform operations or retrieve data that is then used by the outer query. Subqueries can be used in SELECT, INSERT, UPDATE, or DELETE statements.

Subqueries can be categorized into different types depending on how they are used:

1. **Single-Row Subquery**: Returns a single value (e.g., a single column, single row).
2. **Multi-Row Subquery**: Returns multiple rows and values.
3. **Multi-Column Subquery**: Returns multiple columns.
4. **Correlated Subquery**: Depends on values from the outer query and is executed once for each row processed by the outer query.
5. **Non-Correlated Subquery**: Does not depend on the outer query and can be executed independently.

**1. Single-Row Subquery**

A **single-row subquery** returns a single value (a scalar value). This is commonly used with comparison operators such as =, >, <, >=, <=.

**Example:** Find the employees with a salary greater than the average salary.

SELECT name, salary

FROM employees

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

* The subquery (SELECT AVG(salary) FROM employees) returns a single scalar value (the average salary), and the outer query retrieves all employees whose salary is greater than that average.

**2. Multi-Row Subquery**

A **multi-row subquery** returns multiple rows. This type of subquery is often used with operators like IN, ANY, or ALL.

**Example:** Find employees who work in departments located in a particular city.

SELECT name

FROM employees

WHERE department\_id IN (SELECT department\_id FROM departments WHERE city = 'New York');

* The subquery (SELECT department\_id FROM departments WHERE city = 'New York') returns multiple rows of department IDs in New York, and the outer query retrieves all employees whose department\_id matches one of those IDs.

**3. Multi-Column Subquery**

A **multi-column subquery** returns more than one column in the result set. It is often used with IN or EXISTS and can be used for comparison between the columns of the subquery and the outer query.

**Example:** Find employees whose salary is greater than the salary of employees in the same department but with less experience.

SELECT name, salary, experience

FROM employees

WHERE (salary, department\_id) IN

(SELECT salary, department\_id

FROM employees

WHERE experience < 5);

* The subquery returns two columns: salary and department\_id. The outer query retrieves employees whose salary and department\_id match any pair returned by the subquery.

**4. Correlated Subquery**

A **correlated subquery** is a subquery that references columns from the outer query. This means the subquery is evaluated once for each row processed by the outer query.

**Example:** Find employees whose salary is higher than the average salary in their department.

SELECT e.name, e.salary, e.department\_id

FROM employees e

WHERE e.salary > (SELECT AVG(salary)

FROM employees

WHERE department\_id = e.department\_id);

* The subquery references the department\_id from the outer query (e.department\_id). For each employee, the subquery is executed to calculate the average salary within their department.

**5. Non-Correlated Subquery**

A **non-correlated subquery** does not reference any columns from the outer query. It can be executed independently of the outer query.

**Example:** Find employees who have the same salary as the highest-paid employee in the company.

SELECT name, salary

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees);

* The subquery (SELECT MAX(salary) FROM employees) returns the maximum salary from all employees, and the outer query retrieves employees with that salary.

**Subquery in SELECT Statement**

Subqueries can also be used in the SELECT clause to return values that are computed for each row.

**Example:** Retrieve the employee name, salary, and the department name for each employee.

SELECT name,

salary,

(SELECT department\_name

FROM departments

WHERE departments.department\_id = employees.department\_id) AS department\_name

FROM employees;

* The subquery in the SELECT clause retrieves the department\_name for each employee based on their department\_id.

**Subquery in UPDATE Statement**

A **subquery** can be used in an UPDATE statement to update data based on the result of another query.

**Example:** Update the salary of employees in a particular department by increasing it based on the average salary in the department.

UPDATE employees

SET salary = salary + (SELECT AVG(salary) FROM employees WHERE department\_id = 1)

WHERE department\_id = 1;

* The subquery calculates the average salary for the department with department\_id = 1, and the outer query updates all employees in that department by increasing their salary by the calculated average.

**Subquery in DELETE Statement**

Subqueries can be used in a DELETE statement to delete rows based on conditions from another table.

**Example:** Delete employees who are in departments located in a specific city.

DELETE FROM employees

WHERE department\_id IN (SELECT department\_id FROM departments WHERE city = 'Los Angeles');

* The subquery returns department IDs for departments located in Los Angeles, and the outer query deletes all employees whose department\_id matches one of those IDs.

**3. JOIN Operations**

The JOIN clause is used to combine rows from two or more tables based on a related column between them. It’s one of the most powerful features of SQL and is essential for working with multiple tables.

**Types of JOIN:**

* **INNER JOIN**: Returns only the rows where there is a match in both tables.

SELECT employees.name, departments.department\_name

FROM employees

INNER JOIN departments

ON employees.department\_id = departments.id;

* **LEFT JOIN (or LEFT OUTER JOIN)**: Returns all rows from the left table and matching rows from the right table. If no match is found, NULL values are returned for the right table.

SELECT employees.name, departments.department\_name

FROM employees

LEFT JOIN departments

ON employees.department\_id = departments.id;

* **RIGHT JOIN (or RIGHT OUTER JOIN)**: Returns all rows from the right table and matching rows from the left table. If no match is found, NULL values are returned for the left table.

SELECT employees.name, departments.department\_name

FROM employees

RIGHT JOIN departments

ON employees.department\_id = departments.id;

* **FULL JOIN (or FULL OUTER JOIN)**: Returns all rows when there is a match in either left or right table. This type of join is not directly supported in MySQL, but you can simulate it using a combination of LEFT JOIN and RIGHT JOIN.

SELECT employees.name, departments.department\_name

FROM employees

LEFT JOIN departments ON employees.department\_id = departments.id

UNION

SELECT employees.name, departments.department\_name

FROM employees

RIGHT JOIN departments ON employees.department\_id = departments.id;

**4. GROUP BY**

The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows, like finding the total number of employees per department.

**Syntax:**

SELECT column1, COUNT(\*)

FROM table\_name

GROUP BY column1;

**Example:**

* **Grouping and Aggregating Data**: Get the count of employees in each department.

SELECT department, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department;

* **Using Aggregate Functions**: You can use aggregate functions like COUNT(), AVG(), SUM(), MAX(), and MIN() along with GROUP BY.

SELECT department, AVG(salary) AS average\_salary

FROM employees

GROUP BY department;

**5. HAVING**

The HAVING clause is used to filter groups created by the GROUP BY clause. It works like WHERE but is used for aggregated data.

**Example:**

* **Filtering Grouped Data**: Retrieve departments with more than 10 employees.

SELECT department, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department

HAVING COUNT(\*) > 10;

**6. ORDER BY**

The ORDER BY clause is used to sort the result set in ascending or descending order. By default, it sorts in ascending order (ASC), but you can explicitly specify the sorting order.

**Example:**

* **Sorting in Ascending Order**: Retrieve all employees sorted by their age in ascending order.

SELECT \* FROM employees

ORDER BY age ASC;

* **Sorting in Descending Order**: Retrieve all employees sorted by their salary in descending order.

SELECT \* FROM employees

ORDER BY salary DESC;

**4. TCL (Transaction Control Language)**

**TCL (Transaction Control Language)** is a subset of SQL commands used to manage transactions in a database. Transactions are used to ensure data integrity, consistency, and correctness when multiple operations are being performed. TCL commands control the behavior of transactions by allowing you to commit, roll back, or save points within a transaction.

The primary TCL commands include **COMMIT**, **ROLLBACK** and **SAVEPOINT**. Let's go over these commands in detail.

**1. COMMIT**

The COMMIT command is used to save all the changes made during the current transaction to the database. Once a transaction is committed, the changes are permanent and cannot be undone.

**Syntax:**

COMMIT;

**Example:**

* **Committing a transaction**: After making several changes (like INSERT, UPDATE, or DELETE), the COMMIT command will make these changes permanent.

START TRANSACTION;

UPDATE employees SET salary = salary + 1000 WHERE department = 'HR';

INSERT INTO log (action) VALUES ('Salary updated for HR department');

COMMIT;

After the COMMIT, the changes made by the UPDATE and INSERT statements are saved permanently.

**2. ROLLBACK**

The ROLLBACK command is used to undo changes made during the current transaction. If a transaction is rolled back, all the changes made by INSERT, UPDATE, or DELETE commands since the last COMMIT are reversed.

**Syntax:**

ROLLBACK;

**Example:**

* **Rolling back a transaction**: If an error occurs, or you decide to not apply changes, you can roll back the transaction.

START TRANSACTION;

UPDATE employees SET salary = salary + 500 WHERE department = 'Sales';

DELETE FROM employees WHERE id = 100;

ROLLBACK;

In this case, neither the UPDATE nor the DELETE will be applied, and the database will return to the state it was in before the START TRANSACTION command.

**3. SAVEPOINT**

A SAVEPOINT allows you to set a marker within a transaction. If needed, you can roll back to a specific savepoint without affecting the entire transaction. This allows for partial rollbacks.

**Syntax:**

SAVEPOINT savepoint\_name;

**Example:**

* **Using SAVEPOINT**: You can set multiple savepoints and roll back to any of them.

START TRANSACTION;

SAVEPOINT sp1;

UPDATE employees SET salary = salary + 500 WHERE department = 'IT';

SAVEPOINT sp2;

UPDATE employees SET salary = salary + 500 WHERE department = 'HR';

-- If something goes wrong, you can roll back to sp1 and keep the changes made before that.

ROLLBACK TO sp1;

COMMIT;

In this example, the changes made after sp1 (for the HR department) will be rolled back, but the changes for the IT department will remain.