1. What is SQL?

Ans: 1. Structures Query Language 2. SQL is a language used to interact with the database.

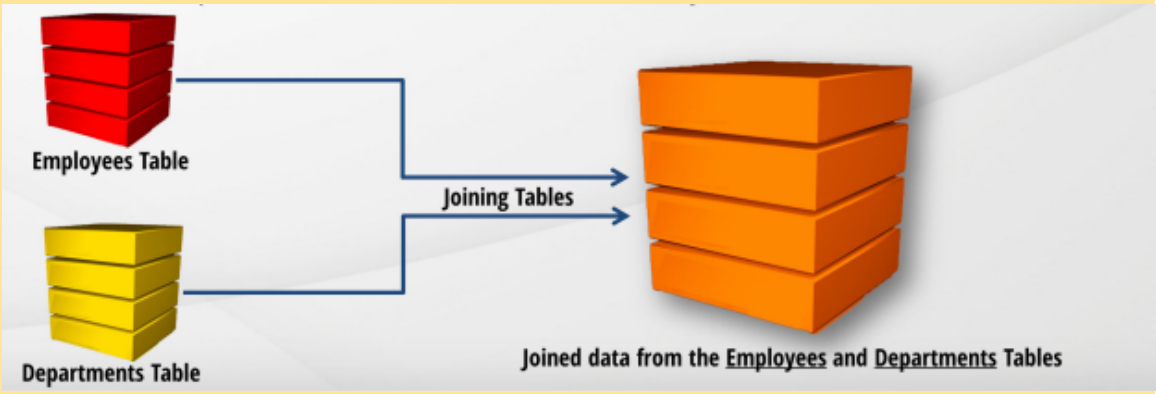
2. Where do we use SQL?

Ans: BI, Data Science, Database Administration, Web Development, etc…

3. SQL Statements:

DML	**Data Manipulation Language** DML is modifying, when you want to modify the data records, but are not allowed to modify the structure of tables and it is used to access data from the database	**SELECT**
		INSERT
		UPDATE
		DELETE
		MERGE(Oracle)
DDL	**Data Definition Language** DDL is, if you want to define the structure of the database and integrity constraints like primary key, alternate key, and foreign key then we are ging to use DDL so, basically when you want to create some table then you are going to use this.	**CREATE**
		ALTER
		DROP
		RENAME
		TRUNCATE
DCL	**Data Control Language** DCL means we have to do something called transactions, lock, shared lock, exclusive lock, commit, rollback, and data control for security so we are going to have grant revoke. So, DCL is used for Consistency and used for security.	**GRANT**
		REVOKE
TCL	**Transaction Control Language**	**COMMIT**
		ROLLBACK
		SAVEPOINT

4. What is a join?



Ans: 1. A join is a concept that allows us to retrieve data from two or more tables in a single query. 2. In SQL, we often need to write queries to get data from two or more tables. Anything but the simplest of queries will usually need data from two or more tables, and to get data from multiple tables, we need to use the joins.

5. What is the purpose of the SELECT statement in MySQL?

Ans: The SELECT statement in MySQL is used to retrieve data from one or more tables in a database. It allows you to specify which columns of data you want to see.

Specific Purpose:

- **Retrieve data:** The primary function of SELECT is to extract information from database tables.
- **Filter data:** You can use WHERE clauses to specify conditions that must be met for rows to be included in the result set.
- **Transform data:** Functions like SUM, AVG, COUNT, MIN, and MAX can be used to perform calculations on the retrieved data.
- **Join tables:** The JOIN keyword allows you to combine data from multiple tables based on related columns.
- **Order results:** The ORDER BY clause can be used to sort the results based on specific columns.
- **Limit results:** The LIMIT clause specifies the maximum number of rows to return.

Real-time Example:

Imagine you have a table called employees with columns like name, position, and salary. If you want to see the names and positions of all employees, you would use:

SELECT name, position

FROM employees;

6. What is Normalization?

Ans: Normalization in a database is the process of organizing data to minimize redundancy and ensure data integrity. It involves dividing large tables into smaller ones and defining relationships between them, making the database more efficient and easier to maintain.

Real-time Scenario:

Imagine you run a **training institute** and have a table storing **student data** that includes:

Student_ID	Student_Name	Course	Instructor	Instructor_Email
1	Alice	Java	John	john@example.com
2	Bob	Java	John	john@example.com
3	Charlie	Python	Sara	sara@example.com

Here, the instructor's information is repeated for every student taking the same course. If John's email changes, you'll need to update it in multiple places.

1. Student Table:

Student_ID	Student_Name	Course
1	Alice	Java
2	Bob	Java
3	Charlie	Python

2. Instructor Table:

Instructor	Instructor_Email
John	john@example.com
Sara	sara@example.com

To normalize this data (e.g., using 2nd Normal Form), you'd split it into two tables: see above two tables 1&2

Now, instructor information is stored only once, and any updates are made in one place, improving consistency and efficiency.

7. What is the different datatype in MySQL?

Ans: In MySQL, data types are categorized into the following main types:

1. **Numeric Types**:
   - INT: Integer values (e.g., 10, 200).
   - FLOAT, DOUBLE: Floating-point numbers (e.g., 1.23, 45.67).
   - DECIMAL: Fixed-point numbers (e.g., 123.45 for precise calculations like currency).
2. **String Types**:
   - VARCHAR: Variable-length strings (e.g., names, addresses).
   - CHAR: Fixed-length strings.
   - TEXT: Large text data.
3. **Date and Time Types**:
   - DATE: Stores date (e.g., '2024-09-27').
   - DATETIME: Stores both date and time (e.g., '2024-09-27 10:30:00').

These are the common data types used based on the kind of data you need to store.

8. What is the difference between a primary key and a unique key?

Ans: The **primary key** and **unique key** in MySQL both ensure uniqueness, but they have key differences:

1. **Primary Key**:
   - Uniquely identifies each record in a table.
   - **Cannot contain NULL values**.
   - A table can have only **one primary key**.
2. **Unique Key**:
   - Ensures all values in a column are unique.
   - **Can contain NULL values**.
   - A table can have **multiple unique keys**.

NOTE:

- **Primary Key**: One per table, no NULLs.
- **Unique Key**: Multiple allowed, can have NULLs.

9. Foreign key constraint?

Ans: A **foreign key constraint** in MySQL ensures that a value in one table corresponds to a value in another table, maintaining referential integrity between them.

Real-time Scenario:

In a **school database**, you have two tables:

1. **Students** table:

Student_ID	Name	Course_ID
1	Alice	101
2	Bob	102

2. **Courses** table:

Course_ID	Course_Name
101	Maths
102	science

The Course_ID in the **Students** table is a **foreign key** referencing the Course_ID in the **Courses** table. This ensures that students are only assigned to valid courses existing in the Courses table.

10. The difference between NULL and zero in MySQL is that:

1. **NULL**: Represents the absence of a value, or an unknown/missing value.
2. **Zero (0)**: Represents a definite value of zero, a numeric value.

Real-time Scenario in a Spring Boot Project:

Imagine you have a **payment** table in your Spring Boot application to store payment amounts:

Payment_ID	Amount	Status
1	1000	Paid
2	NULL	Pending
3	0	Failed

- **NULL (Amount: NULL)**: This means no payment has been made yet (missing value).

  For example: NULL+1=NULL

- **Zero (Amount: 0)**: This means the payment was attempted but failed, or no amount was charged. For example: 0+1=1

In your code, checking for NULL and 0 values would have different meanings when deciding the status of the payment.

11. What is a Database transaction?

Ans: A **database transaction** is a sequence of operations performed as a single logical unit of work, where either all operations succeed or none do. It ensures **ACID properties** (Atomicity, Consistency, Isolation, Durability).

Real-time Scenario:

In a **banking application** built with Spring Boot, when a user transfers money:

1. **Debit** amount from the sender's account.
2. **Credit** amount to the receiver's account.

Both operations must succeed together. If one fails (e.g., debit succeeds but credit fails), the transaction **rolls back**, ensuring no partial updates occur.

This prevents issues like money being deducted from one account without being added to the other.

12. Difference between INNER JOIN and NATURAL JOIN:

1. **INNER JOIN**: Returns records that have matching values in both tables based on a specified condition. You explicitly define the columns to join on.

   Example:
   ```
   SELECT * FROM employees e

   INNER JOIN departments d ON e.department_id = d.department_id;
   ```

2. **NATURAL JOIN**: Automatically joins tables based on columns with the same name and data type in both tables, without needing to specify the condition.

   Example:
   ```
   SELECT * FROM employees NATURAL JOIN departments;
   ```

NOTE:

- **INNER JOIN**: You specify the joining condition.
- **NATURAL JOIN**: Automatically matches columns with the same name in both tables.

13. How do you perform a self-join in MYSQL?

Ans: Self-join is a technique for combining rows from the same table based on a related column, typically with the help of an alias. In MYSQL you can perform a self-join using the following syntax:

```
SELECT A.column1, A.column2, B.column1, B.column2

FROM table_name AS A

JOIN table_name AS B

ON A.related_column = B.related_column;
```

14. What is a trigger, and how do you create one in MySQL?

Ans: A **trigger** is a special piece of code in a MySQL database that runs automatically in response to certain actions, such as adding, updating, or deleting data. Triggers help ensure that data remains accurate and consistent, enforcing rules without needing to manually write code every time.

Why Use Triggers?

- **Maintain Data Integrity**: They help keep your data consistent.
- **Enforce Business Rules**: Automatically perform actions based on specific conditions.
- **Automate Processes**: Save time by automating routine tasks.

Real-time Scenario:

Imagine you're running an **e-commerce store** with a table called orders. Whenever a new order is placed, you want to automatically log this action in an order_logs table for tracking purposes.

Steps to Create a Trigger:

1. **Choose the Event**: You want the trigger to activate on an **INSERT** event when a new order is added.
2. **Specify Timing**: The trigger should execute **AFTER** the order is inserted.
3. **Define the Table**: The trigger will be associated with the orders table.
4. **Write the Action**: You'll log the new order details in the order_logs table.

Example of Creating a Trigger:

Here's how you would write the SQL to create this trigger:

```
CREATE TRIGGER log_order_insert

AFTER INSERT ON orders

FOR EACH ROW

BEGIN

   INSERT INTO order_logs (Order_ID, Action, Timestamp)

   VALUES (NEW.Order_ID, 'Order Placed', NOW());

END;
```

Terms used in the above example:

- **CREATE TRIGGER log_order_insert**: This names the trigger.
- **AFTER INSERT ON orders**: This sets the trigger to run after a new order is added.
- **FOR EACH ROW**: This means the trigger will execute for every row affected by the insert.

- **BEGIN ... END**: This section contains the code to execute, which logs the order details into the order_logs table.

NOTE:

With this trigger in place, every time a new order is added to the orders table, the system automatically records this action in the order_logs table, making it easier to track orders without manual intervention. This ensures you have a complete and accurate log of all transactions.

15. What is the stored procedure, and how do you create one in MySQL?

What is a Stored Procedure?

A **stored procedure** is a set of pre-defined SQL commands stored in the database. It can be executed multiple times by different applications, helping to improve performance and ensure consistency in operations. Stored procedures can accept input parameters, return results, and perform various data manipulation tasks.

Why Use Stored Procedures?

- **Reusability**: Write the code once and use it many times.
- **Performance**: Reduces the amount of code sent over the network and optimizes execution.
- **Consistency**: Ensures that the same logic is applied whenever the procedure is called.

Real-time Scenario:

Imagine you're working on an **e-commerce application**. You need to calculate and apply a discount to a customer's order frequently. Instead of writing the discount logic each time you process an order, you can create a stored procedure.

Example of Creating a Stored Procedure:

Here's how you would create a stored procedure to calculate the discount:

1. **Define the Procedure**: You want to create a procedure that takes the order amount and discount rate as inputs.

SQL to Create the Stored Procedure:

```
DELIMITER //

CREATE PROCEDURE ApplyDiscount(IN orderAmount DECIMAL(10, 2), IN discountRate DECIMAL(5, 2))

BEGIN

    DECLARE finalAmount DECIMAL(10, 2);

    SET finalAmount = orderAmount - (orderAmount * discountRate / 100);

    SELECT finalAmount AS Final_Amount;

END //

DELIMITER ;
```

Terms used in the above example:

- **DELIMITER //**: Changes the statement delimiter so that MySQL knows where the procedure ends.
- **CREATE PROCEDURE ApplyDiscount**: This names the procedure ApplyDiscount.
- **(IN orderAmount DECIMAL(10, 2), IN discountRate DECIMAL(5, 2))**: These are the input parameters for the procedure.
- **BEGIN ... END**: This section contains the code that will execute when the procedure is called.
- **SET finalAmount**: This calculates the final amount after applying the discount.

- **SELECT finalAmount AS Final_Amount**: This returns the final amount to the caller.

NOTE:

With this stored procedure, anytime you need to apply a discount to an order, you just call ApplyDiscount with the order amount and discount rate. This ensures that the discount logic is consistent and efficient throughout your application.

16. What is a cursor, and how do you use one in MySQL?

What is a Cursor?

A **cursor** is a database object that allows you to retrieve and manipulate rows from a result set one at a time. Cursors are useful when you need to process complex data or handle large amounts of data in a controlled manner.

Why Use Cursors?

- **Row-by-Row Processing**: Useful for operations that need to be performed on each row individually.
- **Complex Calculations**: Ideal for calculations or actions that depend on the results of previous rows.

Real-time Scenario:

Imagine you are developing a **banking application** where you need to calculate the interest for each customer's account balance on a monthly basis. Instead of processing all accounts at once, you can use a cursor to handle each account one at a time.

Example of Using a Cursor in MySQL:

Create a Sample Table: Let's assume you have a table named accounts that stores customer account details.
Table: accounts

Account_ID	Customer_Name	Balance
1	Alice	1000
2	Bob	2000
3	Charlie	1500

Create a Cursor: Here's how you would define and use a cursor to calculate interest for each account:

```
DELIMITER //

CREATE PROCEDURE CalculateInterest()

BEGIN

    DECLARE done INT DEFAULT FALSE;

    DECLARE account_id INT;

    DECLARE balance DECIMAL(10, 2);

    DECLARE interest DECIMAL(10, 2);

    -- Declare a cursor for selecting accounts

    DECLARE account_cursor CURSOR FOR

    SELECT Account_ID, Balance FROM accounts;
```

```
    -- Declare a CONTINUE HANDLER for the end of the cursor
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    -- Open the cursor
    OPEN account_cursor;
    -- Loop through each row in the cursor
    read_loop: LOOP
        FETCH account_cursor INTO account_id, balance;
        IF done THEN
            LEAVE read_loop;
        END IF;
        -- Calculate interest (e.g., 5% interest)
        SET interest = balance * 0.05;
        -- Output the result (you could also update a table or take other actions)
        SELECT account_id, interest AS Calculated_Interest;
    END LOOP;
    -- Close the cursor
    CLOSE account_cursor;
END //
DELIMITER ;
```

Terms used in the above example:

- **DECLARE**: Variables are declared to hold the values from the cursor.
- **DECLARE account_cursor**: A cursor is defined to select Account_ID and Balance from the accounts table.
- **OPEN account_cursor**: The cursor is opened to start fetching rows.
- **FETCH account_cursor INTO**: Retrieves the next row from the cursor into the declared variables.
- **LOOP**: Iterates through the rows until all are processed.
- **CLOSE account_cursor**: Closes the cursor once processing is complete.

Note:

In this example, the cursor allows you to calculate the interest for each customer's account one by one, making it easy to handle any specific logic needed for each account while ensuring that you don't overwhelm your application with too much data at once.

17. What is a user-defined function, and how do you create one in MySQL?

A **user-defined function (UDF)** in MySQL is a reusable piece of code that you can call within SQL queries. It allows you to perform specific calculations or data manipulations that may be too complex or repetitive to handle with standard SQL commands.

Why Use User-Defined Functions?

- **Modularity**: Encapsulate complex logic that can be reused in multiple queries.
- **Simplicity**: Break down complex operations into simpler, manageable components.

- **Improved Readability**: Makes queries easier to read and understand.

Real-time Scenario:

Imagine you are working on a **sales application** where you need to calculate the total sales tax for different products based on their price and a fixed tax rate. Instead of recalculating this logic every time in your queries, you can create a user-defined function.

Example of Creating a User-Defined Function:

Let's create a function that calculates sales tax based on a given price.

1. **Create the User-Defined Function**: Here's how you would write the SQL to create this function:

```
DELIMITER //

CREATE FUNCTION CalculateSalesTax(price DECIMAL(10, 2))

RETURNS DECIMAL(10, 2)

DETERMINISTIC

BEGIN

   DECLARE tax_rate DECIMAL(5, 2) DEFAULT 0.07; -- 7% sales tax

   RETURN price * tax_rate; -- Calculate and return the sales tax

END //

DELIMITER ;
```

Terms used in the above example:

- **DELIMITER //**: Changes the statement delimiter so that MySQL recognizes where the function ends.
- **CREATE FUNCTION CalculateSalesTax(price DECIMAL(10, 2))**: This defines a new function called CalculateSalesTax that takes one input parameter: the price.
- **RETURNS DECIMAL(10, 2)**: Specifies the data type of the value that the function will return.
- **DETERMINISTIC**: Indicates that the function will always produce the same result for the same input.
- **BEGIN ... END**: This section contains the logic of the function, where we declare the tax rate and calculate the sales tax.
- **RETURN price * tax_rate**: This returns the calculated sales tax.

How to Use the User-Defined Function:

Once the function is created, you can use it in your queries:

```
SELECT Product_Name, Price, CalculateSalesTax(Price) AS Sales_Tax

FROM products;
```

NOTE:

In this example, the CalculateSalesTax function allows you to easily compute sales tax for any product price without rewriting the logic each time. This not only saves time but also makes your SQL queries cleaner and easier to understand.

18. What are aggregate functions in SQL?

In SQL, **aggregate functions** are used to perform calculations on multiple rows of data, returning a single result. They are commonly used with the GROUP BY clause to group rows that share a common value into summary rows, but they can also be used without grouping to perform calculations across an entire dataset.

Here are some common aggregate functions in SQL:

- **COUNT()**: Returns the number of rows that match a specified condition. It can count all rows or only rows with a non-NULL value.

```
SELECT COUNT(*) FROM users;        -- Counts all rows

SELECT COUNT(email) FROM users;    -- Counts only rows where 'email' is not NULL
```

- **SUM()**: Returns the total sum of a numeric column .

```
SELECT SUM(salary) FROM employees; -- Adds up all salaries
```

- **AVG()**: Returns the average value of a numeric column.

```
SELECT AVG(age) FROM users; -- Calculates the average age
```

- **MIN()**: Returns the smallest value in a column.

```
SELECT MIN(price) FROM products; -- Finds the lowest price
```

- **MAX()**: Returns the largest value in a column.

```
SELECT MAX(salary) FROM employees; -- Finds the highest salary
```

- **GROUP_CONCAT()** (MySQL-specific): Returns a concatenated string of non-NULL values from a group.

```
SELECT GROUP_CONCAT(name) FROM users; -- Concatenates names into a single string
```

19. What is the difference between WHERE and HAVING clause ?

The **WHERE** and **HAVING** clauses in SQL differ based on when they apply and what they filter.

- **WHERE** is used to filter rows before any grouping or aggregation. It works on individual rows and can only be applied to non-aggregated columns.

```
SELECT name, salary

FROM employees

WHERE salary > 50000;
```

This query retrieves employees with a salary greater than 50,000, filtering individual rows.

- **HAVING** is used to filter after the **GROUP BY** clause, meaning it works on groups of rows. It can filter based on the result of aggregate functions.

```
SELECT department, AVG(salary) AS avg_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 50000;
```

This query groups employees by department, calculates the average salary for each department, and then filters out departments where the average salary is less than or equal to 50,000.

Combining WHERE and HAVING:

They can be used together in queries where you need to filter both rows and groups.

```
SELECT department, SUM(salary) AS total_salary
```

```
FROM employees

WHERE role = 'Engineer'

GROUP BY department

HAVING SUM(salary) > 100000;
```

WHERE filters rows to include only engineers.

HAVING filters the result to show only departments where the total salary of engineers exceeds 100,000.

20. What are indexes in SQL ?

Indexes in SQL are special data structures that improve the speed of data retrieval operations on a database table. They work similarly to the index of a book, which helps you locate information quickly without scanning the entire content.

Indexes are used to make searching and retrieving data faster, particularly for large datasets. Instead of scanning the entire table row by row, the index allows the database to jump to the relevant rows directly.

Types of Indexes:

Single-column index: Created on a single column of a table.

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

Composite (multi-column) index: Created on more than one column.

```
CREATE INDEX idx_name_salary ON employees(name, salary);
```

Unique index: Ensures that all the values in the indexed column are unique (same as a UNIQUE constraint).

```
CREATE UNIQUE INDEX idx_unique_email ON users(email);
```

Internally, indexes often use data structures like **B-trees** or **hash tables** to efficiently locate the rows matching a query. This reduces the need for a full table scan.

21. How can you identify which indexes are being used in a query?

To identify which indexes are being used in a query, you can use **query execution plans** provided by most relational databases. These plans show how the database processes a query and whether indexes are being used.

Here's how you can identify index usage in common databases:

1. Using EXPLAIN or EXPLAIN PLAN

Most SQL databases provide an **EXPLAIN** or **EXPLAIN PLAN** command that displays the query execution plan, detailing the steps the database takes to execute the query, including whether an index is being used.

SQL Example:

In MySQL, you can use the EXPLAIN statement to see if an index is being used:

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

2. Using ANALYZE with EXPLAIN

In some databases like PostgreSQL, you can use **EXPLAIN ANALYZE** to not only see the execution plan but also get runtime statistics for the query:

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

This will display the actual runtime of the query along with the plan, helping you confirm whether an index is being used and how effectively.

22. Can you have an index on a view? If yes, how?

Yes, you can create an index on a view in SQL, but the view must meet certain conditions and be a **materialized view** or an **indexed view** (depending on the database system). Here's how this works for some popular databases:

1. SQL Server (Indexed Views)

In SQL Server, you can create an **indexed view** (which SQL Server calls a "materialized view" under the hood). This means the view's result is physically stored on disk, and you can create indexes on it to improve performance.

Steps to Create an Indexed View in SQL Server:

Create the View: The view must meet certain requirements, such as:

- It must be SCHEMABINDING (i.e., the view is bound to the schema of the base tables, preventing changes to the underlying tables that would invalidate the view).
- All functions used must be deterministic (i.e., they return the same result for the same input).

Example:

CREATE VIEW SalesView

WITH SCHEMABINDING

AS

SELECT StoreID, COUNT_BIG(*) AS SalesCount

FROM dbo.Sales

GROUP BY StoreID;

2. Create the Index on the View:

After creating the view, you can create a **clustered index** on it. This materializes the view and allows further indexing.
Example:

CREATE UNIQUE CLUSTERED INDEX idx_SalesView ON SalesView(StoreID);

After this, SQL Server physically stores the view's data and updates the index as the underlying tables are modified.

23. You have two tables, shop_1 and shop_2 , both having the same structure with customer_id and customer_name. Write a query that retrieves the names of customers who appear in both tables.

To retrieve the names of customers who appear in both **shop_1** and **shop_2** tables, you can use an INNER JOIN or a **INTERSECT** (if supported by your database). Here's how you can do it using both methods:

1. Using INNER JOIN:

This approach joins the two tables based on the **customer_id** (or **customer_name** if you prefer) and retrieves customers who are present in both tables.

SELECT s1.customer_name

FROM shop_1 s1

INNER JOIN shop_2 s2

ON s1.customer_id = s2.customer_id;

- This query joins the **shop_1** table with **shop_2** based on the **customer_id**.
- It returns only those rows where there is a match in both tables.

2. Using INTERSECT:

Some databases like PostgreSQL, Oracle, and SQL Server support the **INTERSECT** operator, which returns only the rows that are common to both queries.

SELECT customer_name

FROM shop_1

INTERSECT

SELECT customer_name

FROM shop_2;

- This query retrieves the **customer_name** that exists in both **shop_1** and **shop_2**.
- Note that **INTERSECT** only returns distinct results by default, so you do not need to use **DISTINCT**

In conclusion :

INNER JOIN works in all SQL databases.

INTERSECT is a simpler and more concise option but might not be available in all databases like MySQL.

Most asked difference between questions in SQL:

24. Difference between INNER JOIN and OUTER JOIN?

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

  SELECT * FROM table1

  INNER JOIN table2

  ON table1.id = table2.id;

  - **Example:** Only retrieves rows that exist in both table1 and table2.

- **OUTER JOIN:** Returns matched rows, plus unmatched rows from either or both tables (depending on type: LEFT, RIGHT, or FULL)**.**
  - **LEFT JOIN:** Returns all rows from the left table and matched rows from the right table. Unmatched rows from the right table are null.
  - **RIGHT JOIN:** Returns all rows from the right table and matched rows from the left table.
  - **FULL OUTER JOIN:** Returns rows when there is a match in either table and unmatched rows from both tables.

  SELECT * FROM table1

  LEFT JOIN table2

  ON table1.id = table2.id;

25. Difference Between WHERE and HAVING

WHERE: Filters rows before aggregation. It is applied to individual rows and cannot work with aggregate functions.

```
SELECT * FROM table1

WHERE condition;
```

HAVING: Filters groups after aggregation. It is used with aggregate functions like **SUM(), COUNT(),** etc., and is applied after the **GROUP BY** clause.

```
SELECT column, COUNT(*)

FROM table1

GROUP BY column

HAVING COUNT(*) > 5;
```

26. Difference Between UNION and UNION ALL?

- **UNION**: Combines results from two or more SELECT queries and removes duplicate rows from the result.

```
SELECT column1

FROM table1

UNION

SELECT column1 FROM table2;
```

- **UNION ALL**: Combines results from two or more SELECT queries but **does not remove duplicates**.

```
SELECT column1

FROM table1

UNION ALL

SELECT column1 FROM table2;
```

27. Difference Between DELETE and TRUNCATE

DELETE: Removes rows from a table based on a condition. It can be rolled back, and each row is deleted one by one. It does not reset auto-increment values.

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

TRUNCATE: Removes all rows from a table without logging individual row deletions. It is faster than DELETE and resets any auto-increment counters. In most databases, it cannot be rolled back.

```
TRUNCATE TABLE table_name;
```

28. Difference Between PRIMARY KEY and UNIQUE

PRIMARY KEY: Uniquely identifies each row in a table. There can only be one primary key in a table, and it cannot have NULL values.

```
CREATE TABLE employees (
```

```
    id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

UNIQUE: Ensures all values in a column or group of columns are unique. Unlike the primary key, a table can have multiple UNIQUE constraints, and it allows one NULL value per column.

```
CREATE TABLE employees (
    email VARCHAR(100) UNIQUE
);
```

29. Difference Between DROP and TRUNCATE?

DROP: Completely removes a table (or other database objects like views or indexes) from the database. All data, structure, and dependencies are removed.

```
DROP TABLE table_name;
```

TRUNCATE: Removes all rows from a table but keeps the table structure intact for future use.

```
TRUNCATE TABLE table_name;
```

30. Difference Between VARCHAR and CHAR?

VARCHAR: Stores variable-length strings. It uses only the required space based on the string length (plus 1-2 bytes for storing length).

```
CREATE TABLE employees (
    name VARCHAR(50)
);
```

CHAR: Stores fixed-length strings. It always uses the defined length, padding the string with spaces if necessary.

```
CREATE TABLE employees (
    code CHAR(10)
);
```

31. Difference Between IN and EXISTS

IN: Checks whether a value exists in a list of values or a subquery. It's generally used when you're dealing with a small list.

```
SELECT *
FROM employees
WHERE id IN (SELECT id FROM managers);
```

EXISTS: Checks if a subquery returns any rows. It's generally more efficient with large datasets because it stops scanning once a match is found.

```
SELECT *
FROM employees
WHERE EXISTS (SELECT 1 FROM managers WHERE employees.id = managers.id);
```

32. Difference Between JOIN and SUBQUERY?

JOIN: Combines rows from two or more tables based on a related column. It's more efficient when you need data from multiple tables in the same result set.

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

SUBQUERY: A query inside another query. It can be used in SELECT, WHERE, or FROM clauses. It's useful when a query depends on the result of another query.

```
SELECT name
FROM employees
WHERE department_id = (SELECT id FROM departments WHERE name = 'HR');
```

SQL Queries Mostly/Commonly Asked by Different Companies:

Most asked Queries in SQL	Asked By
1 . SQL query to find Nth highest salary. SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 2;	Commonly
2 . SQL to write 2nd highest salary in MYSQL . SELECT MAX(Salary) FROM Employee WHERE Salary < (SELECT MAX(Salary) FROM Employee)	Commonly

3 . Find all employees with duplicate names. SELECT name, COUNT(*) FROM employees GROUP BY name HAVING COUNT(*) > 1;	TCS
4 . Find the Second Highest Salary from the table. SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM employees);	Commonly
5 . How to create an empty table with the same structure as another table. SELECT * INTO student_copy FROM students WHERE 1=2;	
6 . Increase the income of all employees by 5% in a table. UPDATE employees SET income = income + (income*5.0/100.0);	i-exceed
7 . Find names of employees starting with "A". SELECT first_name FROM employees WHERE first_name LIKE 'A%';	EY
8 . Find a number of employees working in department 'ABC'. SELECT count(*) FROM employees WHERE deparment_name = 'ABC';	
9 . Print details of employees whose first name ends with 'A' and contains 6 alphabets. SELECT * FROM employees WHERE first-name LIKE '_ _ _ _ _ A';	

10 . Print details of employees whose salary lies between 10000 to 50000. SELECT * FROM employees WHERE salary BETWEEN 10000 AND 50000;	
11 . Fetch duplicate records from the table. SELECT column_name, COUNT(*) AS count FROM table_name GROUP BY column_name HAVING COUNT(*) > 1;	**Capgemini Fresher's interview question**
12 . Fetch Top N Records by Salary. SELECT * FROM employees ORDER BY salary DESC LIMIT N;	
13 . Find All Employees Working Under a Particular Manager. SELECT name FROM employees WHERE manager_id = (SELECT id FROM employees WHERE name = 'ManagerName');	**TCS**
14 . Fetch only the first name from the full-name column. SELECT substring(fullname,1,locate(' ',fullname)) AS FirstName FROM employee;	**Newgen**
15 . Get Employees Hired in the Last 8 Months. SELECT * FROM employees WHERE hire_date >= CURDATE() - INTERVAL 8 MONTH;	**Commonly**
16 . Retrieve the Name of the Employee With the Maximum Salary. SELECT name FROM employees ORDER BY salary DESC LIMIT 1;	

17 . Find employees who have worked for more than one department. SELECT employee_id FROM employees_history GROUP BY employee_id HAVING COUNT(DISTINCT department_id) > 1;	**Cognizant**
18 . Find employees who have worked for more than one department. SELECT employee_id FROM employees_history GROUP BY employee_id HAVING COUNT(DISTINCT department_id) > 1;	**Cognizant**
19 . Write a query using UNION to display employees who have either worked on 'Project A' or 'Project B' but without duplicates. SELECT employee_name FROM project_a UNION SELECT employee_name FROM project_b;	**Capgemini**
20 . Fetch common records between two tables. SELECT * FROM table1 intersect SELECT * FROM table2;	
21 . Create an empty table with the same structure as the other table. SELECT * INTO newtable FROM oldtable WHERE 1=0;	
22. To display users who have placed fewer than 3 orders, let's assume you have two tables Accounts and Orders SELECT a.user_id, a.name, COUNT(o.order_id) AS order_count FROM Accounts a JOIN Orders o ON a.user_id = o.user_id GROUP BY a.user_id, a.name HAVING COUNT(o.order_id) < 3;	**EPAM**

23. Database query i want the employee salary > 15000, here I have two different tables so you have to write a sql query for that. SELECT e.employee_id, e.name, s.salary FROM employees e JOIN salaries s ON e.employee_id = s.employee_id WHERE s.salary > 15000;	**Innova Solutions**