MySQL

1. Database [Syntax]

   i. Create Database syntax:

```
CREATE DATABASE database_name;
CREATE DATABASE IF NOT EXISTS database_name;
```

   Example:

```
CREATE DATABASE my_database;
```

   ii. Drop Database syntax:

```
DROP DATABASE database_name;
DROP DATABASE IF EXISTS database_name;
```

   Example:

```
DROP DATABASE my_database;
```

   iii. Use/Select Database syntax:

```
USE database_name;
```

   Example:

```
USE my_database;
```

2. Table Syntax

   i. Create Table

To create a new table in MySQL, use the following syntax:

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    ...
);
```

Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    dept_id INT,
    salary DECIMAL(10, 2)
);
```

ii. Alter Table
   To modify an existing table,
   use the `ALTER TABLE` command.

   Add a Column:

```
        ALTER TABLE table_name ADD column_name datatype constraint;

        #Example:

            ALTER TABLE employees ADD bonus DECIMAL(10, 2);
```

Modify a Column:

```
ALTER TABLE table_name MODIFY column_name new_datatype new_constraint;
```

#Example:

```
ALTER TABLE employees MODIFY name VARCHAR(100);
```

Drop a Column:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

#Example:

```
ALTER TABLE employees DROP COLUMN bonus;
```

iii. Drop Table

To delete a table, use the following syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE employees;
```

iv.

Data types, including their category, example usage, and explanation:

Category	Data Type	Example	Explanation
Numeric	`INT`	`INT(11)`	Stores whole numbers. `11` specifies display width, though it doesn’t affect storage.
	`DECIMAL`	`DECIMAL(10, 2)`	Fixed-point number with 10 digits in total, 2 of which are after the decimal.
	`FLOAT`	`FLOAT(7, 3)`	Floating-point number with 7 digits total, 3 after the decimal.
	`DOUBLE`	`DOUBLE(16, 4)`	Double-precision floating-point number, larger range and precision than `FLOAT`.
	`TINYINT`	`TINYINT(4)`	Small integer, useful for storing very small numbers, like boolean flags.
	`BIGINT`	`BIGINT(20)`	Large integer, useful for storing very large whole numbers.
String	`VARCHAR`	`VARCHAR(255)`	Variable-length string up to 255 characters. Ideal for general text fields.
	`CHAR`	`CHAR(10)`	Fixed-length string, padded with spaces if necessary. Good for fixed-format data.
	`TEXT`	`TEXT`	Large text string up to 65,535 characters. Used for storing large bodies of text.
	`ENUM`	`ENUM('small', 'medium', 'large')`	String with a predefined set of values. Used for storing categorical data.
Date and Time	`DATE`	`DATE`	Stores a date value in `YYYY-MM-DD` format. Suitable for date-only data.
	`DATETIME`	`DATETIME`	Stores date and time in `YYYY-MM-DD HH:MM:SS` format. Used when both are needed.
	`TIMESTAMP`	`TIMESTAMP`	Similar to `DATETIME`, but also tracks the last update time automatically.
	`TIME`	`TIME`	Stores time in `HH:MM:SS` format. Useful for time-only data.
	`YEAR`	`YEAR(4)`	Stores a year as a 4-digit value. Good for storing years, like birth years.
Binary	`BINARY`	`BINARY(16)`	Fixed-length binary string. Used for storing binary data of a specific length.
	`VARBINARY`	`VARBINARY(255)`	Variable-length binary string. Ideal for binary data where size can vary.
	`BLOB`	`BLOB`	Binary large object, for large binary data like images or multimedia files.
Spatial	`POINT`	`POINT`	Stores a geographic point in 2D space. Used in GIS applications.

	`LINESTRING`	`LINESTRING`	Stores a line made up of points in 2D space. Useful for geographic paths.
	`POLYGON`	`POLYGON`	Stores a polygon defined by multiple points. Used for complex geographic shapes.

v.

Constraints are
rules applied to columns
in a table
to enforce data integrity.

Common constraints, including examples and explanations:

Constraint	Example	Explanation
PRIMARY KEY	`PRIMARY KEY (id)`	Uniquely identifies each row in the table. A table can have only one primary key.
FOREIGN KEY	`FOREIGN KEY (dept_id) REFERENCES departments(id)`	Ensures that the value in a column matches a value in another table's column, establishing a relationship between tables.
UNIQUE	`UNIQUE (email)`	Ensures all values in the column are unique across the table. Duplicate values are not allowed.
NOT NULL	`name VARCHAR(50) NOT NULL`	Ensures that a column cannot have a `NULL` value. It must always have a value.
CHECK	`CHECK (age >= 18)`	Ensures that all values in a column satisfy a specific condition.
DEFAULT	`salary DECIMAL(10, 2) DEFAULT 50000.00`	Sets a default value for a column if no value is specified during insertion.
AUTO_INCREMENT	`id INT PRIMARY KEY AUTO_INCREMENT`	Automatically generates a unique number for each new row. Typically used with primary keys.
INDEX	`INDEX (name)`	Creates an index on a column to improve query performance. Not a strict constraint, but a way to optimize data retrieval.

Constraints Usages:

a. PRIMARY KEY :

- Uniquely identifies each record in the table.
- A combination of `NOT NULL` and `UNIQUE`.
- Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

b.   FOREIGN KEY :

- Enforces a link between the data in two tables.
- Ensures referential integrity.
- Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(id)
);
```

c.   UNIQUE :

- Prevents duplicate values in a column.
- Example:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

d. NOT NULL :

- Ensures that a column must always have a value.
- Example:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL
);
```

e. CHECK :

- Validates the values before inserting or updating them.
- Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    age INT CHECK (age >= 18)
);
```

f. DEFAULT :

- Sets a default value for a column if no value is specified.
- Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    salary DECIMAL(10, 2) DEFAULT 50000.00
);
```

g. AUTO_INCREMENT :

- Automatically increments the value for each new row.
- Typically used for primary key columns.
- Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50)
);
```

h. INDEX :

- Improves the speed of data retrieval operations on a column.
- Not enforced as a strict constraint but used for optimization.
- Example:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    INDEX (name)
);
```

3. DML [Syntax, Examples]

i. INSERT Syntax

Insert a Single Row

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

- Example:

```
INSERT INTO Department (name)
VALUES ('Marketing');
```

Insert Multiple Rows

```
INSERT INTO table_name (column1, column2, column3)
VALUES
   (value1a, value2a, value3a),
   (value1b, value2b, value3b),
   (value1c, value2c, value3c);
```

- Example:

```
INSERT INTO Department (name)
VALUES
   ('Logistics'),
   ('Operations'),
   ('Public Relations');
```

Insert with Default Values

```
INSERT INTO table_name (column1, column2)
VALUES (value1, DEFAULT);
```

- Example:

```
INSERT INTO Department (name)
VALUES ('Business Development');
```

ii. UPDATE Syntax

Update Specific Rows

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

- Example:

```
UPDATE Department
SET name = 'Human Resources'
WHERE name = 'HR';
```

Update All Rows

```
UPDATE table_name
SET column1 = value1, column2 = value2;
```

- Example:

```
UPDATE Department
SET name = 'General Administration';
```

iii. SELECT Syntax Variations

Select All Columns

```
SELECT * FROM table_name;
```

- Example:

```
SELECT * FROM Department;
```

Select Specific Columns

```
SELECT column1, column2
FROM table_name;
```

- Example:

```
SELECT id, name
FROM Department;
```

Select with Conditions

```
SELECT column1, column2
FROM table_name
WHERE condition;
```

- Example:

```
SELECT id, name
FROM Department
WHERE name LIKE 'IT%';
```

Select with Sorting

```
SELECT column1, column2
FROM table_name
ORDER BY column1 [ASC|DESC];
```

- Example:

```
SELECT id, name
FROM Department
ORDER BY name ASC;
```

Select with Aggregation

```
SELECT aggregate_function(column)
FROM table_name;
```

- Example:

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

iv. DELETE Syntax Variations

Delete Specific Rows

```
DELETE FROM table_name
WHERE condition;
```

- Example:

```
DELETE FROM Department
WHERE name = 'Legal';
```

Delete All Rows (but keep the table structure)

```
DELETE FROM table_name;
```

- Example:

```
DELETE FROM Department;
```

Delete All Rows (with truncation)

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE Department;
```

4. TCL (Transaction Control Language)

i. COMMIT

- This command is used to save all the changes made in the current transaction.
- Once you execute `COMMIT`,

the changes become permanent and cannot be undone.

```
COMMIT;
```

- Example:

```
INSERT INTO Employee (name, dept_id, job_title, salary)
VALUES ('Alice', 2, 'Manager', 80000.00);
COMMIT;
```

ii. ROLLBACK

- This command is used to undo changes made in the current transaction since the last `COMMIT`.
- It restores the database to the last committed state.

```
ROLLBACK;
```

- Example:

```
DELETE FROM Employee WHERE id = 10;
ROLLBACK;
```

iii. SAVEPOINT

- This command is used to set a point within a transaction to which you can later roll back.
- It helps in partially rolling back a transaction.

```
SAVEPOINT savepoint_name;
```

- Example:

```
    UPDATE Employee SET salary = 90000.00 WHERE id = 5;
    SAVEPOINT before_bonus;
    UPDATE Employee SET bonus = 5000.00 WHERE id = 5;
    ROLLBACK TO before_bonus;
```

iv. SET TRANSACTION

- This command is used to specify the characteristics of the current transaction,
such as the isolation level or whether it is read-only.

```
SET TRANSACTION [READ WRITE | READ ONLY];
```

- Example:

```
    SET TRANSACTION READ ONLY;
    SELECT * FROM Employee;
    COMMIT;
```

v. Example of TCL Usage:

```
BEGIN;

-- Insert a new employee
INSERT INTO Employee (name, dept_id, job_title, salary)
VALUES ('Chris', 3, 'Analyst', 50000.00);

-- Set a savepoint
SAVEPOINT before_raise;

-- Update salary
```

```sql
UPDATE Employee SET salary = 55000.00 WHERE name = 'Chris';

-- Rollback to savepoint (undo salary change)
ROLLBACK TO before_raise;

-- Commit the transaction (saving the insert)
COMMIT;
```

vi ACID Properties:

ACID properties ensure
reliability,
correctness, and
robustness
in database transactions.

A. Atomicity

- Ensures that a transaction is treated
  as a single, indivisible unit.
  Either all the operations within the transaction
  are executed successfully,
  or none of them are.
- Example:

If a bank transfer operation involves
debiting one account and
crediting another,
both operations must succeed or fail together.

- Key Point: No partial transactions should occur.

```
BEGIN;
UPDATE Account
SET balance = balance - 100
WHERE account_id = 1;

UPDATE Account
SET balance = balance + 100
WHERE account_id = 2;
COMMIT;
```

B. Consistency

- Ensures that a transaction brings the database from one valid state to another, maintaining database rules such as integrity constraints (e.g., primary key, foreign key).
- A valid state means the database follows all its rules, such as data types, unique values, foreign key relationships, etc.
- The transaction cannot break any of these rules—if it does, the transaction will fail, and the database will remain unchanged (in its original valid state).
- Example:
  A transaction cannot violate referential integrity (e.g., a foreign key constraint).

- Key Point: The database must always be in a consistent state before and after the transaction.

```
INSERT INTO Orders (order_id, customer_id, product_id)
VALUES (1, 100, 500);
```

```
-- This will only succeed if customer_id 100 and product_id 500 exist.
```

C. Isolation

- Ensures that concurrently executed transactions
are isolated from each other,
meaning the intermediate state of a transaction
is invisible to other transactions until it is complete.
- Example:
Two users updating the same account balance concurrently
will not see each other's changes
until the transactions are committed.

- Key Point: Transactions should appear
as if they are executed one after the other,
even if they are executed concurrently.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
-- Transaction 1 reads the balance.
SELECT balance FROM Account WHERE account_id = 1;
```

D. Durability

- Ensures that once a transaction is committed,
the changes are permanent,
even if the system crashes immediately afterward.
- Example: After transferring money and committing the transaction,
the updated balances remain stored,
even in the event of a power failure.

- Key Point: Data is never lost once a transaction is successfully committed.

```
COMMIT;
-- Data is permanently written and safe.
```

E. Summary Table:

ACID Property	Description	Example
Atomicity	Ensures all or none of the transaction's operations are performed.	In a fund transfer, both debit and credit operations must succeed or be rolled back if one fails.
Consistency	Guarantees that a transaction brings the database from one valid state to another.	Ensures that database constraints like foreign keys are respected throughout the transaction.
Isolation	Ensures that concurrent transactions do not interfere with each other.	Two users updating the same account will not affect each other’s updates, and transactions will appear sequential.
Durability	Ensures that once a transaction is committed, the data will persist even after a crash.	After a successful transaction, such as adding a new record, the data will remain in the database permanently, even in case of a system failure.

5. DCL (Data Control Language)

i. GRANT

- The `GRANT` command is used to give privileges (permissions)

to users on database objects
like tables, views, and procedures.
- Privileges include actions
like `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and more.

```
GRANT privilege_name ON object_name TO user_name;
```

- Example:

```
GRANT SELECT, INSERT ON Employee TO 'john';
```

  - This command allows the user `john`
  to execute `SELECT` and `INSERT` queries
  on the `Employee` table.

ii. REVOKE
- The `REVOKE` command is used
to remove previously granted privileges from users.
- After privileges are revoked,
users can no longer perform
the associated actions.

```
REVOKE privilege_name ON object_name FROM user_name;
```

- Example:

```
REVOKE INSERT ON Employee FROM 'john';
```

  - This command removes the `INSERT` privilege
  from user `john` on the `Employee` table.

iii. Example of DCL in Use:

```
-- Grant SELECT and UPDATE privileges
-- on the Employee table to the user 'alice'
GRANT SELECT, UPDATE ON Employee TO 'alice';

-- Revoke UPDATE privilege from the user 'alice'
REVOKE UPDATE ON Employee FROM 'alice';
```

iv. SQL privileges

Privilege	Description	Example
SELECT	Allows querying or retrieving data from a table/view.	`GRANT SELECT ON Employee TO 'john';`
INSERT	Allows inserting new data into a table.	`GRANT INSERT ON Employee TO 'john';`
UPDATE	Allows modifying existing data in a table.	`GRANT UPDATE ON Employee TO 'john';`
DELETE	Allows deleting rows from a table.	`GRANT DELETE ON Employee TO 'john';`
ALTER	Allows altering the structure of a table (e.g., columns).	`GRANT ALTER ON Employee TO 'john';`
INDEX	Allows creating and dropping indexes on a table.	`GRANT INDEX ON Employee TO 'john';`
CREATE	Allows creating new tables, views, or databases.	`GRANT CREATE ON DATABASE company TO 'john';`
DROP	Allows deleting tables or databases.	`GRANT DROP ON Employee TO 'john';`
REFERENCES	Allows creating foreign key constraints.	`GRANT REFERENCES ON Employee TO 'john';`
EXECUTE	Allows executing stored procedures or functions.	`GRANT EXECUTE ON PROCEDURE update_salary TO 'john';`
ALL PRIVILEGES	Grants all privileges on a specified object.	`GRANT ALL PRIVILEGES ON Employee TO 'admin';`

6. Functions

i. Date Functions

Date functions are used to manipulate and retrieve date and time values in SQL.

Function	Description	Example
`NOW()`	Returns the current date and time.	`SELECT NOW();`
`CURDATE()`	Returns the current date.	`SELECT CURDATE();`
`CURTIME()`	Returns the current time.	`SELECT CURTIME();`
`DATE()`	Extracts the date part from a datetime expression.	`SELECT DATE('2024-09-04 12:00:00');`
`YEAR()`	Extracts the year from a date.	`SELECT YEAR('2024-09-04');`
`MONTH()`	Extracts the month from a date.	`SELECT MONTH('2024-09-04');`
`DAY()`	Extracts the day from a date.	`SELECT DAY('2024-09-04');`
`DATE_ADD()`	Adds a time interval to a date.	`SELECT DATE_ADD('2024-09-04', INTERVAL 10 DAY);`
`DATEDIFF()`	Returns the difference in days between two dates.	`SELECT DATEDIFF('2024-09-04', '2024-08-01');`
`DAYOFWEEK()`	Returns the day of the week (1 = Sunday, 7 = Saturday).	`SELECT DAYOFWEEK('2024-09-04');`

Function	Description	Example	Output
`DATE_FORMAT()`	Formats a date according to a specified format.	`SELECT DATE_FORMAT('2024-09-04', '%Y-%m-%d');`	`2024-09-04`
`STR_TO_DATE()`	Converts a string to a date using the specified format.	`SELECT STR_TO_DATE('04-09-2024', '%d-%m-%Y');`	`2024-09-04`
`LAST_DAY()`	Returns the last day of the month for a given date.	`SELECT LAST_DAY('2024-09-04');`	`2024-09-30`
`DAYNAME()`	Returns the name of the day for a given date.	`SELECT DAYNAME('2024-09-04');`	`Wednesday`
`DAYOFYEAR()`	Returns the day of the year (1–366) for a given date.	`SELECT DAYOFYEAR('2024-09-04');`	`248`
`WEEK()`	Returns the week number of the year for a given date.	`SELECT WEEK('2024-09-04');`	`36`
`ADDDATE()`	Adds a specified number of days to a date.	`SELECT ADDDATE('2024-09-04', INTERVAL 10 DAY);`	`2024-09-14`
`SUBDATE()`	Subtracts a specified number of days from a date.	`SELECT SUBDATE('2024-09-04', INTERVAL 5 DAY);`	`2024-08-30`
`EXTRACT()`	Extracts a part (e.g., year, month, day) from a date.	`SELECT EXTRACT(YEAR FROM '2024-09-04');`	`2024`
`TIMESTAMPDIFF()`	Returns the difference between two dates in the specified unit (e.g., days, months).	`SELECT TIMESTAMPDIFF(DAY, '2024-09-04', '2024-09-14');`	`10`

`TIMESTAMPADD()`	Adds an interval to a date and returns the result.	`SELECT TIMESTAMPADD(MONTH, 1, '2024-09-04');`	`2024-10-04`
`TO_DAYS()`	Returns the total number of days between a date and year 0.	`SELECT TO_DAYS('2024-09-04');`	`738011`

ii. String Functions

String functions are used
to manipulate or retrieve data from text (string) values.

Function	Description	Example
`LENGTH()`	Returns the length of a string.	`SELECT LENGTH('SQL Tutorial');`
`LOWER()`	Converts a string to lowercase.	`SELECT LOWER('HELLO');`
`UPPER()`	Converts a string to uppercase.	`SELECT UPPER('hello');`
`SUBSTRING()`	Extracts a substring from a string.	`SELECT SUBSTRING('SQL Tutorial', 5, 7);`
`CONCAT()`	Concatenates two or more strings.	`SELECT CONCAT('SQL', ' ', 'Tutorial');`
`TRIM()`	Removes leading and trailing spaces from a string.	`SELECT TRIM('  SQL  ');`
`REPLACE()`	Replaces occurrences of a substring within a string.	`SELECT REPLACE('Hello World', 'World', 'SQL');`
`LEFT()`	Returns the left part of a string with a given length.	`SELECT LEFT('SQL Tutorial', 3);`
`RIGHT()`	Returns the right part of a string with a given length.	`SELECT RIGHT('SQL Tutorial', 4);`
`INSTR()`	Returns the position of the first occurrence of a substring.	`SELECT INSTR('SQL Tutorial', 'Tut');`

iii. Math Functions

Math functions are used
to perform mathematical calculationsin SQL.

Function	Description	Example
`ABS()`	Returns the absolute value of a number.	`SELECT ABS(-10);`
`ROUND()`	Rounds a number to a specified number of decimal places.	`SELECT ROUND(123.4567, 2);`

`CEIL()` / `CEILING()`	Returns the smallest integer greater than or equal to a number.	`SELECT CEIL(4.2);`
`FLOOR()`	Returns the largest integer less than or equal to a number.	`SELECT FLOOR(4.8);`
`MOD()`	Returns the remainder of a division operation.	`SELECT MOD(10, 3);`
`SQRT()`	Returns the square root of a number.	`SELECT SQRT(16);`
`POWER()`	Raises a number to the power of another number.	`SELECT POWER(2, 3);`
`EXP()`	Returns e raised to the power of a number.	`SELECT EXP(1);`
`PI()`	Returns the value of pi (π).	`SELECT PI();`
`RANDOM()`	Returns a random number.	`SELECT RAND();`

iv. Other Functions (Non-Aggregate)

These functions don't fall into specific categories
like date, string, or math
but are useful in various contexts.

Function	Description	Example
`IFNULL()`	Returns an alternate value if an expression is `NULL`.	`SELECT IFNULL(NULL, 'Unknown');`
`COALESCE()`	Returns the first non-NULL value in a list.	`SELECT COALESCE(NULL, NULL, 'Value');`
`CASE`	Performs conditional logic in a query.	`SELECT CASE WHEN age >= 18 THEN 'Adult' ELSE 'Child' END;`
`CAST()`	Converts one data type to another.	`SELECT CAST('123' AS INT);`
`CONVERT()`	Converts a value from one data type to another.	`SELECT CONVERT('2024-09-04', DATE);`
`ISNULL()`	Checks if a value is `NULL`.	`SELECT ISNULL(NULL);`
`GREATEST()`	Returns the largest value in a list of expressions.	`SELECT GREATEST(10, 20, 30);`
`LEAST()`	Returns the smallest value in a list of expressions.	`SELECT LEAST(10, 20, 30);`

7. Selectors

i. `SELECT`

The `SELECT` clause specifies which columns to retrieve from a table.

- Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

- Example:

```
SELECT first_name, last_name
FROM employees;
```

ii. `DISTINCT`

The `DISTINCT` keyword is used t o return only unique values, removing duplicates.

- Syntax:

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

- Example:

```
SELECT DISTINCT department
FROM employees;
```

iii. `WHERE`
The `WHERE` clause filters records based on specified conditions.

- Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- Example:

```
SELECT first_name, salary
FROM employees
WHERE salary > 50000;
```

iv. `ORDER BY`
The `ORDER BY` clause sorts the result set in ascending or descending order.

- Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC];
```

- Example:

```
SELECT first_name, salary
FROM employees
ORDER BY salary DESC;
```

v. `GROUP BY`

The `GROUP BY` clause groups rows that have the same values into summary rows, like finding the sum or average.

- Syntax:

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

- Example:

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

vi. `HAVING`

The `HAVING` clause filters records after the `GROUP BY` clause has been applied. It is similar to the `WHERE` clause but used for aggregated data.

- Syntax:

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

- Example:

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department
HAVING AVG(salary) > 60000;
```

vii. `JOIN`

The `JOIN` clause is used
to combine rows from two or more tables
based on a related column.

- Syntax:

```
SELECT columns
FROM table1
JOIN table2
ON table1.column = table2.column;
```

- Example (Inner Join):

```
SELECT employees.first_name, departments.department_name
```

```
FROM employees
JOIN departments
ON employees.department_id = departments.department_id;
```

viii. `UNION`

The `UNION` operator combines the results of
two or more `SELECT` statements and removes duplicates.
Use `UNION ALL` to include duplicates.

- Syntax:

```
SELECT column1, column2
FROM table1
UNION
SELECT column1, column2
FROM table2;
```

- Example:

```
SELECT first_name FROM employees
UNION
SELECT manager_name FROM managers;
```

ix. `LIMIT`

The `LIMIT` clause specifies the number of records to return.

- Syntax:

```
SELECT column1, column2, ...
FROM table_name
LIMIT number;
```

- Example:

```
SELECT first_name
FROM employees
LIMIT 10;
```

x. `OFFSET`

The `OFFSET` clause skips a specific number of records before starting to return rows.

- Syntax:

```
SELECT column1, column2, ...
FROM table_name
OFFSET number;
```

- Example:

```
SELECT first_name
FROM employees
LIMIT 10 OFFSET 20;
```

8. Arithmetic Operators and Columns Selectors

Arithmetic Operators

Operator	Description	Syntax	Example	Output
`+`	Addition: Adds two numeric values.	`SELECT column1 + column2 AS result`	`SELECT salary + bonus AS total_compensation`	`50000`
`-`	Subtraction: Subtracts one numeric value from another.	`SELECT column1 - column2 AS result`	`SELECT salary - deductions AS net_salary`	`45000`
`*`	Multiplication: Multiplies two numeric values.	`SELECT column1 * column2 AS result`	`SELECT quantity * unit_price AS total_price`	`2000`
`/`	Division: Divides one numeric value by another.	`SELECT column1 / column2 AS result`	`SELECT total_sales / number_of_orders AS average_order_value`	`150`
`%`	Modulo: Returns the remainder of a division operation.	`SELECT column1 % column2 AS result`	`SELECT total_items % items_per_box AS remaining_items`	`5`
`^`	Exponentiation: Raises one number to the power of another (not supported in all SQL databases).	`SELECT POW(column1, column2) AS result`	`SELECT POW(2, 3) AS result`	`8`

i. Basic Column Selection

Selecting specific columns:

```
SELECT column1, column2, ...
FROM table_name;
```

- Example:

```
SELECT first_name, last_name
FROM employees;
```

ii. Aliasing Columns

Using aliases to rename columns in the result set:

```
SELECT column1 AS alias_name1, column2 AS alias_name2
FROM table_name;
```

- Example:

```
SELECT first_name AS 'First Name', last_name AS 'Last Name'
FROM employees;
```

iii. Selecting All Columns

Selecting all columns from a table:

```
SELECT *
FROM table_name;
```

- Example:

```
SELECT *
FROM employees;
```

iv. Using Expressions in Column Selection

Using expressions or calculations in column selection:

```
SELECT column1, (column2 * 1.1) AS adjusted_column
FROM table_name;
```

- Example:

```
SELECT first_name, salary * 1.1 AS adjusted_salary
FROM employees;
```

v. Selecting Distinct Values

Selecting distinct values to remove duplicates:

```
SELECT DISTINCT column1, column2
FROM table_name;
```

- Example:

```
SELECT DISTINCT department
FROM employees;
```

vi. Selecting Based on Conditions

Selecting columns with conditional logic:

```
SELECT column1, column2
FROM table_name
WHERE condition;
```

- Example:

```
SELECT first_name, salary
FROM employees
WHERE salary > 50000;
```

vii. Aggregating Columns

Using aggregate functions to summarize data:

```
SELECT aggregate_function(column)
FROM table_name;
```

- Examples:

```
SELECT COUNT(*)
FROM employees;

SELECT AVG(salary)
FROM employees;
```

viii. Grouping and Aggregating

Selecting columns and using `GROUP BY` to aggregate data:

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

- Example:

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

ix. Combining Results with `UNION`

Selecting columns from multiple queries and combining results:

```
SELECT column1, column2
FROM table1
UNION
SELECT column1, column2
FROM table2;
```

- Example:

```
SELECT first_name FROM employees
UNION
SELECT manager_name FROM managers;
```

x. Limiting Results

Selecting a subset of rows:

```
SELECT column1, column2
FROM table_name
LIMIT number;
```

- Example:

```
SELECT first_name
FROM employees
LIMIT 10;
```

xi. Skipping Rows

Skipping a specific number of rows and then selecting:

```
SELECT column1, column2
FROM table_name
LIMIT number OFFSET number;
```

- Example:

```
SELECT first_name
FROM employees
LIMIT 10 OFFSET 20;
```

9. Filtering with `WHERE` Condition

The `WHERE` clause is used to filter records that meet certain criteria.

Condition	Description	Syntax	Example	Output
Basic Condition	Filters rows based on a simple condition.	`SELECT column1 FROM table_name WHERE condition;`	`SELECT first_name FROM employees WHERE salary > 50000;`	Names of employees with salary > 50000

Multiple Conditions	Filters rows based on multiple conditions combined with logical operators.	`SELECT column1 FROM table_name WHERE condition1 AND/OR condition2;`	`SELECT first_name FROM employees WHERE salary > 50000 AND department = 'IT';`	Names of employees in IT with salary > 50000

Comparison Operators

Comparison operators are used in the `WHERE` clause to compare values.

Operator	Description	Syntax	Example	Output
`=`	Equals	`column1 = value`	`SELECT * FROM employees WHERE department = 'IT';`	Employees in the IT department
`!=` or `<>`	Not equals	`column1 != value` or `column1 <> value`	`SELECT * FROM employees WHERE department != 'HR';`	Employees not in the HR department
`>`	Greater than	`column1 > value`	`SELECT * FROM employees WHERE salary > 60000;`	Employees with salary greater than 60000
`<`	Less than	`column1 < value`	`SELECT * FROM employees WHERE salary < 40000;`	Employees with salary less than 40000
`>=`	Greater than or equal to	`column1 >= value`	`SELECT * FROM employees WHERE salary >= 50000;`	Employees with salary greater than or equal to 50000
`<=`	Less than or equal to	`column1 <= value`	`SELECT * FROM employees WHERE salary <= 30000;`	Employees with salary less than or equal to 30000

Logical Operators

Logical operators are used to combine multiple conditions.

Operator	Description	Syntax	Example	Output
`AND`	Returns true if both conditions are true.	`condition1 AND condition2`	`SELECT * FROM employees WHERE salary > 50000 AND department = 'IT';`	Employees with salary > 50000 and in IT department
`OR`	Returns true if at least one of the conditions is true.	`condition1 OR condition2`	`SELECT * FROM employees WHERE salary > 50000 OR department = 'HR';`	Employees with salary > 50000 or in HR department
`NOT`	Reverses the logical value of the condition.	`NOT condition`	`SELECT * FROM employees WHERE NOT department = 'IT';`	Employees not in the IT department
`BETWEEN`	Returns true if the value is within a specified range.	`column BETWEEN value1 AND value2`	`SELECT * FROM employees WHERE salary BETWEEN 40000 AND 60000;`	Employees with salary between 40000 and 60000
`LIKE`	Searches for a specified pattern.	`column LIKE pattern`	`SELECT * FROM employees WHERE first_name LIKE 'J%';`	Employees whose first name starts with 'J'
`IN`	Checks if a value is within a set of values.	`column IN (value1, value2, ...)`	`SELECT * FROM employees WHERE department IN ('IT', 'HR');`	Employees in either IT or HR department

Other Operators

Other operators used in SQL queries:

Operator	Description	Syntax	Example	Output
`IS NULL`	Checks if a column value is `NULL`.	`column IS NULL`	`SELECT * FROM employees WHERE middle_name IS NULL;`	Employees with no middle name

`IS NOT NULL`	Checks if a column value is not `NULL`.	`column IS NOT NULL`	`SELECT * FROM employees WHERE middle_name IS NOT NULL;`	Employees with a middle name
`EXISTS`	Tests for the existence of any record in a subquery.	`EXISTS (subquery)`	`SELECT * FROM employees WHERE EXISTS (SELECT 1 FROM departments WHERE employees.department_id = departments.id);`	Employees with existing departments
`ALL`	Compares a value to all values in another set or subquery.	`column operator ALL (subquery)`	`SELECT * FROM employees WHERE salary > ALL (SELECT salary FROM employees WHERE department = 'HR');`	Employees with salary greater than all in HR
`ANY`	Compares a value to any value in another set or subquery.	`column operator ANY (subquery)`	`SELECT * FROM employees WHERE salary > ANY (SELECT salary FROM employees WHERE department = 'HR');`	Employees with salary greater than any in HR

10. Sorting Data with `ORDER BY` Clause

The `ORDER BY` clause is used
to sort the result set of a query by one or more columns.
You can specify the sorting order as ascending or descending.

Sorting Option	Description	Syntax	Example	Output
`ORDER BY`	Specifies the column(s) to sort the result set.	`SELECT columns FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];`	`SELECT * FROM employees ORDER BY salary DESC, first_name ASC;`	Employees sorted by salary descending and then first name ascending
`ASC`	Sorts the result set in ascending order (default).	`ORDER BY column ASC`	`SELECT * FROM employees ORDER BY salary ASC;`	Employees sorted by salary ascending
`DESC`	Sorts the result set in descending order.	`ORDER BY column DESC`	`SELECT * FROM employees ORDER BY salary DESC;`	Employees sorted by salary descending
Ordinal Number	Sorts by the position of the column in the `ORDER BY` clause.	`ORDER BY 1, 2` (where `1` and `2` are column positions)	`SELECT * FROM employees ORDER BY 2 DESC, 1 ASC;`	Employees sorted by second column descending, first column ascending

AliasName	Uses an alias for sorting columns.	`ORDER BY alias_name`	`SELECT first_name AS name, salary AS income FROM employees ORDER BY income DESC;`	Employees sorted by salary (income) descending

Explanation of Syntax and Examples

i. Basic Sorting:

- Syntax:

```
SELECT column1, column2
FROM table_name
ORDER BY column1 [ASC|DESC];
```

- Example:

```
SELECT first_name, salary
FROM employees
ORDER BY salary DESC;
```

- Output: Employees are listed with the highest salary first.

ii. Sorting by Multiple Columns:

- Syntax:

```
SELECT column1, column2
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

- Example:

```
SELECT first_name, department, salary
FROM employees
```

```
ORDER BY department ASC, salary DESC;
```

- Output: Employees are first sorted by department in ascending order; within each department, employees are sorted by salary in descending order.

iii. Sorting Using Ordinal Numbers:

- Syntax:

```
SELECT column1, column2, column3
FROM table_name
ORDER BY 1 [ASC|DESC], 2 [ASC|DESC];
```

- Example:

```
SELECT first_name, department, salary
FROM employees
ORDER BY 3 DESC, 1 ASC;
```

- Output: Employees are sorted by the third column (salary) in descending order,
then by the first column (first name) in ascending order.

iv. Sorting with Aliases:

- Syntax:

```
SELECT column1 AS alias_name1, column2 AS alias_name2
FROM table_name
ORDER BY alias_name1 [ASC|DESC];
```

- Example:

```
SELECT first_name AS name, salary AS income
FROM employees
ORDER BY income DESC;
```

- Output: Employees are sorted by the alias `income` (which represents salary) in descending order.

11. MySQL Aggregate Functions

Aggregate functions are used to perform calculations on multiple rows of a table's column and return a single value.

Function	Description	Syntax	Example	Output
`MIN()`	Returns the minimum value in a set of values.	`MIN(column)`	`SELECT MIN(salary) FROM employees;`	Minimum salary from the employees table
`MAX()`	Returns the maximum value in a set of values.	`MAX(column)`	`SELECT MAX(salary) FROM employees;`	Maximum salary from the employees table
`SUM()`	Returns the sum of all values in a numeric column.	`SUM(column)`	`SELECT SUM(salary) FROM employees;`	Total salary of all employees
`AVG()`	Returns the average value of a numeric column.	`AVG(column)`	`SELECT AVG(salary) FROM employees;`	Average salary of all employees
`COUNT(*)`	Returns the number of rows in a table.	`COUNT(*)`	`SELECT COUNT(*) FROM employees;`	Total number of rows in the employees table
`COUNT(column)`	Returns the number of non-NULL values in a column.	`COUNT(column)`	`SELECT COUNT(salary) FROM employees;`	Number of employees with a salary specified

| `COUNT(DISTINCT column)` | Returns the number of distinct non-NULL values in a column. | `COUNT(DISTINCT column)` | `SELECT COUNT(DISTINCT department) FROM employees;` | Number of distinct departments |

|----------------------|---|---|---|---|

i. `MIN()` Function:

- Syntax:

```
SELECT MIN(column) FROM table_name;
```

- Example:

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

- Output: Returns the lowest salary from the `employees` table.

ii. `MAX()` Function:

- Syntax:

```
SELECT MAX(column) FROM table_name;
```

- Example:

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

- Output: Returns the highest salary from the `employees` table.

iii. `SUM()` Function:

- Syntax:

```
SELECT SUM(column) FROM table_name;
```

- Example:

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

- Output: Returns the total sum of salaries from the `employees` table.

iv. `AVG()` Function:

- Syntax:

```
SELECT AVG(column) FROM table_name;
```

- Example:

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

- Output: Returns the average salary from the `employees` table.

v. `COUNT(*)` Function:

- Syntax:

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

- Example:

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

- Output: Returns the total number of rows in the `employees` table.

vi. `COUNT(column)` Function:

- Syntax:

```
SELECT COUNT(column) FROM table_name;
```

- Example:

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

- Output: Returns the number of non-NULL `salary` values in the `employees` table.

vii. `COUNT(DISTINCT column)` Function:

- Syntax:

```
SELECT COUNT(DISTINCT column) FROM table_name;
```

- Example:

```
SELECT COUNT(DISTINCT department) FROM employees;
```

- Output: Returns the number of distinct departments in the `employees` table.

12. `GROUP BY` and `HAVING` Clauses

`GROUP BY` Clause

The `GROUP BY` clause is used to group rows
that have the same values into summary rows,
like "total salary per department".
It is often used with aggregate functions
like `SUM()`, `COUNT()`, `AVG()`, `MIN()`, and `MAX()`.

Clause	Description	Syntax	Example	Output

`GROUP BY`	Groups rows that have the same values into aggregated rows.	`SELECT column1, AGGREGATE_FUNCTION(column2) FROM table_name GROUP BY column1;`	`SELECT department, COUNT(*) FROM employees GROUP BY department;`	Number of employees in each department

`HAVING` Clause

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

Clause	Description	Syntax	Example	Output
`HAVING`	Filters groups based on a specified condition after aggregation.	`SELECT column1, AGGREGATE_FUNCTION(column2) FROM table_name GROUP BY column1 HAVING condition;`	`SELECT department, COUNT(*) FROM employees GROUP BY department HAVING COUNT(*) > 5;`	Departments with more than 5 employees

1. Basic `GROUP BY` Usage:
   - Syntax:

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

   - Example:

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department;
```

- Output: Lists the average salary for each department.

2. `GROUP BY` with `HAVING`:
   - Syntax:

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

   - Example:

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department
HAVING COUNT(*) > 10;
```

   - Output: Lists departments that have more than 10 employees.

3. Using `GROUP BY` with Multiple Columns:
   - Syntax:

```
SELECT column1, column2, AGGREGATE_FUNCTION(column3)
FROM table_name
GROUP BY column1, column2;
```

- Example:

```
SELECT department, job_title, AVG(salary)
FROM employees
GROUP BY department, job_title;
```

- Output: Lists the average salary for each combination of department and job title.

4. Combining `GROUP BY` and `HAVING` with Multiple Conditions:
   - Syntax:

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

- Example:

```
SELECT department, SUM(salary)
FROM employees
GROUP BY department
HAVING SUM(salary) > 100000;
```

- Output: Lists departments where the total salary is greater than 100,000.

13. Types of Joins
Joins are used
to combine rows from two or more tables based
on a related column between them.

```
|-----------------|----------------------------------------------------------|----------------------------------------------------------------------------|
```

Join Type	Returns	Syntax Example
INNER JOIN	Rows with matching values in both tables.	`SELECT * FROM employees INNER JOIN departments ON employees.department_id = departments.id;`
LEFT JOIN	All rows from the left table, matched rows from the right.	`SELECT * FROM employees LEFT JOIN departments ON employees.department_id = departments.id;`
RIGHT JOIN	All rows from the right table, matched rows from the left.	`SELECT * FROM employees RIGHT JOIN departments ON employees.department_id = departments.id;`
FULL OUTER JOIN	All rows from both tables, NULLs where no match.	`SELECT * FROM employees LEFT JOIN departments ON employees.department_id = departments.id UNION SELECT * FROM employees RIGHT JOIN departments ON employees.department_id = departments.id;`
CROSS JOIN	Cartesian product of both tables.	`SELECT * FROM employees CROSS JOIN departments;`

i. INNER JOIN

- Description: The `INNER JOIN` keyword returns records
that have matching values in both tables.
If there is no match, the row is not included in the result set.

- Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

- Example:

```
SELECT employees.name, departments.name
FROM employees
```

```
INNER JOIN departments
ON employees.department_id = departments.id;
```

- Output: This query returns a list of employee names along
with their respective department names,
but only for employees who have a matching department
in the `departments` table.

ii. LEFT OUTER JOIN (LEFT JOIN)

- Description: The `LEFT OUTER JOIN` returns all rows from the left table,
and the matched rows from the right table.
If no match is found, NULL values are returned for columns from the right table.

- Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

- Example:

```
SELECT employees.name, departments.name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.id;
```

- Output: This query returns a list of all employees,

along with their department names.
If an employee does not belong to any department
(i.e., no match in the `departments` table),
the department name will be NULL.

iii. RIGHT OUTER JOIN (RIGHT JOIN)

- Description: The `RIGHT OUTER JOIN` returns all rows from the right table,
and the matched rows from the left table.
If no match is found, NULL values are returned for columns from the left table.

- Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

- Example:

```
SELECT employees.name, departments.name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.id;
```

- Output: This query returns a list of all departments,
along with the names of employees in those departments.
If a department has no employees (i.e., no match in the `employees` table),
the employee name will be NULL.

iv. FULL OUTER JOIN

- Description: The `FULL OUTER JOIN` returns all rows from both tables,
  with NULLs where there is no match.
  MySQL does not directly support `FULL OUTER JOIN`,
  but you can simulate it using a combination of `LEFT JOIN` and `RIGHT JOIN` with `UNION`.

- Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column
UNION
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

- Example:

```
SELECT employees.name, departments.name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.id
UNION
SELECT employees.name, departments.name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.id;
```

- Output: This query returns all employees and all departments, with NULLs where there are no matching rows in the other table.

v. CROSS JOIN

- Description:

  The `CROSS JOIN` returns the Cartesian product of the two tables. This means it will return all possible combinations of rows from both tables.

- Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

- Example:

```
SELECT employees.name, departments.name
FROM employees
CROSS JOIN departments;
```

- Output: This query returns a list of every possible combination of employee names and department names. If there are 10 employees and 5 departments, the result will contain 50 rows (10 x 5).