SQL Interview Questions

1. What is SQL? And Purpose of SQL?

Ans. SQL (Structured Query Language) is a programming language specifically designed for managing, querying, and manipulating relational databases. It provides a standardized way to interact with databases to retrieve, insert, update, delete, and manage data. The primary purpose of SQL is to enable users and applications to interact with relational databases in a structured and efficient manner. It serves various purposes: SQL is a versatile tool for managing and analyzing data in relational databases. Its purpose is to enable efficient and standardized data storage, retrieval, and manipulation, making it a cornerstone of modern data management systems.

2. What is a Database?

Ans. A database is an organized collection of data that is stored, managed, and accessed electronically. It is designed to efficiently handle large volumes of data, allowing for easy retrieval, modification, and management. Databases are used in a wide range of applications, from small-scale personal use to large-scale enterprise systems, and can store diverse types of data such as text, numbers, images, and more. Examples of Databases in Real Life - E-commerce Websites, Banking Systems, Healthcare Systems, Social Media Platforms.

A database is the backbone of modern applications, providing an efficient and structured way to store, manage, and access data. It plays a critical role in almost every aspect of technology and data-driven decision-making.

- **Key Components of a Database -**
  - **Data** - The actual information stored, such as customer details, sales records, or product inventories.
  - **Database Management System (DBMS)** - A software tool (e.g., MySQL, Oracle, SQL Server) that provides an interface for users to interact with the database.
  - **Schema** - The structure of the database, defining how data is organized (e.g., tables, columns, relationships).
  - **Query Language** - Tools like SQL that allow users to retrieve or manipulate data.

- **Types of Databases –**
  - **Relational Database -** Organizes data into tables (rows and columns) with relationships between them. Example: MySQL, PostgreSQL, Oracle Database.
  - **NoSQL Database** - Stores unstructured or semi-structured data, such as documents or key-value pairs. Example: MongoDB, Cassandra, Redis.
  - **Cloud Database** - Hosted on cloud platforms for scalability and accessibility. Example: AWS RDS, Google Cloud Firestore.

3. What Are SQL Commands? And Types of SQL Commands?

Ans. SQL commands are instructions used to interact with a database. These commands allow users to perform various tasks, such as creating databases, managing data, retrieving information, and controlling user access.

- **Types of SQL Commands -** SQL commands are broadly categorized into five types based on their functionality
    - **Data Definition Language (DDL)** - Define or modify the structure of the database. Eg. Create, Alter, Drop, Truncate
    - **Data Manipulation Language (DML) -** Manipulate data stored in the database. Eg. Insert, Update, delete
    - **Data Query Language (DQL)** - Retrieve data from the database. Eg. Select
    - **Transaction Control Language (TCL) -** Manage database transactions, ensuring data integrity. Eg. Commit, RollBack, SavePoint, Set Transaction
    - **Data Control Language (DCL) -** Control access to the database. Eg. Grant, Revoke

4. How an SQL Query is Executed?

Ans. When you write and execute an SQL query, it goes through several stages before producing the desired output. These stages involve parsing, optimization, and execution, managed by the Database Management System (DBMS).

- **Step-by-Step Execution of a SELECT Query –**

  FROM & JOIN(Merged) → WHERE(Filtered) → GROUP BY(Grouped) → HAVING(Filtered) →

  SELECT(Selected) → ORDER BY(Ordered) → LIMIT & OFFSET(Limited)

- **Steps (How an SQL Query is Executed) –**
    - **Query Submission** - The user submits an SQL query through an application or directly to the database using a database interface or tool (e.g., SQL Server Management Studio, MySQL Workbench).
        - SELECT Name, Department FROM Employees WHERE Salary > 50000;
    - **Parsing** - The DBMS parses the query to:
        - **Validate Syntax**: Ensures the query follows SQL syntax rules.
        - **Validate Semantics**: Checks if the tables, columns, or functions referenced in the query exist.
        - If there are errors (syntax or semantic), the DBMS throws an error at this stage.
    - **Query Optimization** - The DBMS generates multiple potential execution plans to fetch the data and chooses the most efficient one. Eg. A query with a WHERE clause may use an index if available, reducing the rows scanned.
    - **Query Compilation** - The chosen execution plan is translated into a low-level executable format that the database engine can understand.
    - **Execution Plan Execution** - The database engine executes the query based on the chosen plan.
        - Steps Involved –
            - Data Access – Read data from tables or indexes
            - Joins - Combines data from multiple tables based on specified conditions.
            - Filtering – Applies conditions from the WHERE clause

        - Sorting and Aggregation - Handles operations like ORDER BY, GROUP BY, or aggregate functions.
    - **Data Retrieval** - After processing, the query returns the results to the user or application.

5. What is diff btn char and varchar in sql?

Ans. Both CHAR and VARCHAR are used to store string data in SQL, but they differ in how they handle storage and retrieval.

- **CHAR -** best for fixed-length data, ensures consistent storage but may waste space. Faster for operations due to fixed size. Eg. CHAR(10) allocates exactly 10 bytes.
- **VARCHAR -** best for variable-length data, saves space but involves additional length storage overhead. Slower compared to CHAR for large datasets because of the overhead of variable length. Eg. VARCHAR(10) allocates up to 10 bytes.

6. What is a subquery and it's use case with one example?

Ans. A **subquery** (also known as an **inner query** or **nested query**) is a query within another SQL query. It is enclosed in parentheses and provides data to the main query (also called the outer query). Subqueries can return a single value, a list of values, or a table.

Eg.
```
SELECT employee_name
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

- **Use Cases of Subqueries –**
    - **Filtering Data** - Use a subquery in the WHERE clause to filter results based on values from another query.
    - **Aggregations** - Use a subquery to calculate aggregates (e.g., max, min, average) and use them in the main query.
    - **Updating Data** – Use a subquery to update rows based on conditions from another table or query.
    - **Comparisons** - Use a subquery to compare data between two tables or datasets.
    - **Correlated Subqueries** - A **correlated subquery** is a type of subquery that depends on data from the outer query for its execution. Unlike a regular subquery, which runs independently and provides a result to the outer query, a correlated subquery executes repeatedly, once for each row processed by the outer query.
      Eg. Find employees whose salary is higher than the average salary of their department.
      ```
      SELECT employee_name, department_id, salary
      FROM employees e1
      WHERE salary > (
              SELECT AVG(salary)
              FROM employees e2
              WHERE e1.department_id = e2.department_id );
      ```

7. Difference Between DELETE, TRUNCATE, and DROP Commands in SQL?

Command	Scope	Impact	Rollback	Performance	Usage
DELETE	Removes specific rows based on a WHERE clause	Removes data, keeps table structure and constraints	Yes, within a transaction	Slower	Removing specific rows from a table.
TRUNCATE	Removes all rows in a table	Removes all data, keeps table structure and constraints	May or may not be recoverable	Faster	Removing all rows in a table quickly.
DROP	Removes an entire table or database object	Removes data, structure, constraints, and indexes	No	Fastest	Removing a table or database object entirely.

8. Describe the difference between WHERE and HAVING in SQL?

Aspect	WHERE	HAVING
Purpose	Filters rows before any grouping or aggregation occurs.	Filters groups or aggregated data after the GROUP BY clause is applied.
Scope	Works on individual rows in the table.	Works on aggregated data or groups created by GROUP BY.
Aggregation	Cannot use aggregate functions (e.g., SUM(), COUNT()) directly.	Can use aggregate functions to filter based on computed values.
Execution Order	Executed before GROUP BY.	Executed after GROUP BY.
Syntax	SELECT ... FROM table GROUP BY column HAVING condition;	SELECT ... FROM table GROUP BY column HAVING condition;

9. what are the operators in SQL?

Ans. In SQL, operators are used to perform operations on values and columns to return results or modify data. These below operators provide flexibility to filter, manipulate, and transform data according to various conditions in SQL queries.

Category	Operators
Arithmetic	+, -, *, /, %
Comparison	=, != / <>, >, <, >=, <=
Logical	AND, OR, NOT
String	LIKE, NOT LIKE, CONCAT(), `
NULL	IS NULL, IS NOT NULL
Set	IN, NOT IN, BETWEEN, NOT BETWEEN
Subquery	EXISTS, ALL, ANY
Aggregate Functions	COUNT(), SUM(), AVG(), MIN(), MAX()
Conditional	CASE, IF, IIF (If language support)

10. What is key in SQL and diff keys in SQL. Explain in detail with example?

Ans. In SQL, a **key** is a field (or a combination of fields) in a table that is used to uniquely identify records in that table or to establish relationships between different tables. Keys are essential for ensuring data integrity and enabling efficient querying.

- **Different Types of Keys in SQL – Super Key ->** Each key type plays a crucial role in maintaining data integrity, preventing duplication, and ensuring that relationships between tables are consistent.
    - **Primary Key** - A **Primary Key** is a column (or set of columns) that uniquely identifies each record in a table. A primary key cannot contain NULL values and must have unique values for each row in the table. **Eg**. EmployeeID INT PRIMARY KEY
    - **Foreign Key** - A **Foreign Key** is a column (or combination of columns) in one table that is used to link to the **Primary Key** or **Unique Key** of another table. It ensures referential integrity between two tables. **Eg**. FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
    - **Unique Key** - A **Unique Key** ensures that all values in a column (or a combination of columns) are unique across the table. Unlike the **Primary Key**, a **Unique Key** can accept NULL values, but only one NULL is allowed per column. **Eg**. Username VARCHAR(50) UNIQUE
    - **Composite Key** - A **Composite Key** is a combination of two or more columns in a table that uniquely identifies a record. Each column in a composite key can contain duplicates, but the combination of the columns must be unique. **Eg**. PRIMARY KEY (StudentID, CourseID)
    - **Candidate Key** - A **Candidate Key** is a column, or a combination of columns, that can uniquely identify records in a table. The difference between a **Candidate Key** and a **Primary Key** is that there can be multiple **Candidate Keys**, but only one is chosen to be the **Primary Key**.
      Eg. Both EmployeeID, Email, and PhoneNumber could serve as **Candidate Keys** because they can uniquely identify an employee.
    - **Alternate Key** - n **Alternate Key** is any candidate key that is not chosen as the **Primary Key**. These keys are still unique and can be used to enforce data integrity.
      Eg. If we have a table where Email and PhoneNumber are candidate keys but only EmployeeID is chosen as the **Primary Key**, then Email and PhoneNumber are **Alternate Keys**.
    - **Candidate Key** - A **Candidate Key** is a column, or a combination of columns, that can uniquely identify records in a table. The difference between a **Candidate Key** and a **Primary Key** is that there can be multiple **Candidate Keys**, but only one is chosen to be the **Primary Key**.
      Eg. Both EmployeeID, Email, and PhoneNumber could serve as **Candidate Keys** because they can uniquely identify an employee.

11. What is a subquery and it's use case with one example?

Ans. A **subquery** (also known as an **inner query** or **nested query**) is a query within another SQL query. It is enclosed in parentheses and provides data to the main query (also called the outer query). Subqueries can return a single value, a list of values, or a table.
Eg.

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

- **Use Cases of Subqueries –**
  - **Filtering Data** - Use a subquery in the WHERE clause to filter results based on values from another query.
  - **Aggregations** - Use a subquery to calculate aggregates (e.g., max, min, average) and use them in the main query.
  - **Updating Data** – Use a subquery to update rows based on conditions from another table or query.
  - **Comparisons** - Use a subquery to compare data between two tables or datasets.
  - **Correlated Subqueries** - A **correlated subquery** is a type of subquery that depends on data from the outer query for its execution. Unlike a regular subquery, which runs independently and provides a result to the outer query, a correlated subquery executes repeatedly, once for each row processed by the outer query.
    Eg. Find employees whose salary is higher than the average salary of their department.

```
SELECT employee_name, department_id, salary
FROM employees e1
WHERE salary > (
        SELECT AVG(salary)
        FROM employees e2
        WHERE e1.department_id = e2.department_id
);
```

12. What SQL constraints. Explain in detail with example?

Ans. SQL constraints are rules that are applied to columns or tables to ensure the accuracy, integrity, and reliability of the data in a database. Constraints are used to define restrictions on the data that can be entered into a table. They help maintain data quality and prevent incorrect or inconsistent data from being stored.

- **Types of SQL Constraints**
  - **NOT NULL -** The NOT NULL constraint ensures that a column cannot have a NULL value. It forces the user to provide a value for the column when inserting or updating a record.
    Eg. EmployeeID INT NOT NULL

- **UNIQUE -** The UNIQUE constraint ensures that all values in a column (or a combination of columns) are unique across the table. It allows NULL values unless explicitly defined otherwise.
  Eg. Email VARCHAR(100) UNIQUE
- **PRIMARY KEY -** The PRIMARY KEY constraint uniquely identifies each record in a table. It combines both the NOT NULL and UNIQUE constraints. A table can have only one primary key, and it cannot accept NULL values.
  Eg. EmployeeID INT PRIMARY KEY
- **FOREIGN KEY -** The FOREIGN KEY constraint is used to enforce a link between two tables. It ensures that the value in a column (or combination of columns) in one table must match a value in the referenced table's primary or unique key column.
  Eg. FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
- **CHECK -** The CHECK constraint ensures that all values in a column satisfy a specified condition. It can be used to enforce rules on data values (such as age being greater than a certain number).
  Eg. Age INT, CHECK (Age >= 18)
- **DEFAULT -** The DEFAULT constraint is used to set a default value for a column when no value is provided during an insert operation. This ensures that a column always has a value, even if it's not explicitly specified.
  Eg. Status VARCHAR(20) DEFAULT 'Active'
- **INDEX –** The INDEX constraint is used to create an index on one or more columns in a table. Indexes are used to speed up data retrieval operations by allowing faster searches.
  Eg. CREATE INDEX idx_employee_name ON Employees (Name);

13. what is function in SQL and its different types with example?

Ans. In SQL, a function is a set of SQL statements that perform a specific task, return a result, and can be invoked in a query. Functions are used to encapsulate reusable logic that can be applied to data within queries. They allow you to perform operations such as mathematical calculations, string manipulation, or data transformation.

- **Types of Functions in SQL –**
  - **Scalar Functions -** These functions operate on a single value and return a single result. They are often used in SELECT, WHERE, or HAVING clauses.
    Example - String Functions, Numeric Functions, Date Functions, Conversion Functions
  - **Aggregate Functions -** These functions perform a calculation on a set of values and return a single result. They are often used with GROUP BY to aggregate data at different levels.
    Example - SUM(), AVG(), MIN(), MAX(), COUNT()
  - **Window Functions (Analytical Functions) -** Window functions perform calculations across a set of table rows that are related to the current row. They do not collapse the result set, unlike aggregate functions.

Example - NTILE(), RANK(), ROW_NUMBER()

- **System Functions -** These functions provide information about the database environment or session, such as system variables, user info, or current timestamp.
  Example - CURRENT_TIME(), CURRENT_DATE(), DATABASE(), USER()
- **Conversion Functions -** These functions are used to convert one data type to another.
  Example - CONVERT(), CAST()
- **User-Defined Functions (UDFs) -** These are custom functions created by users to extend the functionality of SQL. UDFs allow the user to encapsulate reusable logic into functions that can be invoked like built-in SQL functions**.**
  Example -

```
CREATE FUNCTION GetEmployeeSalary (emp_id INT)
        RETURNS DECIMAL(10, 2)
AS
BEGIN
        DECLARE @salary DECIMAL(10, 2);
SELECT @salary = salary FROM employees WHERE employee_id = emp_id;
RETURN @salary; END;
```

- **USING Clause -** The **USING** clause simplifies the join condition when the columns in both tables have the **same name**. It only allows equality checks and is used to specify one or more columns to match on without having to repeat the column names in both tables. The **USING** clause is typically used for **equality joins** where the columns have identical names.

14. What is the CASE() function and it's alternative if any?

Ans. The **CASE()** function in SQL is used to perform conditional logic within a query. It allows you to return specific values based on conditions, similar to an IF-THEN-ELSE statement in programming. The CASE function is typically used to: Replace conditional expressions with specific value, Modify output based on certain conditions, Create new columns or modify existing columns based on conditions.

- **Simple CASE() Example –**

SELECT column1, column2,
CASE column1
WHEN value1 THEN result1
WHEN value2 THEN result2
ELSE default_result
END AS result_column
FROM table_name;
	SELECT employee_name, salary,
	CASE
	WHEN salary < 15000 THEN 'Low Salary'
	WHEN salary BETWEEN 15000 AND 25000
	THEN 'Medium Salary'
	WHEN salary > 25000 THEN 'High Salary'
	ELSE 'Unknown Salary'
	END AS salary_category
	FROM employees;

- **Searched CASE() Example –**

SELECT column1, column2, CASE WHEN condition1 THEN result1	SELECT employee_name, salary, CASE WHEN salary < 15000 THEN 'Low Salary'

WHEN condition2 THEN result2
ELSE default_result
END AS result_column
FROM table_name;
	WHEN salary BETWEEN 15000 AND 25000
	THEN 'Medium Salary'
	WHEN salary > 25000 THEN 'High Salary'
	ELSE 'Unknown Salary'
	END AS salary_category
	FROM employees;

- **Alternative to the CASE() Function –** While the CASE function is the most common and versatile way to handle conditional logic in SQL, there are alternatives depending on the database system and scenario.

IF() Function (MySQL)	IIF() Function (SQL Server, MS Access)
In MySQL, you can use the IF() function as a more concise alternative to CASE. The IF() function takes three parameters: a condition, a result if the condition is true, and a result if the condition is false.	SQL Server and MS Access support the IIF() function, which works similarly to the IF() function in MySQL. It is often used for simpler conditions.

SELECT employee_name, salary,
IF(salary < 15000, 'Low Salary', IF(salary BETWEEN 15000 AND 25000, 'Medium Salary', 'High Salary')) AS salary_category
FROM employees;
	SELECT employee_name, salary,
	IIF(salary < 15000, 'Low Salary', IIF(salary BETWEEN 15000 AND 25000, 'Medium Salary', 'High Salary')) AS salary_category
	FROM employees;

15. What is a join? What types of joins do you know? explain in detail with example?

Ans. A **SQL JOIN** is a SQL operation that combines columns from two or more tables based on a related column between them. Joins are used to retrieve data from multiple tables based on a condition, usually involving a foreign key that references a primary key in another table. Joins are a powerful feature in SQL that allow you to combine data from different tables in a way that is logical and meaningful. The most common type of relationship between tables is the **one-to-many** or **many-to-many** relationships.

Semi-Join and Anti-Join – Advanced Join in SQL

- **INNER JOIN -** The INNER JOIN returns only the rows that have matching values in both tables. If there is no match between the two tables, the row will not be included in the result set.
  Eg. You have an Employees table and a Departments table. You only want to list employees who are assigned to a department (ignoring employees without departments).
- **LEFT JOIN (or LEFT OUTER JOIN) -** The LEFT JOIN returns all rows from the left table and the matched rows from the right table. If there is no match, the result will still include all rows from the left table with NULL values for columns from the right table.
  Eg. You want to find all employees and their respective departments, including employees who are not assigned to any department.
- **RIGHT JOIN (or RIGHT OUTER JOIN) -** The RIGHT JOIN returns all rows from the right table and the matched rows from the left table. If there is no match, the result will include all rows from the right table with NULL values for columns from the left table.
  Eg. You want to find all departments, including departments that have no employees assigned.

- **FULL JOIN (or FULL OUTER JOIN) -** The FULL JOIN returns all rows when there is a match in one of the tables. If there is no match, the result will still include the rows from both tables, with NULL values where there is no match.
  Eg. You want to list all employees and all departments, including employees with no department and departments with no employees.
- **CROSS JOIN -** The CROSS JOIN returns the Cartesian product of the two tables. This means it returns every combination of rows from both tables.
  Eg. You want to create all possible combinations of employees and projects, even if an employee hasn't been assigned a project.
- **SELF JOIN -** A SELF JOIN is a join where a table is joined with itself. It can be useful for querying hierarchical data or when you need to compare rows within the same table.
  Eg. You want to list all employees and their managers. The Employees table contains a ManagerID column that refers to the EmployeeID of the manager.

16. What is a NATURAL JOIN?

Ans. A **NATURAL JOIN** is a type of join in SQL that automatically joins two tables based on **columns with the same name** in both tables. It eliminates the need to explicitly specify the columns in the ON clause. The **NATURAL JOIN** will compare all columns with the same name in both tables and create a result set where rows with equal values in those columns are combined.

- **Key Points about NATURAL JOIN** –
  - **Automatic Matching**: It automatically matches columns between the tables with the same name and combines them, without explicitly specifying them in a condition.
  - **Equality Condition**: The join is performed using the **equality** operator (=) for columns with the same name.
  - **Eliminates Duplicates**: The columns that are common between the tables will appear only once in the result set.
  - **Column Names Must Match**: The columns must have the **same name** in both tables for the NATURAL JOIN to work.
- **Drawbacks of NATURAL JOIN –**
  - **Ambiguity in column matching**: If two tables have multiple columns with the same name, **NATURAL JOIN** will match all of them, which might not always be the intended behavior. This can lead to unexpected results if the column names are not consistent.
  - **Less control**: Because the join condition is automatic, you have less control over which columns are matched, and may unintentionally join columns that shouldn't be.

17. Difference Between GROUP BY and PARTITION BY in SQL?

Aspect	GROUP BY	PARTITION BY
Purpose	Groups rows into summary rows and applies aggregate functions to each group.	Divides the result set into partitions for window functions without collapsing rows.
Result Set	Collapses the result set to one row per group.	Retains the original rows in the result set while calculating aggregate functions.

Aggregation	Typically used with aggregate functions (e.g., SUM(), COUNT(), etc.) to summarize data.	Used with window functions (e.g., ROW_NUMBER(), RANK(), SUM(), etc.) to compute values across partitions.
Example Use Case	Finding the sum or average of a column per group.	Calculating running totals, ranks, or moving averages while retaining individual rows.
Grouping	Groups data by one or more columns, and each group produces a single row in the result.	Partitions data but does not reduce the number of rows—each row in the result belongs to a partition.
Functions	Works only with aggregate functions.	Works with both aggregate and window functions.

18. Difference between CROSS JOIN and other types of SQL joins?

Aspect	CROSS JOIN	Other Joins (INNER, LEFT, RIGHT, FULL)
Purpose	Returns the Cartesian product of two tables (every combination of rows).	Returns matching rows based on a condition (ON clause).
Condition	No condition (no ON clause).	Requires a condition (ON clause) to match rows between tables.
Result Set Size	Result size = Number of rows in table 1 * Number of rows in table 2.	Result size depends on the type of join (inner match, or full match).
Use Case	Generating combinations of rows, matrix-like queries.	Relational data, filtering records based on relationships between tables.
Example Scenario	Generating a combination of all products and stores.	Listing employees with their assigned departments, or all orders with product details.
Performance	Can be very expensive if tables are large.	Usually more optimized for relational data and specific matching.

19. What is the difference between the ON and USING clauses in a join?

Ans. The **ON** and **USING** clauses in SQL are both used to specify conditions for how tables are related when performing a join. However, there are important differences in how they are used and what they can express.

- **ON Clause -** The **ON** clause is used to specify the condition or criteria for the join, including the exact columns from each table that should be matched. Use the **ON** clause when you need to join on columns with different names, or if you have multiple conditions to satisfy. You can also use it when you need to apply more complex expressions beyond equality or when there are multiple conditions for the join.
- **USING Clause -** The **USING** clause simplifies the join condition when the columns in both tables have the **same name**. It only allows equality checks and is used to specify one or more columns to match on without having to repeat the column names in both tables. The **USING** clause is typically used for **equality joins** where the columns have identical names.

20. Difference Between the JOIN Clause and the UNION Clause?

Feature	JOIN	UNION
Purpose	Combines columns from multiple tables	Combines rows from multiple queries or tables
Operation	Horizontal combination (side-by-side)	Vertical combination (stacking rows)
Matching	Requires a condition to match rows between tables	Does not require matching columns, but they must have the same structure
Types	INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, etc.	UNION, UNION ALL
Result	Multiple columns from different tables	Rows stacked with the same columns
Performance	Can be slow with large datasets due to matching rows	Can be slower due to duplicate elimination (unless UNION ALL)
Use Case	When you need data from multiple related tables	When you need to combine data from queries with the same structure

21. Difference Between INTERSECT Clause and INNER JOIN?

Aspect	INTERSECT Clause	INNER JOIN
Purpose	Finds common rows between two SELECT statements.	Combines rows from two or more tables based on a matching condition.
Scope	Works at the result set level.	Works at the table level using a join condition.
Syntax	SELECT ... FROM table1 INTERSECT SELECT ... FROM table2;	SELECT ... FROM table1 INNER JOIN table2 ON condition;
Condition	No explicit condition; implicitly compares all selected columns for equality.	Requires an explicit join condition (e.g., matching columns or expressions).
Result	Returns rows that are identical in both SELECT statements.	Combines matching rows from both tables, allowing for column selection.
Duplicate Handling	Removes duplicates by default (unless ALL is specified).	May include duplicates depending on the join condition and data in the tables.
Performance	Can be slower due to set operations (sorting and deduplication).	Generally faster with proper indexing, as it evaluates join conditions directly.
Use Cases	To find common rows between two result sets (e.g., common employees in two groups).	To combine related data from two or more tables (e.g., orders with customer info).
Column Comparison	Compares all selected columns automatically.	Allows you to specify which columns to compare for joining.
Example (Finding common rows)	SELECT emp_id FROM dept1 INTERSECT SELECT emp_id FROM dept2;	SELECT d1.emp_id FROM dept1 d1 INNER JOIN dept2 d2 ON d1.emp_id = d2.emp_id;

22. Difference between count(*), count(1), count(column_name) and if any explain me in detail with example?

Ans. In SQL, the COUNT() function is used to count rows based on certain conditions or criteria. The key difference between COUNT(*), COUNT(1), and COUNT(column_name) lies in what they count and how they behave in certain situations.

- **COUNT(*) -** COUNT(*) counts the total number of rows in a table, including rows with NULL values in any column. This is the most general form of the COUNT() function. It counts every row, regardless of whether a column contains NULL values or not.
- **COUNT(1) -** COUNT(1) is very similar to COUNT(*). It counts all rows in the table, including those with NULL values. The number 1 is just a constant, so it doesn't affect the result. t is often used as an optimization trick, but in practice, there is no significant difference between COUNT(*) and COUNT(1) in most databases because both count all rows.
- **COUNT(column_name) -** COUNT(column_name) counts only the rows where the specified column has non-NULL values. It ignores rows where the column has a NULL value. This version of COUNT() ignores NULL values in the specified column and only counts the rows where the value of that column is not NULL.

23. What is an Index in Database and Types of Indexes?

Ans. An **index** in SQL is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and decreased performance on data modification operations (INSERT, UPDATE, DELETE). Think of an index like the index of a book: instead of searching through the entire book, you go to the specific page number mentioned in the index. Similarly, in SQL, an index helps speed up queries by providing a quick lookup to rows in a table.

In simple terms, an index is created on columns of a table to enable fast searching and retrieval of data. When you create an index on one or more columns, it helps SQL queries that use these columns in search conditions (WHERE clause) to execute more quickly.

- **Types of Indexes in SQL –**
  - **Single-Column Index** - This is the simplest form of index, where an index is created on a single column.
    Eg. CREATE INDEX idx_age ON students(age);
  - **Multi-Column (Composite) Index** - A composite index is created on multiple columns of a table. This is useful when queries filter on more than one column.
    Eg. CREATE INDEX idx_name_age ON students(last_name, age);
  - **Unique Index** - A unique index ensures that the values in the indexed column(s) are unique. This type of index is automatically created when you define a primary key or unique constraint.
    Eg. CREATE UNIQUE INDEX idx_unique_student_id ON students(student_id);
  - **Clustered Index** - A clustered index determines the physical order of data in the table. In most database systems, the primary key automatically creates a clustered index. A table can only have one clustered index because the data rows themselves can only be sorted in one order.

Eg. CREATE CLUSTERED INDEX idx_clustered_student_id ON students(student_id);

- Non-Clustered Index - A non-clustered index is an index structure separate from the table's data. It stores the index key values and a reference to the data rows where the actual data is stored. A table can have multiple non-clustered indexes.
  Eg. CREATE NONCLUSTERED INDEX idx_nonclustered_name ON students(first_name);

Indexes are crucial for performance, especially when working with large tables and complex queries. They are especially helpful in speeding up SELECT queries but may degrade performance on INSERT, UPDATE, and DELETE operations due to the overhead of maintaining the index. The type of index chosen depends on the specific use case and the structure of the data and queries.

24. What are the Window Functions and their types. Explain detail with one example each?

Ans. Window functions are a feature in SQL and data processing frameworks like PySpark that allow users to perform calculations across a set of table rows that are related to the current row. These functions are often used to calculate running totals, ranks, averages, and more, without requiring the aggregation or grouping of data.

- **Types of Window Functions –**
  - **Aggregate Functions** - These apply aggregate operations (e.g., SUM, AVG) over a defined window of rows.
    Example: Calculate the running total of sales for each employee.

```
SELECT
   EmployeeID,
   SaleAmount,
   SUM(SaleAmount) OVER (PARTITION BY EmployeeID ORDER BY SaleDate) AS RunningTotal
FROM Sales;
```

Function	Description
SUM()	Calculates the sum of values in the window.
AVG()	Calculates the average of values in the window.
MIN()	Returns the smallest value in the window.
MAX()	Returns the largest value in the window.
COUNT()	Counts the number of rows in the window.

  - **Ranking Functions** - These assign ranks to rows based on the order specified in the query.
    Example: Assign a rank to employees based on their sales performance.

```
SELECT
   EmployeeID,
   SaleAmount,
   RANK() OVER (PARTITION BY DepartmentID ORDER BY SUM(SaleAmount) DESC) AS
Rank
FROM Sales
GROUP BY EmployeeID, DepartmentID, SaleAmount;
```

Function	Description

RANK()	Assigns a rank, with ties receiving the same rank; skips ranks after ties.
DENSE_RANK()	Similar to RANK(), but does not skip ranks after ties.
ROW_NUMBER()	Assigns a unique number to each row starting from 1.
NTILE(n)	Divides rows into n equal parts and assigns a bucket number to each row.

- **Value Functions** - These return specific rows or values relative to the current row.
  Example: Get the previous or next order date for each customer.
  ```
  SELECT
     CustomerID,
     OrderDate,
     LAG(OrderDate) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS
  PreviousOrderDate
  FROM Orders;
  ```

Function	Description
LAG(column, offset, default)	Fetches the value of a column from a preceding row.
LEAD(column, offset, default)	Fetches the value of a column from a following row.
FIRST_VALUE(column)	Returns the first value in the window.
LAST_VALUE(column)	Returns the last value in the window.
NTH_VALUE(column, n)	Returns the nth value in the window.

25. What is a View in SQL?

Ans. A **view** in SQL is essentially a virtual table created by a query that combines data from one or more tables. It does not store data itself but instead dynamically retrieves data from the underlying tables when queried. Views are useful for simplifying complex queries, encapsulating business logic, and improving security by restricting direct access to certain columns or rows of data.

- **Key Features of Views –**
  - **Virtual Table:** A view behaves like a table but doesn't actually store data. It generates results from the underlying base tables when queried.
  - **Simplifies Complex Queries:** By creating a view, complex queries can be saved and reused, reducing redundancy in code.
  - **Security:** Views can provide a way to restrict access to sensitive data by only exposing necessary columns and rows.
  - **Data Consistency:** Views can help ensure consistency in query results by encapsulating commonly used joins or filters.

- **Types of Views in SQL –**
  - **Simple Views -** A simple view is based on a single table and does not contain any complex joins or subqueries. Supports SELECT, INSERT, UPDATE, and DELETE operations on the underlying table (depending on the database system).
    Eg. CREATE VIEW employee_names AS
    SELECT employee_id, first_name, last_name FROM Employees;

- **Complex Views -** A complex view is based on multiple tables and may contain joins, subqueries, and groupings. May contain aggregate functions, subqueries, or complex joins. May not always support operations like UPDATE, DELETE, or INSERT on the view itself because it may not map directly to a single underlying table.
  Eg. CREATE VIEW employee_department AS
  SELECT e.first_name, e.last_name, d.department_name
  FROM Employees e
  JOIN Departments d ON e.department_id = d.department_id;
- **Materialized Views -** A materialized view is like a regular view but it stores the results of the query physically in the database. This allows for faster access to the data, especially for complex queries. Stores the result of the query physically and requires periodic refresh to stay up-to-date with the underlying data. Unlike regular views, materialized views offer performance benefits for repeated queries.
  Eg. CREATE MATERIALIZED VIEW employee_sales_summary AS
  SELECT employee_id, SUM(sales_amount) AS total_sales
  FROM Sales
  GROUP BY employee_id;
- **Updatable Views -** These views allow updates, inserts, or deletes to be performed on the view itself, which are then propagated to the underlying base tables. Can be updated directly if the view is simple enough (no joins or aggregate functions).
  Eg. CREATE VIEW active_employees AS
  SELECT employee_id, first_name, last_name
  FROM Employees
  WHERE status = 'active';

  UPDATE view_name
  SET column_name = new_value
  WHERE condition;

26. what is Common Table Expressions (CTEs)? Explain in detail with all it's types and examples?

Ans. A **Common Table Expression (CTE)** is a temporary result set that is defined within the execution scope of a SELECT, INSERT, UPDATE, or DELETE statement. CTEs help improve the readability and organization of complex queries by allowing you to break them into simpler, modular components. CTEs are defined using the WITH keyword and can be used to reference complex subqueries, recursive queries, and simplify joins or other complex operations.

- **Readability**: CTEs can make complex queries easier to read and maintain.
- **Reusability**: You can refer to the CTE multiple times in the same query.
- **Recursion**: CTEs support recursion, which is useful for hierarchical or tree-structured data.
  Example -
  ```
  WITH cte_name AS (
          -- SQL query here
  ) -- Main query that uses the CTE -> SELECT * FROM cte_name;
  ```

Non-Recursive CTE	Recursive CTE
A **Non-Recursive CTE** is a simple CTE that contains a single SELECT statement, and its results are used in the outer query.	A **Recursive CTE** refers to itself. It's useful for hierarchical data (e.g., organizational charts or tree structures). Recursive CTEs have two parts: 1. **Base query**: The initial query that does not reference the CTE. 2. **Recursive query**: The query that refers to the CTE itself.

WITH cte_name AS (
  SELECT column1, column2
  FROM table_name
  WHERE condition
)
SELECT *
FROM cte_name;
	WITH RECURSIVE cte_name AS (
	  -- Base case
	  SELECT column1, column2
	  FROM table_name
	  WHERE condition
	
	  UNION ALL
	
	  -- Recursive part
	  SELECT column1, column2
	  FROM table_name
	  JOIN cte_name
	  ON table_name.column = cte_name.column
	)
	SELECT * FROM cte_name;

WITH DepartmentEmployees AS (
  SELECT EmployeeID, FirstName, LastName
  FROM Employees
  WHERE Department = 'HR'
)
SELECT * FROM DepartmentEmployees;
	WITH RECURSIVE EmployeeHierarchyCTE AS (
	  -- Base case: Find top-level managers (no manager)
	  SELECT EmployeeID, EmployeeName, ManagerID
	  FROM EmployeeHierarchy
	  WHERE ManagerID IS NULL
	
	  UNION ALL
	
	  -- Recursive case: Find employees managed by the previous level's employees
	  SELECT e.EmployeeID, e.EmployeeName, e.ManagerID
	  FROM EmployeeHierarchy e
	  JOIN EmployeeHierarchyCTE eh
	  ON e.ManagerID = eh.EmployeeID
	)
	SELECT * FROM EmployeeHierarchyCTE;

27. Difference between CTE and View? With example

Aspect	CTE (Common Table Expression)	View
Definition	Temporary result set within a single query.	Permanent database object stored for reuse.
Persistence	Exists only for the duration of the query.	Stored permanently in the database.
Syntax	Defined using the WITH clause.	Defined using the CREATE VIEW statement.
Reusability	Cannot be reused in multiple queries.	Can be reused across multiple queries.
Performance	Recalculated for each reference within the query.	May be optimized by the database engine.
Data Modification	Cannot directly modify data.	Some views are updatable, depending on their structure.
Use Case	Ideal for temporary query organization or recursion.	Ideal for reusable, abstracted queries.

Flexibility	Great for complex queries within a single execution context.	Ideal for frequently used, complex joins or aggregations.

28. what is stored procedure? explain in detail with example?

Ans. A **stored procedure** is a precompiled set of one or more SQL statements that are stored in a database and executed as a single unit. Stored procedures are used to encapsulate logic, reduce code redundancy, improve performance, and enhance security in database systems.

Example -

```
CREATE PROCEDURE procedure_name

        @parameter_name datatype [IN|OUT|INOUT],

...

AS

BEGIN

                -- SQL statements

END;
```

- **Key Features of Stored Procedures –**
    - **Encapsulation**: Combines multiple SQL statements into a single reusable unit.
    - **Reusability**: Can be executed multiple times with different input parameters
    - **Performance**: Precompiled and optimized by the database engine, leading to faster execution.
    - **Security**: Restricts direct access to data; users only need execution privileges.
    - **Maintainability**: Changes made to the procedure are automatically reflected wherever it is used.
- **Disadvantages of Stored Procedures –**
    - **Debugging Difficulty**: Harder to debug compared to application code.
    - **Maintenance Overhead**: Changes require redeployment of the procedure.
    - **Portability Issues**: Procedures are database-specific and may not be compatible across systems.

29. what is recursive stored procedure? What is purpose of recursive stored procedure?

Ans. A **recursive stored procedure** is a stored procedure in SQL that calls itself either directly or indirectly in its execution. It is a useful technique for solving problems that involve repeated or nested operations, such as traversing hierarchical data structures like organizational charts or bill-of-materials (BOM) hierarchies.

- **How the Recursive Stored Procedure Works –**
    - A **base case**: The condition that stops the recursion. This is the simplest form of the problem that can be solved without further recursive calls.
    - A **recursive case**: The condition where the stored procedure calls itself, progressively solving smaller parts of the problem until the base case is reached.

Example -

```
CREATE PROCEDURE CalculateFactorial (IN n INT, OUT result INT)
BEGIN
    -- Base case
    IF n = 0 THEN
        SET result = 1;
    ELSE
        -- Recursive case
        CALL CalculateFactorial(n - 1, result);
        SET result = result * n;
    END IF;
END;
```

- **Purpose of Recursive Stored Procedure –**
    - **Handling Hierarchical Data:** Recursive stored procedures are often used to work with data that is structured hierarchically, such as organizational charts (employees reporting to managers), file systems (folders containing files and subfolders), or tree-like structures in databases.
    - **Problem-Solving Efficiency:** Recursive stored procedures provide a more natural and efficient way to implement solutions for problems that involve repeating or nested operations, like computing factorials, Fibonacci series, or traversing multi-level data.
    - **Complex Calculations:** Recursive procedures help in performing complex calculations that require repetitive steps, such as summing up values at multiple levels of a hierarchy.
    - **Data Traversal:** Recursive stored procedures are commonly used for traversing and processing data that exists in tree-like structures (e.g., a table representing categories and subcategories of products).
- **Limitations and Considerations –**
    - **Performance:** Recursive stored procedures can become inefficient if not designed correctly, especially if the recursion depth is too large.
    - **Maximum Recursion Limit:** Some databases impose a maximum recursion depth to prevent infinite recursion from causing stack overflows or consuming too much memory.
    - **Termination:** It is important to define a base case in recursive stored procedures to ensure that the recursion terminates.

30. What is Trigger? Explain in detail with examples

Ans. A **trigger** is a special type of stored procedure in a database that automatically executes or "fires" when certain events occur on a specified table or view. Triggers are often used to enforce business rules, automate tasks, or maintain referential integrity in the database without needing to explicitly call them. Triggers can be defined to execute after (AFTER), before (BEFORE), or instead of an event (INSTEAD OF), and they are typically used in situations like: **Auditing** changes to data, **Enforcing data integrity** (e.g., preventing invalid data), **Automatically updating other tables** based on changes in one table.

- **Types of Triggers –**
  - **BEFORE Trigger -** A BEFORE trigger executes before an INSERT, UPDATE, or DELETE statement is applied to the table. It can be used to modify data before the database action takes place.
    Example -
    ```
    CREATE TRIGGER trigger_name
    BEFORE INSERT/UPDATE/DELETE ON table_name
    FOR EACH ROW
        BEGIN
            -- Trigger logic here

        END;
    ```
  - **AFTER Trigger -** An AFTER trigger executes after an INSERT, UPDATE, or DELETE operation is completed. It's often used to perform additional actions (such as logging changes) after the modification.
    Example -
    ```
    CREATE TRIGGER trigger_name
    AFTER INSERT/UPDATE/DELETE ON table_name
    FOR EACH ROW
        BEGIN
            -- Trigger logic here

        END;
    ```
  - **Row-Level Trigger -** A **row-level trigger** is executed once for each row affected by the triggering event. These triggers use the NEW and OLD keywords to refer to the new or old values of the row being modified.
  - **Example -**
    ```
    CREATE TRIGGER update_salary
    AFTER UPDATE ON Employees
    FOR EACH ROW
    BEGIN
        IF OLD.salary != NEW.salary THEN
        INSERT INTO SalaryHistory (EmployeeID, OldSalary, NewSalary,
        ChangeDate)
        VALUES (NEW.EmployeeID, OLD.salary, NEW.salary, NOW());
        END IF;

    END;
    ```
  - **Statement-Level Trigger -** A statement-level trigger is executed once per SQL statement, regardless of how many rows are affected. It doesn't refer to individual rows, but instead operates on the entire statement.
    Example -
    ```
    CREATE TRIGGER log_salary_update
    ```

```
AFTER UPDATE ON Employees
BEGIN
    INSERT INTO ActionLog (ActionType, ActionDate)
    VALUES ('Salary Update', NOW()); END;
```

- **Read About -** Benefits of Using Triggers, Drawbacks of Using Triggers, When to Use Triggers

31. Explain Normalization and Denormalization in detail with example?

Ans. Normalization and denormalization are processes used in database design to organize data effectively and efficiently. These processes help to balance data integrity, reduce redundancy, and optimize query performance, but they approach data storage from different perspectives.

- **Normalization -** Normalization is the process of designing a relational database in a way that reduces redundancy and ensures data integrity. The primary goal is to organize data so that it adheres to a set of rules or "normal forms" to eliminate undesirable characteristics such as **data redundancy** (duplicate data) and **anomalies** (update, insert, and delete anomalies). Normalization is typically performed in several steps, with each step corresponding to a **normal form (NF)**. There are multiple normal forms (1NF, 2NF, 3NF, BCNF, etc.), each with its own set of rules. A database is considered normalized when it is in one of the higher normal forms.
    - **Key Normal Forms –**
        - 1NF (First Normal Form): Ensures that each column contains atomic (indivisible) values, and there are no repeating groups or arrays.
        - 2NF (Second Normal Form): Ensures that the table is in 1NF and all non-key attributes are fully functionally dependent on the primary key (no partial dependency).
        - 3NF (Third Normal Form): Ensures that the table is in 2NF and there are no transitive dependencies (i.e., non-key attributes depend on other non-key attributes).
        - BCNF (Boyce-Codd Normal Form): A stricter version of 3NF that ensures every determinant is a candidate key.
- **Denormalization -** Denormalization is the process of combining tables that were previously split during normalization. It involves intentionally introducing redundancy in order to improve performance, especially for read-heavy workloads. Denormalization can speed up query performance by reducing the number of joins and simplifying complex queries, but it sacrifices some degree of data integrity and increases storage requirements.
    - When performance is critical and read queries are more frequent than write operations.
    - When complex joins are slowing down the database and increasing response times.
    - When large-scale reporting requires quick aggregations or complex filtering.
    - **Denormalization Techniques**:
        - **Adding Redundant Data**: Copying data from one table into another to eliminate the need for joins.
        - **Combining Tables**: Merging tables that were previously normalized.

- **Pre-computing Aggregates**: Storing pre-calculated summary data to speed up query performance.

Aspect	Normalization	Denormalization
Purpose	Eliminate redundancy and ensure data integrity.	Improve query performance by reducing joins.
Data Integrity	High data integrity due to minimal redundancy.	Lower data integrity due to redundancy.
Complexity	More complex schema with multiple tables.	Simpler schema with fewer tables.
Storage	More efficient storage by reducing redundancy.	Increased storage due to redundancy.
Query Performance	Slower read queries due to joins.	Faster read queries by reducing joins.
Use Case	OLTP (Online Transaction Processing) systems.	OLAP (Online Analytical Processing) systems.

32. What are the Query Optimization techniques in SQL ? Explain in detail with examples?

Ans. SQL query optimization is the process of improving the performance of a SQL query by reducing its execution time and resource consumption (e.g., CPU, memory, I/O). Query optimization is essential in large-scale systems where fast query execution is critical. Below are some of the most common query optimization techniques with explanations and examples:

- **Indexing -** Indexes are database objects that improve the speed of data retrieval operations by allowing the database to quickly locate the required rows. They are most useful for columns that are frequently queried in WHERE clauses or used for sorting (ORDER BY).
- **Avoiding SELECT * (Wildcard Selection) -** Using SELECT * retrieves all columns from the table, which can be inefficient if you only need a few columns. Specifying the required columns explicitly can improve performance by reducing the amount of data retrieved.
- **Use of WHERE Clause for Filtering -** Filtering data as early as possible in the query using the WHERE clause can significantly reduce the number of rows processed. Applying filters on indexed columns will speed up the query further.
- **Use of Joins Effectively -** Joins are a common operation in SQL queries. Optimizing the use of joins can improve performance. Using the correct type of join (e.g., INNER JOIN, LEFT JOIN) and joining on indexed columns can improve performance.
- **Limiting the Results with LIMIT or TOP -** When you're only interested in a subset of rows, using LIMIT (in MySQL, PostgreSQL) or TOP (in SQL Server) can limit the number of rows returned, thus improving query performance.
- **Using Proper Data Types -** Using the appropriate data types for columns can reduce the amount of storage required and improve performance. For example, using INT instead of BIGINT when the values do not exceed the range of INT can reduce storage space and make data retrieval faster.
- **Avoiding Subqueries (Using Joins Instead) -** Subqueries (queries within queries) can be inefficient, especially if they are in the SELECT clause or the WHERE clause. Rewriting subqueries as joins often improves performance.

- **Using UNION Instead of OR (in Some Cases) -** The OR operator can be slow, especially when combined with multiple conditions. Using UNION instead of OR can sometimes improve performance because the database can execute the separate queries independently.
- **Avoiding Functions on Indexed Columns -** Avoid using functions (such as UPPER(), LOWER(), DATE(), etc.) on columns that are indexed in the WHERE clause, as this can prevent the database from using the index efficiently.
- **Using Query Execution Plans -** Most databases provide an execution plan for a query, which shows how the database engine executes the query. Analyzing the execution plan can help identify bottlenecks and areas where the query can be optimized.
- **Read About -** can we check how much time query take. Or what causing issue while running query