

SQL Interview Questions

1. What is SQL?

SQL (Structured Query Language) is a standard programming language used to communicate with **relational databases**. It allows users to create, read, update, and delete data, and provides commands to define database schema and manage database security.

2. What is a database?

A database is an **organized collection of data** stored electronically, typically structured in tables with rows and columns. It is managed by a **database management system (DBMS)**, which allows for efficient **storage, retrieval, and manipulation** of data.

3. What are the main types of SQL commands?

SQL commands are broadly classified into:

- **DDL (Data Definition Language):** CREATE, ALTER, DROP, TRUNCATE.
- **DML (Data Manipulation Language):** SELECT, INSERT, UPDATE, DELETE.
- **DCL (Data Control Language):** GRANT, REVOKE.
- **TCL (Transaction Control Language):** COMMIT, ROLLBACK, SAVEPOINT.

4. What is the difference between CHAR and VARCHAR2 data types?

- **CHAR:** Fixed-length storage. If the defined length is not fully used, it is padded with spaces.
- **VARCHAR2:** Variable-length storage. Only the actual data is stored, saving space when the full length is not needed.

5. What is a primary key?

A primary key is a unique identifier for each record in a table. It ensures that no two rows have the same value in the primary key column(s), and it does not allow NULL values.

6. What is a foreign key?

A foreign key is a column (or set of columns) in one table that refers to the primary key in another table. It establishes and enforces a relationship between the two tables, ensuring data integrity.

7. What is the purpose of the DEFAULT constraint?

The DEFAULT constraint assigns a default value to a column when no value is provided during an **INSERT operation**. This helps maintain consistent data and simplifies data entry.

8. What is normalization in databases?

Normalization is the process of organizing data in a database to **reduce redundancy** and **improve data integrity**. This involves dividing large tables into smaller, related tables and defining relationships between them to ensure consistency and avoid anomalies.

9. What is denormalization, and when is it used?

Denormalization is the process of combining **normalized tables** into larger tables for performance reasons. It is used when **complex queries** and joins slow down data retrieval, and the performance benefits outweigh the **drawbacks of redundancy**.

10. What is a query in SQL?

A query is a SQL statement used to retrieve, update, or manipulate data in a **database**. The most common type of query is a SELECT statement, which fetches data from one or more tables based on specified conditions.

11. What are the different operators available in SQL?

- **Arithmetic Operators:** +, -, *, /, %
- **Comparison Operators:** =, !=, <>, >, <, >=, <=
- **Logical Operators:** AND, OR, NOT
- **Set Operators:** UNION, INTERSECT, EXCEPT
- **Special Operators:** BETWEEN, IN, LIKE, IS NULL

12. What is a view in SQL?

A view is a **virtual table** created by a **SELECT query**. It does not store data itself, but presents data from one or more tables in a structured way. Views simplify complex queries, improve readability, and enhance security by restricting access to specific rows or columns.

13. What is the purpose of the UNIQUE constraint?

The UNIQUE constraint ensures that all values in a column (or combination of columns) are **distinct**. This prevents duplicate values and helps maintain data integrity.

14. What are the different types of joins in SQL?

- **INNER JOIN:** Returns rows that have matching values in both tables.
- **LEFT JOIN (LEFT OUTER JOIN):** Returns all rows from the left table, and matching rows from the right table.
- **RIGHT JOIN (RIGHT OUTER JOIN):** Returns all rows from the right table, and matching rows from the left table.
- **FULL JOIN (FULL OUTER JOIN):** Returns all rows when there is a match in either table.
- **CROSS JOIN:** Produces the Cartesian product of two tables.

15. What is the difference between INNER JOIN and OUTER JOIN?

- **INNER JOIN:** Returns only rows where there is a match in both tables.
- **OUTER JOIN:** Returns all rows from one table (LEFT, RIGHT, or FULL), and the matching rows from the other table. If there is no match, NULL values are returned for the non-matching side.

16. What is the purpose of the GROUP BY clause?

The GROUP BY clause is used to arrange **identical data** into **groups**. It is typically used with aggregate functions (such as COUNT, SUM, AVG) to perform calculations on each group rather than on the entire dataset.

17. What are aggregate functions in SQL?

Aggregate functions perform calculations on a set of values and return a single value. Common aggregate functions include:

- **COUNT():** Returns the number of rows.
- **SUM():** Returns the total sum of values.
- **AVG():** Returns the average of values.
- **MIN():** Returns the smallest value.
- **MAX():** Returns the largest value.

18. What is a subquery?

A subquery is a query nested within another query. It is often used in the **WHERE clause** to filter data based on the results of another query, making it easier to handle complex conditions.

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

- **WHERE:** Filters rows before any grouping takes place.
- **HAVING:** Filters grouped data after the GROUP BY clause has been applied.
In short, WHERE applies to individual rows, while HAVING applies to groups.

20. What are indexes, and why are they used?

Indexes are **database objects** that improve query performance by allowing **faster retrieval of rows**. They function like a book's index, making it quicker to find specific data without scanning the entire table. However, indexes require **additional storage** and can slightly slow down **data modification** operations.

21. What is the difference between DELETE and TRUNCATE commands?

- **DELETE:** Removes rows one at a time and records each deletion in the transaction log, allowing rollback. It can have a WHERE clause.
- **TRUNCATE:** Removes all rows at once without logging individual row deletions. It cannot have a WHERE clause and is faster than DELETE for large data sets.

22. What is the purpose of the SQL ORDER BY clause?

The ORDER BY clause sorts the result set of a query in either **ascending** (default) or **descending order**, based on one or more columns. This helps present the data in a more meaningful or readable sequence.

23. What are the differences between SQL and NoSQL databases?

- **SQL Databases:**
 - Use structured tables with rows and columns.
 - Rely on a fixed schema.
 - Offer **ACID** properties.
- **NoSQL Databases:**

- Use flexible, schema-less structures (e.g., key-value pairs, document stores).
- Are designed for horizontal scaling.
- Often focus on performance and scalability over strict consistency.

24. What is a table in SQL?

A table is a **structured collection** of related data organized into rows and columns. Columns define the type of data stored, while rows contain individual records.

25. What are the types of constraints in SQL?

Common constraints include:

- **NOT NULL:** Ensures a column cannot have NULL values.
- **UNIQUE:** Ensures all values in a column are distinct.
- **PRIMARY KEY:** Uniquely identifies each row in a table.
- **FOREIGN KEY:** Ensures referential integrity by linking to a primary key in another table.
- **CHECK:** Ensures that all values in a column satisfy a specific condition.
- **DEFAULT:** Sets a default value for a column when no value is specified.

26. What is a cursor in SQL?

A cursor is a database object used to **retrieve**, **manipulate**, and traverse through rows in a result set one row at a time. Cursors are helpful when performing operations that must be processed sequentially rather than in a set-based manner.

27. What is a trigger in SQL?

A trigger is a set of SQL statements that automatically execute in response to certain events on a table, such as **INSERT**, **UPDATE**, or **DELETE**. Triggers help maintain **data consistency**, enforce business rules, and implement complex integrity constraints.

28. What is the purpose of the SQL SELECT statement?

The SELECT statement retrieves data from one or more tables. It is the most commonly used command in SQL, allowing users to filter, sort, and display data based on specific criteria.

29. What are NULL values in SQL?

NULL represents a missing or unknown value. It is different from zero or an empty string. **NULL** values indicate that the data is not available or applicable.

30. What is a stored procedure?

A stored procedure is a precompiled set of SQL statements stored in the **database**. It can take input parameters, perform logic and queries, and return output values or result sets. Stored procedures improve **performance** and **Maintainability** by centralizing business logic.

SQL Intermediate Interview Questions

This section covers moderately complex **SQL topics** like **advanced queries**, **multi-table joins**, **subqueries**, and **basic optimization techniques**. These questions help enhance skills for both **database developers** and **administrators**, preparing us for more **technical SQL challenges** in the field.

31. What is the difference between DDL and DML commands?

1. DDL (Data Definition Language):

These commands are used to **define** and **modify** the structure of database objects such as **tables**, **indexes**, and **views**. For example, the **CREATE command** creates a new table, the **ALTER command** modifies an existing table, and the **DROP command** removes a table entirely. DDL commands primarily focus on the schema or structure of the database.

Example:

```
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

2. DML (Data Manipulation Language):

These commands deal with the **actual data stored** within database objects. For instance, the **INSERT command** adds rows of data to a table, the **UPDATE command** modifies existing data, and the **DELETE command** removes rows from a table. In short, DML commands allow you to query and manipulate the data itself rather than the structure.

Example:

```
INSERT INTO Employees (ID, Name) VALUES (1, 'Alice');
```

32. What is the purpose of the ALTER command in SQL?

The ALTER command is used to **modify the structure** of an existing database object. This command is essential for adapting our **database schema** as requirements evolve.

- Add or drop a column in a table.
- Change a column's data type.
- Add or remove constraints.
- Rename columns or tables.
- Adjust indexing or storage settings.

33. What is a composite primary key?

A **composite primary key** is a primary key made up of two or more columns. Together, these columns must form a unique combination for each row in the table. It's used when a single column isn't sufficient to uniquely identify a record.

Example:

Consider an Orders table where OrderID and ProductID together uniquely identify each record because multiple orders might include the same product, but not within the same order.

```
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
);
```

34. How is data integrity maintained in SQL databases?

Data integrity refers to the **accuracy, consistency, and reliability** of the data stored in the database. SQL databases maintain data integrity through several mechanisms:

- **Constraints:** Ensuring that certain conditions are always met. For example, NOT NULL ensures a column cannot have missing values, FOREIGN KEY ensures a valid relationship between tables, and UNIQUE ensures no duplicate values.

- **Transactions:** Ensuring that a series of operations either all succeed or all fail, preserving data consistency.
 - **Triggers:** Automatically enforcing rules or validations before or after changes to data.
 - **Normalization:** Organizing data into multiple related tables to minimize redundancy and prevent anomalies.
- These measures collectively ensure that the data remains reliable and meaningful over time.

35. What are the advantages of using stored procedures?

- **Improved Performance:** Stored procedures are precompiled and cached in the database, making their execution faster than sending multiple individual queries.
- **Reduced Network Traffic:** By executing complex logic on the server, fewer round trips between the application and database are needed.
- **Enhanced Security:** Stored procedures can restrict direct access to underlying tables, allowing users to execute only authorized operations.
- **Reusability and Maintenance:** Once a procedure is written, it can be reused across multiple applications. If business logic changes, you only need to update the stored procedure, not every application that uses it.

36. What is a UNION operation, and how is it used?

The UNION operator combines the result sets of two or more SELECT queries into a single result set, removing **duplicate rows**. The result sets must have the same number of columns and compatible data types for corresponding columns.

Example:

```
SELECT Name FROM Customers
UNION
SELECT Name FROM Employees;
```

37. What is the difference between UNION and UNION ALL?

- **UNION:** Combines result sets from two queries and removes **duplicate rows**, ensuring only unique records are returned.
- **UNION ALL:** Combines the result sets without removing duplicates, meaning all records from both queries are included.

- Performance-wise, UNION ALL is faster than UNION because it doesn't perform the additional operation of eliminating duplicates.

Example:

```
SELECT Name FROM Customers
UNION ALL
SELECT Name FROM Employees;
```

38. How does the CASE statement work in SQL?

The CASE statement is SQL's way of implementing **conditional logic** in queries. It evaluates conditions and returns a value based on the first condition that evaluates to true. If no condition is met, it can return a default value using the **ELSE clause**.

Example:

```
SELECT ID,
CASE
    WHEN Salary > 100000 THEN 'High'
    WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'
    ELSE 'Low'
END AS SalaryLevel
FROM Employees;
```

39. What are scalar functions in SQL?

Scalar functions operate on individual values and return a single value as a result. They are often used for formatting or converting data. Common examples include:

- **LEN()**: Returns the length of a string.
- **ROUND()**: Rounds a numeric value.
- **CONVERT()**: Converts a value from one data type to another.

Example:

```
SELECT LEN('Example') AS StringLength;
```

40. What is the purpose of the COALESCE function?

The **COALESCE function** returns the first non-NULL value from a list of expressions. It's commonly used to provide default values or handle missing data gracefully.

Example:

```
SELECT COALESCE(NULL, NULL, 'Default Value') AS Result;
```

41. What are the differences between SQL's COUNT() and SUM() functions?

- COUNT():** Counts the number of rows or non-NULL values in a column.

Example:

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

- SUM():** Adds up all numeric values in a column.

Example:

```
SELECT SUM(TotalAmount) FROM Orders;
```

42. What is the difference between the NVL and NVL2 functions?

- NVL():** Replaces a NULL value with a specified replacement value. **Example:** NVL(Salary, 0) will replace NULL with 0.
- NVL2():** Evaluates two arguments:
 - If the first argument is **NOT NULL**, returns the second argument.
 - If the first argument is **NULL**, returns the third argument.

Example:

```
SELECT NVL(Salary, 0) AS AdjustedSalary FROM Employees; -- Replaces NULL with 0
```

```
SELECT NVL2(Salary, Salary, 0) AS AdjustedSalary FROM Employees; -- If Salary is NULL, returns 0; otherwise, returns Salary.
```

43. How does the RANK() function differ from DENSE_RANK()?

- RANK():** Assigns a rank to each row, with gaps if there are ties.
- DENSE_RANK():** Assigns consecutive ranks without any gaps.

Example:

```
SELECT Name, Salary, RANK() OVER (ORDER BY Salary DESC) AS Rank
FROM Employees;
```

If two employees have the same salary, they get the same rank, but RANK() will skip a number for the next rank, while DENSE_RANK() will not.

44. What is the difference between ROW_NUMBER() and RANK()?

- **ROW_NUMBER():** Assigns a unique number to each row regardless of ties.
- **RANK():** Assigns the same number to tied rows and leaves gaps for subsequent ranks.

Example:

```
SELECT Name, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum  
FROM Employees;
```

45. What are common table expressions (CTEs) in SQL?

A CTE is a temporary result set defined within a query. It improves query readability and can be referenced multiple times.

Example:

```
WITH TopSalaries AS (  
    SELECT Name, Salary  
    FROM Employees  
    WHERE Salary > 50000  
)  
SELECT * FROM TopSalaries WHERE Name LIKE 'A%';
```

46. What are window functions, and how are they used?

Window functions allow you to perform calculations across a set of table rows that are related to the current row within a result set, without collapsing the result set into a single row. These functions can be used to compute running totals, moving averages, rank rows, etc.

Example: Calculating a running total

```
SELECT Name, Salary,  
       SUM(Salary) OVER (ORDER BY Salary) AS RunningTotal  
FROM Employees;
```

47. What is the difference between an index and a key in SQL?

1. Index

- An index is a database object created to **speed up data retrieval**. It stores a sorted reference to table data, which helps the database engine find rows more quickly than scanning the entire table.
- **Example:** A non-unique index on a column like LastName allows quick lookups of rows where the last name matches a specific value.

2. Key

- A key is a logical concept that enforces rules for uniqueness or relationships in the data.
- For instance, a **PRIMARY KEY** uniquely identifies each row in a table and ensures that no duplicate or NULL values exist in the key column(s).
- A **FOREIGN KEY** maintains referential integrity by linking rows in one table to rows in another.

48. How does indexing improve query performance?

Indexing allows the database to locate and access the rows corresponding to a **query condition** much faster than scanning the entire table. Instead of reading each row sequentially, the database uses the index to **jump directly** to the relevant data pages. This reduces the number of disk **I/O operations** and speeds up query execution, especially for large tables.

Example:

```
CREATE INDEX idx_lastname ON Employees(LastName);
SELECT * FROM Employees WHERE LastName = 'Smith';
```

The index on LastName lets the database quickly find all rows matching 'Smith' without scanning every record.

49. What are the trade-offs of using indexes in SQL databases?

Advantages

- Faster query performance, especially for SELECT queries with WHERE clauses, JOIN conditions, or ORDER BY clauses.
- Improved sorting and filtering efficiency.

Disadvantages:

- Increased storage space for the index structures.
- Additional overhead for write operations (INSERT, UPDATE, DELETE), as indexes must be updated whenever the underlying data changes.
- Potentially **slower bulk data loads** or batch inserts due to the need to maintain index integrity.
In short, indexes make read operations faster but can slow down write operations and increase storage requirements.

50. What is the difference between clustered and non-clustered indexes?

1. Clustered Index:

- Organizes the physical data in the table itself in the order of the indexed column(s).
- A table can have only one clustered index.
- Improves range queries and queries that sort data.
- Example: If EmployeeID is the clustered index, the rows in the table are stored physically sorted by EmployeeID.

2. Non-Clustered Index:

- Maintains a separate structure that contains a reference (or pointer) to the physical data in the table.
- A table can have multiple non-clustered indexes.
- Useful for specific query conditions that aren't related to the primary ordering of the data.
- Example: A non-clustered index on LastName allows fast lookups by last name even if the table is sorted by another column.

51. What are temporary tables, and how are they used?

Temporary tables are tables that exist only for the duration of a **session** or a **transaction**. They are useful for storing intermediate results, simplifying complex queries, or performing operations on subsets of data without modifying the main tables.

1. Local Temporary Tables:

- Prefixed with # (e.g., #TempTable).

- Only visible to the session that created them.
- Automatically dropped when the session ends.

2. Global Temporary Tables:

- Prefixed with ## (e.g., ##GlobalTempTable).
- Visible to all sessions.
- Dropped when all sessions that reference them are closed.

Example:

```
CREATE TABLE #TempResults (ID INT, Value VARCHAR(50));
INSERT INTO #TempResults VALUES (1, 'Test');
SELECT * FROM #TempResults;
```

52. What is a materialized view, and how does it differ from a standard view?

- **Standard View:**
 - A virtual table defined by a query.
 - Does not store data; the underlying query is executed each time the view is referenced.
 - A standard view shows real-time data.
- **Materialized View:**
 - A physical table that stores the result of the query.
 - Data is precomputed and stored, making reads faster.
 - Requires periodic refreshes to keep data up to date.
 - materialized view is used to store aggregated sales data, updated nightly, for fast reporting.

53. What is a sequence in SQL?

A sequence is a database object that generates a series of **unique numeric values**. It's often used to produce unique identifiers for primary keys or other columns requiring sequential values.

Example:

```
CREATE SEQUENCE seq_emp_id START WITH 1 INCREMENT BY 1;
SELECT NEXT VALUE FOR seq_emp_id; -- Returns 1
SELECT NEXT VALUE FOR seq_emp_id; -- Returns 2
```

54. What are the advantages of using sequences over identity columns?

1. Greater Flexibility:

- Can specify start values, increments, and maximum values.
- Can be easily reused for multiple tables.

2. Dynamic Adjustment: Can alter the sequence without modifying the table structure.

3. Cross-Table Consistency: Use a single sequence for multiple related tables to ensure unique identifiers across them.

In short, sequences offer more control and reusability than identity columns.

55. How do constraints improve database integrity?

Constraints enforce rules that the data must follow, preventing invalid or inconsistent data from being entered:

- **NOT NULL:** Ensures that a column cannot contain NULL values.
- **UNIQUE:** Ensures that all values in a column are distinct.
- **PRIMARY KEY:** Combines NOT NULL and UNIQUE, guaranteeing that each row is uniquely identifiable.
- **FOREIGN KEY:** Ensures referential integrity by requiring values in one table to match primary key values in another.
- **CHECK:** Validates that values meet specific criteria (e.g., CHECK (Salary > 0)).
By automatically enforcing these rules, constraints maintain data reliability and consistency.

56. What is the difference between a local and a global temporary table?

• Local Temporary Table:

- Prefixed with # (e.g., #TempTable).
- Exists only within the session that created it.
- Automatically dropped when the session ends.

- **Global Temporary Table:**

- Prefixed with ## (e.g., ##GlobalTempTable).
- Visible to all sessions.
- Dropped only when all sessions referencing it are closed.

Example:

```
CREATE TABLE #LocalTemp (ID INT);
CREATE TABLE ##GlobalTemp (ID INT);
```

57. What is the purpose of the SQL MERGE statement?

The **MERGE** statement combines multiple operations INSERT, UPDATE, and DELETE into one. It is used to synchronize two tables by:

- Inserting rows that don't exist in the target table.
- Updating rows that already exist.
- Deleting rows from the target table based on conditions

Example:

```
MERGE INTO TargetTable T
USING SourceTable S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.Value = S.Value
WHEN NOT MATCHED THEN
    INSERT (ID, Value) VALUES (S.ID, S.Value);
```

58. How can you handle duplicates in a query without using DISTINCT?**1. GROUP BY:** Aggregate rows to eliminate duplicates

```
SELECT Column1, MAX(Column2)
FROM TableName
GROUP BY Column1;
```

2. ROW_NUMBER(): Assign a unique number to each row and filter by that

```
WITH CTE AS (
    SELECT Column1, Column2, ROW_NUMBER() OVER (PARTITION BY Column1 ORDER BY
Column2) AS RowNum
    FROM TableName
)
SELECT * FROM CTE WHERE RowNum = 1;
```

59. What is a correlated subquery?

A correlated subquery is a subquery that references columns from the outer query. It is re-executed for each row processed by the outer query. This makes it more dynamic, but potentially less efficient.

Example:

```
SELECT Name,
    (SELECT COUNT(*)
        FROM Orders
        WHERE Orders.CustomerID = Customers.CustomerID) AS OrderCount
    FROM Customers;
```

60. What are partitioned tables, and when should we use them?

Partitioned tables divide data into **smaller, more manageable segments** based on a column's value (e.g., date or region). Each partition is stored separately, making queries that target a specific partition more efficient. It is used when

- Large tables with millions or billions of rows.
- Scenarios where queries frequently filter on partitioned columns (e.g., year, region).
- To improve maintenance operations, such as archiving older partitions without affecting the rest of the table.

SQL Advanced Interview Questions

This section covers **complex SQL topics**, including **performance tuning, complex indexing strategies, transaction isolation levels, and advanced query optimization techniques**. By tackling these challenging questions, we'll gain a deeper understanding of SQL, preparing us for **senior-level roles and technical interviews**.

61. What are the ACID properties of a transaction?

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability—four key properties that ensure database transactions are processed reliably.

1. Atomicity:

- A transaction is treated as a single unit of work, meaning all operations must succeed or fail as a whole.
- If any part of the transaction fails, the entire transaction is rolled back.

2. Consistency:

- A transaction must take the database from one valid state to another, maintaining all defined rules and constraints.
- This ensures data integrity is preserved throughout the transaction process.

3. Isolation:

- Transactions should not interfere with each other.
- Even if multiple transactions occur simultaneously, each must operate as if it were the only one in the system until it is complete.

4. Durability:

- Once a transaction is committed, its changes must persist, even in the event of a system failure.
- This ensures the data remains stable after the transaction is successfully completed.

62. What are the differences between isolation levels in SQL?

Isolation levels define the extent to which the operations in one transaction are isolated from those in other transactions. They are critical for **managing concurrency** and ensuring data integrity. Common isolation levels include:

1. Read Uncommitted:

- Allows reading uncommitted changes from other transactions.
- Can result in dirty reads, where a transaction reads data that might later be rolled back.

2. Read Committed:

- Ensures a transaction can only read committed data.
- Prevents dirty reads but does not protect against non-repeatable reads or phantom reads.

3. Repeatable Read:

- Ensures that if a transaction reads a row, that row cannot change until the transaction is complete.
- Prevents dirty reads and non-repeatable reads but not phantom reads.

4. Serializable:

- The highest level of isolation.
- Ensures full isolation by effectively serializing transactions, meaning no other transaction can read or modify data that another transaction is using.
- Prevents dirty reads, non-repeatable reads, and phantom reads, but may introduce performance overhead due to locking and reduced concurrency.

63. What is the purpose of the WITH (NOLOCK) hint in SQL Server?

- The **WITH (NOLOCK)** hint allows a query to read data without acquiring shared locks, effectively reading uncommitted data.
- It can improve performance by **reducing contention for locks**, especially on large tables that are frequently updated.
- Results may be inconsistent or unreliable, as the data read might change or be rolled back.

Example:

```
SELECT *  
FROM Orders WITH (NOLOCK);
```

This query fetches data from the Orders table without waiting for other transactions to release their locks.

64. How do you handle deadlocks in SQL databases?

Deadlocks occur when two or more transactions hold resources that the other transactions need, resulting in a cycle of dependency that prevents progress. Strategies to handle deadlocks include:

1. Deadlock detection and retry:

- Many database systems have mechanisms to detect deadlocks and terminate one of the transactions to break the cycle.
- The terminated transaction can be retried after the other transactions complete.

2. Reducing lock contention:

- Use indexes and optimized queries to minimize the duration and scope of locks.
- Break transactions into smaller steps to reduce the likelihood of conflicts.

3. Using proper isolation levels:

- In some cases, lower isolation levels can help reduce locking.
- Conversely, higher isolation levels (like Serializable) may ensure a predictable order of operations, reducing deadlock risk.

4. Consistent ordering of resource access:

- Ensure that transactions acquire resources in the same order to prevent cyclical dependencies.

65. What is a database snapshot, and how is it used?

A database snapshot is a read-only, static view of a database at a specific point in time.

- **Reporting:** Allowing users to query a consistent dataset without affecting live operations.
- **Backup and recovery:** Snapshots can serve as a point-in-time recovery source if changes need to be reversed.
- **Testing:** Providing a stable dataset for testing purposes without the risk of modifying the original data.

Example:

```
CREATE DATABASE MySnapshot ON
```

```
(
```

```
NAME = MyDatabase_Data,  
FILENAME = 'C:\Snapshots\MyDatabase_Snapshot.ss'  
)  
AS SNAPSHOT OF MyDatabase;
```

66. What are the differences between OLTP and OLAP systems?

1. OLTP (Online Transaction Processing)

- Handles large volumes of simple transactions (e.g., order entry, inventory updates).
- Optimized for fast, frequent reads and writes.
- Normalized schema to ensure data integrity and consistency.
- Examples: e-commerce sites, banking systems.

2. OLAP (Online Analytical Processing)

- Handles complex queries and analysis on large datasets.
- Optimized for read-heavy workloads and data aggregation.
- Denormalized schema (e.g., star or snowflake schemas) to support faster querying.
- Examples: Business intelligence reporting, data warehousing.

67. What is a live lock, and how does it differ from a deadlock?

1. Live Lock

- Occurs when two or more transactions keep responding to each other's changes, but no progress is made.
- Unlike a deadlock, the transactions are not blocked; they are actively running, but they cannot complete.

2. Deadlock

- A **deadlock** occurs when two or more transactions are waiting on each other's resources indefinitely, blocking all progress.
- No progress can be made unless one of the transactions is terminated

68. What is the purpose of the SQL EXCEPT operator?

The EXCEPT operator is used to return rows from one query's result set that are not present in another query's result set. It effectively performs a set difference, showing only the data that is **unique** to the first query.

Example:

```
SELECT ProductID FROM ProductsSold
EXCEPT
SELECT ProductID FROM ProductsReturned;
```

Use Case:

- To find discrepancies between datasets.
- To verify that certain data exists in one dataset but not in another.

Performance Considerations:

- **EXCEPT works** best when the datasets involved have appropriate indexing and when the result sets are relatively small.
- Large datasets without indexes may cause slower performance because the database has to compare each row.

69. How do you implement dynamic SQL, and what are its advantages and risks?

Dynamic SQL is SQL code that is constructed and executed at **runtime** rather than being fully defined and static. In SQL Server: Use `sp_executesql` or EXEC. In other databases: Concatenate query strings and execute them using the respective command for the database platform.

Syntax:

```
DECLARE @sql NVARCHAR(MAX)
SET @sql = 'SELECT * FROM ' + @TableName
EXEC sp_executesql @sql;
```

Advantages:

- **Flexibility:** Dynamic SQL can adapt to different conditions, tables, or columns that are only known at runtime.
- **Simplifies Complex Logic:** Instead of writing multiple queries, a single dynamically constructed query can handle multiple scenarios.

Risks:

- **SQL Injection Vulnerabilities:** If user input is not sanitized, attackers can inject malicious SQL code.
- **Performance Overhead:** Because dynamic SQL is constructed at runtime, it may not benefit from cached execution plans, leading to slower performance.
- **Complexity in Debugging:** Dynamic queries can be harder to read and troubleshoot.

70. What is the difference between horizontal and vertical partitioning?

Partitioning is a database technique used to divide data into smaller, more manageable pieces.

- **Horizontal Partitioning:**

- Divides the rows of a table into multiple partitions based on values in a specific column.
- Example: Splitting a customer table into separate partitions by geographic region or by year.
- **Use Case:** When dealing with large datasets, horizontal partitioning can improve performance by limiting the number of rows scanned for a query.

- **Vertical Partitioning:**

- Divides the columns of a table into multiple partitions.
- Example: Storing infrequently accessed columns (e.g., large text or binary fields) in a separate table or partition.
- **Use Case:** Helps in optimizing storage and query performance by separating commonly used columns from less frequently accessed data.

- **Key Difference:**

- Horizontal partitioning is row-based, focusing on distributing the dataset's rows across partitions.
- Vertical partitioning is column-based, aiming to separate less-used columns into different partitions or tables.

71. What are the considerations for indexing very large tables?

1. Indexing Strategy:

- Focus on the most frequently queried columns or those involved in JOIN and WHERE conditions.
- Avoid indexing every column, as it increases storage and maintenance costs.

2. Index Types:

- Use clustered indexes for primary key lookups and range queries.
- Use non-clustered indexes for filtering, ordering, and covering specific queries.

3. Partitioned Indexes:

- If the table is partitioned, consider creating **local indexes** for each partition. This improves manageability and can speed up queries targeting specific partitions.

4. Maintenance Overhead:

- Index rebuilding and updating can be resource-intensive. Plan for regular index maintenance during off-peak hours.
- Monitor index fragmentation and rebuild indexes as necessary to maintain performance.

5. Monitoring and Tuning:

- Continuously evaluate query performance using execution plans and statistics.
- Remove unused or rarely accessed indexes to reduce maintenance costs.

6. Indexing large tables requires a careful approach to ensure that performance gains from faster queries outweigh the costs of increased storage and maintenance effort.

72. What is the difference between database sharding and partitioning?

1. Sharding

- Sharding involves splitting a database into multiple smaller, **independent databases** (shards). Each shard operates on a subset of the overall data and can be hosted on separate servers.
- Sharding is a horizontal scaling strategy that distributes data across multiple databases, typically to handle massive data volumes and high traffic.

- **Purpose:** Horizontal scaling to handle large volumes of data and high query loads.
- **Example:** A global user database might be divided into shards by region, such as a shard for North America, Europe, and Asia.
- **Key Benefit:** Each shard can be queried independently, reducing the load on any single server.

2. Partitioning

- Partitioning splits a single table into smaller, logical pieces, usually within the same database.
- Partitioning is a **logical organization of data** within a single database to optimize performance and manageability.
- **Purpose:** Improve query performance by reducing the amount of data scanned, and simplify maintenance tasks such as archiving or purging old data.
- **Example:** A sales table could be partitioned by year so that queries targeting recent sales do not need to scan historical data.

73. What are the best practices for writing optimized SQL queries?

1. Write Simple, Clear Queries:

- Avoid overly complex joins and subqueries.
- Use straightforward, well-structured SQL that is easy to read and maintain.

2. Filter Data Early:

- Apply WHERE clauses as early as possible to reduce the amount of data processed.
- Consider using indexed columns in WHERE clauses for faster lookups.

3. **Avoid SELECT *:

- Retrieve only the columns needed. This reduces I/O and improves performance.

4. Use Indexes Wisely:

- Create indexes on columns that are frequently used in WHERE clauses, JOIN conditions, and ORDER BY clauses.
- Regularly review index usage and remove unused indexes.

5. Leverage Query Execution Plans:

- Use execution plans to identify bottlenecks, missing indexes, or inefficient query patterns.

6. Use Appropriate Join Types:

- Choose INNER JOIN, LEFT JOIN, or OUTER JOIN based on the data relationships and performance requirements.

7. Break Down Complex Queries:

- Instead of a single monolithic query, use temporary tables or CTEs to process data in stages.

8. Optimize Aggregations:

- Use GROUP BY and aggregate functions efficiently.
- Consider pre-aggregating data if queries frequently require the same computations.

9. Monitor Performance Regularly:

- Continuously analyze query performance and fine-tune as data volumes grow or usage patterns change.

74. How can you monitor query performance in a production database?

1. Use Execution Plans:

Review the execution plan of queries to understand how the database is retrieving data, which indexes are being used, and where potential bottlenecks exist.

2. Analyze Wait Statistics:

Identify where queries are waiting, such as on locks, I/O, or CPU, to pinpoint the cause of slowdowns.

3. Leverage Built-in Monitoring Tools:

- SQL Server: Use Query Store, DMVs (Dynamic Management Views), and performance dashboards.
- MySQL: Use EXPLAIN, SHOW PROFILE, and the Performance Schema.
- PostgreSQL: Use EXPLAIN (ANALYZE), pg_stat_statements, and log-based monitoring.

4. Set Up Alerts and Baselines:

- Monitor key performance metrics (query duration, IOPS, CPU usage) and set thresholds.
- Establish baselines to quickly identify when performance degrades.

5. Continuous Query Tuning:

- Regularly revisit and tune queries as data grows or application requirements change.
- Remove unused or inefficient indexes and re-evaluate the indexing strategy.

75. What are the trade-offs of using indexing versus denormalization?

1. Indexing

- **Advantages:**
 - Speeds up read operations and improves query performance without changing the data structure.
 - Can be applied incrementally and is reversible if not effective.
 - Consider indexing when you need faster lookups without altering the data model.
- **Disadvantages:**
 - Slows down write operations as indexes need to be maintained.
 - Requires additional storage.

2. Denormalization

- **Advantages:**
 - Simplifies query logic by storing pre-joined or aggregated data.
 - Can improve performance for read-heavy workloads where complex joins are frequent.
 - Consider denormalization when complex joins or repeated aggregations significantly slow down queries
- **Disadvantages:**

- Introduces data redundancy, which can lead to inconsistencies.
- Increases storage requirements.
- Makes updates more complex, as redundant data must be synchronized.

76. How does SQL handle recursive queries?

SQL handles **recursive queries** using **Common Table Expressions (CTEs)**. A recursive CTE repeatedly references itself to process hierarchical or tree-structured data.

Key Components:

- **Anchor Member:** The initial query that starts the recursion.
- **Recursive Member:** A query that references the CTE to continue building the result set.
- **Termination Condition:** Ensures that recursion stops after a certain depth or condition is met.

Example:

```
WITH RecursiveCTE (ID, ParentID, Depth) AS (
    SELECT ID, ParentID, 1 AS Depth
    FROM Categories
    WHERE ParentID IS NULL
    UNION ALL
    SELECT c.ID, c.ParentID, r.Depth + 1
    FROM Categories c
    INNER JOIN RecursiveCTE r
    ON c.ParentID = r.ID
)
SELECT * FROM RecursiveCTE;
```

77. What are the differences between transactional and analytical queries?

1. Transactional Queries:

- Focus on individual, short-term operations such as inserts, updates, and deletes.
- Optimize for high-throughput and low-latency.
- Often used in OLTP (Online Transaction Processing) systems.

2. Analytical Queries:

- Involve complex aggregations, multi-dimensional analysis, and data transformations.
- Typically read-heavy, processing large amounts of historical or aggregated data.
- Often used in OLAP (Online Analytical Processing) systems.

3. Key Differences:

- **Transactional queries** support day-to-day operations and maintain data integrity.
- **Analytical queries** support decision-making by providing insights from large datasets

78. How can you ensure data consistency across distributed databases?

1. **Use Distributed Transactions:** Implement two-phase commit (2PC) to ensure all participating databases commit changes simultaneously or roll back if any part fails.
2. **Implement Eventual Consistency:** If strong consistency isn't required, allow data to become consistent over time. This approach is common in distributed systems where high availability is a priority.
3. **Conflict Resolution Mechanisms:** Use versioning, timestamps, or conflict detection rules to resolve inconsistencies.
4. **Data Replication and Synchronization:** Use reliable replication strategies to ensure that changes made in one database are propagated to others.
5. **Regular Audits and Validation:** Periodically verify that data remains consistent across databases and fix discrepancies as needed.

79. What is the purpose of the SQL PIVOT operator?

The PIVOT operator transforms rows into columns, making it easier to summarize or rearrange data for reporting.

Example:

Converting a dataset that lists monthly sales into a format that displays each month as a separate column.

```
SELECT ProductID, [2021], [2022]
FROM (
    SELECT ProductID, YEAR(SaleDate) AS SaleYear, Amount
```

```
FROM Sales
) AS Source
PIVOT (
    SUM(Amount)
    FOR SaleYear IN ([2021], [2022])
) AS PivotTable;
```

80. What is a bitmap index, and how does it differ from a B-tree index?

1. Bitmap Index:

- Represents data with bitmaps (arrays of bits) to indicate the presence or absence of a value in each row.
- Efficient for low-cardinality columns, such as “gender” or “yes/no” fields.
- Can perform fast logical operations (AND, OR, NOT) on multiple columns simultaneously.

2. B-tree Index:

- Uses a balanced tree structure to store indexed data in a sorted order.
- Suitable for high-cardinality columns (e.g., unique identifiers, large ranges of values).
- Supports range-based queries efficiently.

3. Key Difference:

- Bitmap indexes excel with low-cardinality data and complex boolean conditions.
- B-tree indexes are better for unique or high-cardinality data and range queries.

Query Based SQL Interview Questions

This section is dedicated to questions that focus on **writing** and **understanding SQL queries**. By practicing these examples, we'll learn **how to retrieve, manipulate, and analyze data** effectively, building the **problem-solving skills** needed for **real-world scenarios**.

81. Write a query to find the second-highest salary of an employee in a table.

```
SELECT MAX(Salary) AS SecondHighestSalary
FROM Employee
WHERE Salary < (SELECT MAX(Salary) FROM Employee);
```

Explanation:

This query identifies the second-highest salary by selecting the maximum salary that is less than the overall highest salary. The subquery determines the top salary, while the outer query finds the next highest value.

82. Write a query to retrieve employees who earn more than the average salary.

```
SELECT *
FROM Employee
WHERE Salary > (SELECT AVG(Salary) FROM Employee);
```

Explanation:

This query fetches details of employees whose salary exceeds the average salary. The subquery calculates the average salary, and the main query filters rows based on that result.

83. Write a query to fetch the duplicate values from a column in a table.

```
SELECT ColumnName, COUNT(*)
FROM TableName
GROUP BY ColumnName
HAVING COUNT(*) > 1;
```

Explanation:

The query uses GROUP BY to group identical values and HAVING COUNT(*) > 1 to identify values that appear more than once in the specified column.

84. Write a query to find the employees who joined in the last 30 days.

```
SELECT *
FROM Employee
WHERE JoiningDate > DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

Explanation:

By comparing the JoiningDate to the current date minus 30 days, this query retrieves all employees who joined within the last month.

85. Write a query to fetch top 3 earning employees.

```
SELECT *
FROM Employee
ORDER BY Salary DESC
LIMIT 3;
```

Explanation:

The query sorts employees by salary in descending order and uses LIMIT 3 to return only the top three earners.

86. Write a query to delete duplicate rows in a table without using the ROWID keyword.

```
DELETE FROM Employee
WHERE EmployeeID NOT IN (
    SELECT MIN(EmployeeID)
    FROM Employee
    GROUP BY Column1, Column2
);
```

Explanation:

This query retains only one row for each set of duplicates by keeping the row with the smallest EmployeeID. It identifies duplicates using GROUP BY and removes rows not matching the minimum ID.

87. Write a query to fetch common records from two tables.

```
SELECT *
FROM TableA
INNER JOIN TableB ON TableA.ID = TableB.ID;
```

Explanation:

An INNER JOIN is used to find rows present in both tables by matching a common column (in this case, ID).

88. Write a query to fetch employees whose names start and end with 'A'.

```
SELECT *
FROM Employee
WHERE Name LIKE 'A%' AND Name LIKE '%A';
```

Explanation:

The query uses LIKE with wildcard characters to filter rows where the Name column starts and ends with the letter 'A'.

89. Write a query to display all departments along with the number of employees in each.

```
SELECT DepartmentID, COUNT(*) AS EmployeeCount  
FROM Employee  
GROUP BY DepartmentID;
```

Explanation:

By grouping employees by their DepartmentID and counting rows in each group, the query produces a list of departments along with the employee count.

90. Write a query to find employees who do not have managers.

```
SELECT *  
FROM Employee  
WHERE ManagerID IS NULL;
```

Explanation:

This query selects employees whose ManagerID column is NULL, indicating they don't report to a manager.

91. Write a query to fetch the 3rd and 4th highest salaries.

```
WITH SalaryRank AS (  
    SELECT Salary, RANK() OVER (ORDER BY Salary DESC) AS Rank  
    FROM Employee  
)  
SELECT Salary  
FROM SalaryRank  
WHERE Rank IN (3, 4);
```

Explanation:

This query uses the **RANK()** window function to rank the salaries in descending order. The outer query then selects the 3rd and 4th highest salaries by filtering for those ranks.

92. Write a query to transpose rows into columns.

```
SELECT  
    MAX(CASE WHEN ColumnName = 'Condition1' THEN Value END) AS Column1,
```

```
MAX(CASE WHEN ColumnName = 'Condition2' THEN Value END) AS Column2  
FROM TableName;
```

Explanation:

This query converts specific row values into columns using conditional aggregation with CASE. Each column's value is determined based on a condition applied to rows.

93. Write a query to fetch records updated within the last hour.

```
SELECT *  
FROM TableName  
WHERE UpdatedAt >= NOW() - INTERVAL 1 HOUR;
```

Explanation:

By comparing the UpdatedAt timestamp to the current time minus one hour, the query retrieves rows updated in the last 60 minutes.

94. Write a query to list employees in departments that have fewer than 5 employees.

```
SELECT *  
FROM Employee  
WHERE DepartmentID IN (  
    SELECT DepartmentID  
    FROM Employee  
    GROUP BY DepartmentID  
    HAVING COUNT(*) < 5  
);
```

Explanation:

The subquery counts employees in each department, and the main query uses those results to find employees working in departments with fewer than 5 members.

95. Write a query to check if a table contains any records.

```
SELECT CASE  
        WHEN EXISTS (SELECT * FROM TableName) THEN 'Has Records'  
        ELSE 'No Records'  
        END AS Status;
```

Explanation:

The query uses EXISTS to determine if any rows exist in the table, returning a status of 'Has Records' or 'No Records' based on the result.

96. Write a query to find employees whose salaries are higher than their managers.

```
SELECT e.EmployeeID, e.Salary  
FROM Employee e  
JOIN Employee m ON e.ManagerID = m.EmployeeID  
WHERE e.Salary > m.Salary;
```

Explanation:

This query joins the Employee table with itself to compare employee salaries to their respective managers' salaries, selecting those who earn more.

97. Write a query to fetch alternating rows from a table.

```
WITH RowNumbered AS (  
    SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum  
    FROM Employee  
)  
SELECT *  
FROM RowNumbered  
WHERE RowNum % 2 = 0;
```

Explanation:

This query assigns a sequential number to each row using ROW_NUMBER(), then selects rows where the row number is even, effectively fetching alternating rows. The ORDER BY (SELECT NULL) is used to avoid any specific ordering and just apply a sequential numbering.

98. Write a query to find departments with the highest average salary.

```
SELECT DepartmentID  
FROM Employee  
GROUP BY DepartmentID  
ORDER BY AVG(Salary) DESC  
LIMIT 1;
```

Explanation:

Grouping by DepartmentID and ordering by the average salary in descending order, the query returns the department with the highest average.

99. Write a query to fetch the nth record from a table.

```
WITH OrderedEmployees AS (
    SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum
    FROM Employee
)
SELECT *
FROM OrderedEmployees
WHERE RowNum = n;
```

Explanation:

This query uses **ROW_NUMBER()** to generate a sequential number for each row. The outer query then retrieves the row where the number matches the desired nth position. The approach is portable across most databases.

100. Write a query to find employees hired in the same month of any year.

```
SELECT *
FROM Employee
WHERE MONTH(JoiningDate) = MONTH(CURDATE());
```

Explanation:

By comparing the month of JoiningDate to the current month, the query selects all employees who were hired in that month regardless of the year.

SQL FAQ's**1. What is SQL?**

SQL (Structured Query Language) is a programming language used to manipulate and manage data in relational databases. SQL is used to communicate and connect with a database to create, modify, retrieve, and delete data.

Some everyday tasks performed using SQL include creating tables and views, inserting and updating data, selecting specific data from tables, and deleting data from tables.

2. What is a database?

A database is an organized collection of structured information or data, typically stored electronically in a computer system. It allows users to store, manage, retrieve, and update data efficiently through database management systems (DBMS).

3. What is a primary key, and why is it important in a database?

A primary key is a crucial component of a Database as it serves as a unique identifier in a table for each record. The primary key enables efficient data retrieval and manipulation while ensuring data integrity by preventing duplicate entries.

Using a primary key, a database management system can quickly locate and retrieve data from a table without scanning the entire table. This makes data retrieval more efficient, especially in large databases with many records. Additionally, primary keys allow for the easy creation of relationships between tables, simplifying complex queries and making database maintenance easier.

4. What is a foreign key, and how is it different from a primary key?

A foreign key is a database constraint that establishes a link between two tables in a relational database. It is used to maintain referential integrity. It is a field or set of fields in one table that also refers to the primary key of another table. It differs from a primary key in that it does not have to be unique and can be used to establish relationships between tables.

5. What is the difference between a LEFT JOIN and a RIGHT JOIN in SQL?

In SQL, a LEFT JOIN and a RIGHT JOIN are both types of outer join that can be used to combine data from two or more tables. The difference between them lies in which table's data is preserved if there is no matching data in the other table.

- In a LEFT JOIN, all the rows from the table on the left-hand side of the JOIN keyword (the "left table") are included in the result set, even if there is no matching data in the table on the right-hand side (the "right table").
- In the Right JOIN, all the rows from the table on the right-hand side of the JOIN keyword (the "right table") are included in the result set, even if there is no matching data in the left table.

In summary, the difference between a LEFT JOIN and a RIGHT JOIN is the table whose data is preserved when there is no match in the other table.

6. How would you retrieve all the records from a "customers" table in SQL?

Ans: To retrieve all the records from a table called "customers" in SQL, you would use the following query:

```
SELECT * FROM customers;
```

7. What is a subquery, and how is it used in SQL?

A subquery is a query that is embedded within another query. It retrieves data used in the main query as a filter or a condition.

Syntax:

```
SELECT column1, column2
```

```
FROM table1
```

```
WHERE column3 IN (SELECT column3 FROM table2 WHERE condition);
```

8. What is the difference between SQL's WHERE and HAVING clauses?

In SQL, the WHERE clause is used to filter rows based on a condition on a column, while the HAVING clause is used to filter groups based on an aggregate function.

The WHERE clause is applied before any grouping takes place and filters individual rows based on a condition. On the other hand, the HAVING clause is applied after the grouping and filter groups based on the results of aggregate functions such as COUNT, SUM, AVG, etc.

9. What is the difference between a function and a stored procedure in SQL?

In SQL, a function returns a value, while a stored procedure does not necessarily return a value and may execute a series of operations or tasks. Functions can be used as part of a SQL statement or expression to return a value. In contrast, stored procedures can be used to encapsulate a series of SQL statements and can be executed as a single unit. Additionally, functions can be used within stored procedures, but stored procedures cannot be used within functions.

10. Write a SQL query to find the second-highest salary from an employee table.

Assuming we have the following "employees" table:

```
CREATE TABLE employees (
```

```
    id INT PRIMARY KEY,
```

```
name VARCHAR(255) NOT NULL,  
salary INT  
);
```

```
INSERT INTO employees (id, name, salary) VALUES (1, 'Sangeeta', 100000);
```

```
INSERT INTO employees (id, name, salary) VALUES (2, 'Ranjita', 150000);
```

```
INSERT INTO employees (id, name, salary) VALUES (3, 'Anita', 70000);
```

```
INSERT INTO employees (id, name, salary) VALUES (4, 'Sunita', 50000);
```

```
INSERT INTO employees (id, name, salary) VALUES (5, 'Anjeeta', 90000);
```

```
SELECT MAX(salary) as second_highest_salary
```

```
FROM employees
```

```
WHERE salary < (SELECT MAX(salary) FROM employees)
```

Output:

```
second_highest_salary
```

```
100000
```

11. What is the purpose of an index in SQL, and how does it work?

An index is used to improve the performance of queries by allowing for faster data retrieval. It creates a separate data structure that stores the values of one or more columns and allows faster access to the data based on those values.

12. What is the difference between NOSQL and SQL?

SQL	NOSQL
It is a relational data model.	It is a non-relational data model.
Vertical scaling is more common.	Supports horizontal scaling.
Handles structured data.	Handles large and unstructured data.
Fixed schema structure.	Flexible structure.
Schema-based.	Schema-less.

13. Write a SQL query to find the names of employees who have not been assigned to any project.

We have created an Employee table and a Project table

```
CREATE TABLE employees (
```

```
    employee_id INT PRIMARY KEY,
```

```
    name VARCHAR(50)
```

```
);
```

```
INSERT INTO employees (employee_id, name)
```

```
VALUES
```

```
    (1, 'Raghav'),
```

```
    (2, 'Raashi'),
```

```
    (3, 'Rohan'),
```

```
    (4, 'Mohan');
```

Employees

employee_id	name
1	Raghav
2	Raashi
3	Rohan
4	Mohan

```
CREATE TABLE projects (
    project_id INT PRIMARY KEY,
    name VARCHAR(50),
    employee_id INT,
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id)
);
```

```
INSERT INTO projects (project_id, name, employee_id)
VALUES
    (1, 'Project A', 1),
    (2, 'Project B', 2),
    (3, 'Project C', 1),
    (4, 'Project D', 3);
```

Projects

project_id	name	employee_id
1	Project A	1
2	Project B	2
3	Project C	1
4	Project D	3

```

SELECT employees.name
FROM employees
LEFT JOIN projects
ON employees.employee_id = projects.employee_id
WHERE projects.employee_id IS NULL;
    
```

Output:

name
Mohan

Explanation:

This is because the employee Mohan has not been assigned to any project, as there are no corresponding rows in the "projects" table with their respective employee IDs. The rest have been assigned to at least one project, so they are not included in the output.

14. In the Student table, the marks column contains a list of values separated by commas. How can you determine the number of values in this comma-separated list?

```
CREATE TABLE Student (
```

```

id INT NOT NULL,
name VARCHAR(50) NOT NULL,
marks VARCHAR(255) NOT NULL,
PRIMARY KEY (id)
);

```

INSERT INTO Student (id, name, marks)

VALUES (1, 'Rohit', '87,92,76,89');

Student

id	name	marks
1	Rohit	87,92,76,89

SELECT id, name, marks, LENGTH(marks) - LENGTH(REPLACE(marks, ',', '')) + 1 AS num_marks

FROM Student

WHERE id = 1;

Output:

id	name	marks	num_marks
1	Rohit	87,92,76,89	4

Student
id

15. State the difference between cross-join and natural join.

Features	CROSS JOIN	NATURAL JOIN
Definition	Returns the Cartesian product of the two tables, meaning every row from the first table	Joins two tables based on columns with the same name, producing a result set with only

Features	CROSS JOIN	NATURAL JOIN
	is combined with every row from the second table.	one column for each pair of same-named columns.
Syntax	SELECT * FROM table1 CROSS JOIN table2;	SELECT * FROM table1 NATURAL JOIN table2;
Join Condition	Joins every row of the first table to every row of the second table.	Joins the two tables based on columns with the same name.
Resulting Rows	Total rows in table1 multiplied by total rows in table2.	Rows where the values in columns with the same name are equal.
Performance	It can be slow for large tables, generates Cartesian product.	More efficient than CROSS JOIN but may require specifying columns explicitly.
Usage	Used when no columns exist to join tables or when you want all possible combinations of rows from the two tables.	Used when two tables have at least one column with the same name, and you want to join them based on that column.

16. List the different types of relationships in SQL.

In SQL, there are four main types of relationships:

- **One-to-One (1:1) Relationship:** In this type of relationship, each record in the first table is associated with only one record in the second table, and vice versa. This is typically used when the two tables have a common attribute or key, and the second

table contains additional information about the first table.

- **One-to-Many (1:N) Relationship:** In this type of relationship, each record in the first table can be associated with multiple records in the second table, but each record in the second table is associated with only one record in the first table. This is used when one record in the first table can have multiple related records in the second table.
- **Many-to-One (N:1) Relationship:** This is the inverse of the one-to-many relationship. In this type of relationship, each record in the second table can be associated with multiple records in the first table, but each record in the first table is associated with only one record in the second table. This is used when multiple records in the second table are related to one record in the first table.
- **Many-to-Many (N:N) Relationship:** In this type of relationship, each record in the first table can be associated with multiple records in the second table and vice versa. This requires the use of an intermediary table, often called a junction table, which contains foreign keys to both the first and second tables. This is used when multiple records in each table can be related to multiple records in the other table.

17. What are Tables and Fields?

In a relational database, a table is a collection of data organized in rows and columns. Tables are used to store and manage data in a structured way, with each row representing a unique record and each column representing specific information about the record. Tables are often named according to the type of data they contain, such as "customers", "orders", or "employees".

A **field**, also known as a column or an attribute, is a single information stored in a table. Each field is named and has a specific data type, such as text, number, date, or Boolean, that determines the type of data that can be stored in the field. For example, a "customers" table might have fields for the customer's name, address, phone number, and email address.

Fields can also have other properties, such as a maximum length or a default value, that define how the data is stored and how it can be used. In addition, fields can be used to define relationships between tables by referencing the primary key of another table or by creating foreign keys that link related records across different tables.

Tables	Fields
customer's	customer_id, name, email, phone, and address.
orders	order_id, customer_id, order_date, total_amount.
products	product_id, name, description, price, quantity.
employees	employee_id, first_name, last_name, job_title, hire_date.

Together, tables and fields form the basic building blocks of a relational database, providing a flexible and powerful way to store, manage, and query large amounts of data in a structured and organized way.

18. What is the difference between the SQL statements DELETE and TRUNCATE?

Statement	DELETE	TRUNCATE
Syntax	DELETE FROM table_name WHERE condition;	TRUNCATE TABLE table_name;
Functionality	Deletes specific rows that match the WHERE condition.	Removes all rows from the table.
Auto-increment	Does not reset auto-increment values.	Resets auto-increment values to their starting value.
Transaction	It can be rolled back within a transaction.	It cannot be rolled back within a transaction.

Statement	DELETE	TRUNCATE
Logging	Logs, each row deletion, can be slower for large tables.	It does not log individual row deletions but can be faster for large tables.
Access	Requires to DELETE privileges on the table.	Requires to DROP and CREATE privileges on the table.

19. How should data be structured to support Join Operations in a one-to-many relationship?

In a one-to-many relationship, where one record in the primary table is related to many records in the related table, the data should be structured in a way that supports efficient join operations.

A practical method to establish a relationship between two tables is to utilize a foreign key column within the related table that points to the primary key column in the primary table. This allows the database to quickly locate all the related records for a given primary record.

Consider two tables

```
CREATE TABLE customers_Table (
```

```
    customer_id INT PRIMARY KEY,
```

```
    name VARCHAR(50),
```

```
    email VARCHAR(100)
```

```
);
```

```
INSERT INTO customers_Table (customer_id, name, email)
```

```
VALUES (1, 'Pallavi Tiwari', 'Pallo@gmail.com');
```

```
INSERT INTO customers_Table (customer_id, name, email)
```

```
VALUES (2, 'Muskan Sharma', 'Muskan@gmail.com');
```

```
INSERT INTO customers_Table (customer_id, name, email)
VALUES (3, 'Raashi Batla', 'Raashi@gmail.com');
```

Customers_Table

customer_id	name	email
1	Pallavi Tiwari	Pallo@gmail.com
2	Muskan Sharma	Muskan@gmail.com
3	Raashi Batla	Raashi@gmail.com

```
CREATE TABLE orders_Table (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

```
INSERT INTO orders_Table(order_id, customer_id, order_date, total_amount)
VALUES (101, 1, '2023-01-05', 100.00);
```

```
INSERT INTO orders_Table (order_id, customer_id, order_date, total_amount)
VALUES (102, 1, '2023-02-12', 50.00);
```

```
INSERT INTO orders_Table (order_id, customer_id, order_date, total_amount)
VALUES (103, 2, '2023-02-18', 200.00);
```

```
INSERT INTO orders_Table (order_id, customer_id, order_date, total_amount)
VALUES (104, 3, '2023-01-25', 75.00);
```

Orders_Table

order_id	customer_id	order_date	total_amount
101	1	2023-01-05	100
102	1	2023-02-12	50
103	2	2023-02-18	200
104	3	2023-01-25	75

In this example, the table name `customer_Table` has a primary key column called "customer_id", a unique identifier for each customer. The `orders_Table` has a foreign key column called "customer_id", which references the `customer_id` column in the `customers_Table`. This establishes a one-to-many relationship between customers and orders: each can have multiple orders, but each order is associated with a single customer.

To retrieve a list of all customers and their associated orders, you could use a SQL statement like this:

```
SELECT customers_Table.customer_id, customers_Table.name, orders_Table.order_id,
       orders_Table.order_date, orders_Table.total_amount
FROM customers_Table
LEFT JOIN orders_Table ON customers_Table.customer_id = orders_Table.customer_id;
```

This statement uses a LEFT JOIN operation to include all records from the `customers_Table`, and any matching records from the `orders_Table`. The ON clause specifies the join condition, which matches records in the `orders` table with the corresponding customer records based on the `customer_id` column.

Output:

customer_id	name	order_id	order_date	total_amount
1	Pallavi Tiwari	101	2023-01-05	100
1	Pallavi Tiwari	102	2023-02-12	50
2	Muskan Sharma	103	2023-02-18	200
3	Raashi Batla	104	2023-01-25	75

As you can see, this query combines the data from both tables, showing the customer and order information together in a single result set.

20. What is a transaction in SQL, and why is it important?

In SQL, a transaction is a logical unit of work consisting of one or more SQL statements executed as a single atomic operation. The primary purpose of a transaction is to ensure that a group of SQL statements is executed as a single, consistent, and reliable unit. When a transaction is initiated, a set of SQL statements are executed, and the changes made to the database are temporarily stored in a buffer area called a transaction log. If all the SQL statements in the transaction are executed successfully, the changes made to the database are committed, which means they are made permanent.

- If any error occurs during the transaction, all changes made to the database are rolled back, and the database is restored to its original state before the transaction starts.
- The importance of transactions in SQL is maintaining the consistency, reliability, and integrity of the data in the database. Transactions help to ensure that the database is always in a valid state and that data is not lost or corrupted due to errors or system failures.
- Transactions also provide a mechanism for concurrent access to the database, allowing multiple users to access the database simultaneously without interfering with each other's work.
- Transactions form the backbone of many business-critical applications and systems by ensuring that database changes are processed reliably and consistently.

21. What is the difference between CHAR and VARCHAR in SQL?

CHAR is a fixed-length data type in SQL, meaning it will always occupy the exact number of characters specified, even if the actual value is shorter. Padding with spaces is done to meet the required length. On the other hand, VARCHAR is a variable-length data type. It stores only the number of characters that are used, without extra padding, making it more efficient for storing variable-length strings. However, CHAR can be faster for fixed-length data as there is no length checking involved.

22. What is the use of the GROUP BY clause in SQL?

The GROUP BY clause in SQL is used to group rows that share the same values in specified columns into summary rows. It is typically used with aggregate functions like COUNT(), SUM(), AVG(), etc., to perform operations on these grouped data. For instance, you can use GROUP BY to find the total sales for each product in a sales table. It helps in producing summary reports and is commonly used for analytical queries.

23. What is the default sorting order in SQL?

The default sorting order in SQL is ascending. When you run a query with the ORDER BY clause, it automatically sorts the results in ascending order unless explicitly specified as DESC (for descending order). For example, in a list of numbers or dates, sorting in ascending order will arrange them from smallest to largest or from oldest to newest, respectively. Ascending sorting can be done by simply using the ORDER BY clause without specifying ASC.

24. What is the difference between UNION and UNION ALL in SQL?

UNION and UNION ALL are both used to combine the result sets of two or more SELECT statements. The key difference is that UNION removes duplicate rows, ensuring that each row in the final result set is unique. In contrast, UNION ALL includes all rows from each query, including duplicates. While UNION is helpful when you need to eliminate duplicates, UNION ALL is faster since it does not perform the duplicate check and simply appends the results.

25. How do you use the DISTINCT keyword in SQL?

The DISTINCT keyword is used in SQL to remove duplicate values from the result set of a SELECT query. It ensures that only unique records are returned. For instance, when retrieving data from a column with repeating values, using DISTINCT will return each unique value only once. This is useful when you want to know distinct categories, items, or

identifiers in a table. For example, SELECT DISTINCT department FROM employees; will give a list of unique departments.

SQL Interview Questions for Intermediate

26. What is ETL in SQL?

ETL (Extract, Transform, Load) is a common process used in data warehousing and business intelligence to move data from various sources into a data warehouse or database. The process involves three steps:

- **Extract:** In this step, data is extracted from various sources such as databases, files, or web services. This may involve using SQL queries to extract data from databases, APIs, or web scraping tools to extract data from web services or files.
- **Transform:** Once the data has been extracted, it is transformed or cleaned to make it suitable for storage and analysis. This may involve applying filters, aggregating data, or converting data types. SQL is commonly used to transform data as part of the ETL process.
- **Load:** The final step is to load the transformed data into a data warehouse or database. This may involve loading data into tables, creating indexes, or performing other database operations.

The ETL process is critical for data integration, as it allows organizations to collect data from various sources, transform it into a consistent format, and store it in a central location for analysis and reporting. ETL tools such as Microsoft SQL Server Integration Services (SSIS) or Talend can automate much of the ETL process and provide a visual interface for designing and managing data flows.

27. What are the types of SQL JOINS?

In SQL, there are four types of JOINs:

- **INNER JOIN:** It returns only the matching rows between two tables. It combines rows from two or more tables where the values in the common columns match.

- **LEFT JOIN:** It returns all the left and the matching rows from the right table. The result will contain null values for the right table's columns if there are no matches.
- **RIGHT JOIN:** It returns all the rows from the right table and the matching ones from the left table. The result will contain null values for the left table's columns if there are no matches.
- **FULL OUTER JOIN:** It returns all the rows from both tables, including unmatched ones. If there are no matches, the result will contain null values for the table's columns that don't have a matching row.

There is also a CROSS JOIN, which returns the Cartesian product of the two tables, which combines every row from the first table with all rows from the second table. However, unlike the other JOIN types, it doesn't use a join condition to match the rows between the tables.

28. What are Aggregate and Scalar functions?

In database management systems, there are two main types of functions used to manipulate data: **aggregate functions** and **scalar functions**.

Aggregate functions perform calculations on a set of values and return a single value representing a summary of that set. These functions are typically used with a GROUP BY clause to group data by one or more columns and then perform calculations on each group. Examples of aggregate functions include SUM, AVG, COUNT, MAX, and MIN.

Scalar functions, on the other hand, operate on a single value and also return a single value. They can manipulate data in various ways, such as performing string operations, date calculations, or mathematical computations. Examples of scalar functions include CONCAT (to concatenate two strings), DATEADD (to add a specified amount of time to date), and ABS (to return the absolute value of a number).

Aggregate functions are used to perform calculations on data sets and return a single value representing some summary of that set. In contrast, scalar functions operate on a single

value and return a single value. Both functions are commonly used in database management systems to manipulate and analyze data.

29. List different Types of Index in SQL?

In SQL, different indexes can be created to improve query performance. Here are some of the most common types of indexes:

- **Clustered Index:** A clustered index in SQL organizes and stores the data rows in a table based on the values of one or more columns. This index determines the physical order of the data within the table, making it highly efficient for range queries and sorting operations. By sorting and storing the data in the table based on the values of the clustered index, queries that filter or sort by those columns can be performed faster since they can utilize the physical order of the data.
- **Non-Clustered Index:** A non-clustered index creates a separate structure that stores a copy of the indexed columns and a pointer to the corresponding data row in the table. It allows for faster retrieval of specific rows or ranges of rows but can be less efficient than a clustered index for sorting operations.
- **Unique Index:** A unique index enforces the constraint that the values in the indexed column(s) must be unique across all rows in the table. Depending on the table's primary key, it can be either a clustered or non-clustered index.
- **Composite Index:** A composite index is an index that is created on two or more columns in a table. It can improve the performance of queries that filter on multiple columns, allowing more efficient sorting and matching of the indexed values.
- **Full-Text Index:** A full-text index searches for text-based data in a table, such as articles, documents, or web pages. It allows for fast and efficient searching of large amounts of text using algorithms that analyze the data's words, phrases, and context.
- **Spatial Index:** A spatial index is used to optimize the querying of geographic or location-based data in a table, such as maps, GPS coordinates, or boundaries. It uses specialized data structures and algorithms to store and search for spatial data.

efficiently.

30. What is the difference between a clustered and a non-clustered index in SQL?

Features	Clustered Index	Non-Clustered Index
Definition	Determines the physical order of the data.	A separate structure that contains the index key and a pointer to the data.
Implementation	Only one clustered index per table.	Multiple non-clustered indexes per table are possible.
Data Retrieval	Faster data retrieval for large data sets.	Slower data retrieval for large data sets.
Storage	Contains the actual data.	Does not contain the actual data.
Key	Determines the order of data on the table.	It helps in searching for the data.
Unique Index	Clustered indexes are unique by default.	Non-clustered indexes can be unique or non-unique.
Data Modification	This may cause more overhead for updates.	It May cause less overhead for updates.
Tables with Clustered Index	It may require defragmentation.	Does not require defragmentation.

31. What are the different types of normalizations?

In database design, several types of normalization are used to reduce data redundancy, improve data integrity, and ensure efficient data retrieval. The different types of normalizations are

- **First Normal Form (1NF):** This normalization ensures that the data in each table's column is atomic, meaning that it cannot be further broken down into smaller pieces. It also eliminates duplicate rows from the table.
- **Second Normal Form (2NF):** This normalization eliminates partial dependencies by ensuring that each non-key column in a table is dependent on the entire primary key rather than on its part of it.
- **Third Normal Form (3NF):** This normalization eliminates transitive dependencies by ensuring that each non-key column in a table is dependent only on the primary key and not on any other non-key columns.
- **Boyce-Codd Normal Form (BCNF):** This normalization is an extension of 3NF and ensures that each determinant in a table is a candidate key.
- **Fourth Normal Form (4NF):** This normalization eliminates multivalued dependencies by ensuring that each non-key column in a table is dependent on the entire primary key and not on any subsets.
- **Fifth Normal Form (5NF):** This normalization is known as the Project-Join Normal Form and ensures that each table in a database has a single theme or topic.

The normalization process is iterative, and it may be necessary to apply multiple normalizations to achieve the desired data organization and efficiency level.

32. Explain Boyce-Codd Normal Form (BCNF).

Boyce-Codd Normal Form (BCNF) is a normalization technique used in database design to eliminate redundancy and improve data integrity. In BCNF, each determinant (i.e., attribute or set of attributes that uniquely determine another attribute) in a table must be a candidate key (i.e., a unique identifier for each row).

In simpler terms, BCNF ensures that each attribute in a table depends on the entire primary key rather than on just a part of it. This helps to reduce data redundancy and improve data

integrity by ensuring that each piece of data in the table is only stored once and can be accessed efficiently.

BCNF is an extension of the Third Normal Form (3NF) and is helpful in situations where 3NF is insufficient to eliminate all forms of redundancy in a table.

However, it is essential to note that achieving BCNF can sometimes result in a higher number of tables and more complex relationships between them. Hence, it is important to balance the benefits of normalization with the practical considerations of database design.

33. What is the difference between a join and a subquery in SQL?

Here's a table format to help illustrate the differences between a join and a subquery in SQL:

Features	Join	Subquery
Syntax	SELECT ... FROM table1 JOIN table2 ...	SELECT ... FROM table1 WHERE condition IN (...)
Purpose	Combine columns from multiple tables.	Retrieve data to use as a condition in the query.
Usage	When querying data from multiple tables.	When querying data based on a condition.
Performance	Typically faster than a subquery.	It can be slower than a join for large datasets.
Result set	Returns columns from multiple tables.	Returns a result set for a single table.
Complexity	More complex syntax and usage.	Less complex syntax and usage
Flexibility	It can be used with various types of joins.	It can be used with various types of subqueries.

Features	Join	Subquery
Code readability	It can be less readable for complex joins	It can be easier to read for simple queries

34. What are some standard clauses used with SELECT queries in SQL?

In SQL, the SELECT statement retrieves data from a database. Along with the introductory SELECT statement, you can use clauses to specify additional details about how the data should be retrieved. Here are some standard clauses used with the SELECT statement example:

Here, we consider a table name List_1 and perform the following SELECT queries.

```
CREATE TABLE `List_1` (
    order_id int(11) NOT NULL,
    customer_id int(11) NOT NULL,
    order_date date NOT NULL,
    total_amount decimal(10,2) NOT NULL,
    PRIMARY KEY (`order_id`)
);
```

```
INSERT INTO `List_1` (`order_id`, `customer_id`, `order_date`, `total_amount`) VALUES
(1, 101, '2022-01-01', 50.00),
(2, 102, '2022-01-02', 100.00),
(3, 103, '2022-01-03', 75.00),
(4, 102, '2022-01-04', 25.00),
(5, 101, '2022-01-05', 80.00);
```

List_1

order_id	customer_id	order_date	total_amount
1	101	2023-02-01	50
2	102	2023-02-02	100
3	103	2023-02-03	75
4	102	2023-02-04	25
5	101	2023-02-05	80

- **WHERE:** This clause filters data based on a specific condition. For example, you can use WHERE to retrieve all records where a particular column is equal to a specific value.

```
SELECT * FROM List_1
```

```
WHERE customer_id = 101;
```

Output:

order_id	customer_id	order_date	total_amount
1	101	2023-02-01	50
5	101	2023-02-05	80

- **ORDER BY:** This clause is used to sort the results in descending or ascending order based on one or more columns. For example, you can use ORDER BY to sort a list of customers by their last name.

```
SELECT * FROM List_1
```

```
ORDER BY total_amount DESC;
```

Output:

order_id	customer_id	order_date	total_amount
2	102	2023-02-02	100
5	101	2023-02-05	80
3	103	2023-02-03	75
1	101	2023-02-01	50
4	102	2023-02-04	25

- **GROUP BY:** This clause groups the results by one or more columns. For example, you can use GROUP BY to group a list of sales by month or by product.

```
SELECT customer_id, COUNT(*) as order_count
```

```
FROM List_1
```

```
GROUP BY customer_id;
```

Output:

customer_id	order_count
101	2
102	2
103	1

- **HAVING:** It filters the groups created by the GROUP BY clause based on a specific condition. For example, you can use HAVING to retrieve only those groups where the total sales are greater than a certain amount.

```
SELECT customer_id, AVG(total_amount) as avg_total
```

```
FROM List_1
```

```
GROUP BY customer_id
```

```
HAVING AVG(total_amount) > 60;
```

Output:

customer_id	avg_total
101	65
102	62.5
103	75

- **LIMIT:** This clause limits the number of rows returned by the query. For example, you can use LIMIT to retrieve the top 10 customers based on their sales.

```
SELECT * FROM List_1
LIMIT 3;
```

Output:

order_id	customer_id	order_date	total_amount
1	101	2023-02-01	50
2	102	2023-02-02	100
3	103	2023-02-03	75

35. How to get unique records from the table without using distinct keywords.

To get unique records from a table without using the DISTINCT keyword in SQL, you can use the GROUP BY clause with aggregate functions like COUNT, SUM, or AVG.

Create a Table Sales

```
CREATE TABLE sales (
    product VARCHAR(50),
    quantity INT
);
```

```
INSERT INTO sales (product, quantity) VALUES ('apple', 20);
```

```
INSERT INTO sales (product, quantity) VALUES ('orange', 15);
```

```
INSERT INTO sales (product, quantity) VALUES ('apple', 30);
```

```
INSERT INTO sales (product, quantity) VALUES ('banana', 35);
```

Sales

product	quantity
apple	20
orange	15
apple	30
banana	35

To get unique products from the sales table, you can group the rows by the product column and use the COUNT aggregate function to count the number of occurrences of each product:

```
SELECT product
```

```
FROM sales
```

```
GROUP BY product;
```

Output:

product
apple
banana
orange

If you want to include additional columns in the output, you can also use aggregate functions for those columns. For example, to get the sum of quantities sold for each product, you can use the SUM aggregate function:

```
SELECT product, SUM(quantity) as total_quantity
FROM sales
GROUP BY product;
```

Output:

product	total_quantity
apple	50
banana	35
orange	15

By grouping the rows by the product column and using the aggregate functions, you can effectively get unique records from the table without using the DISTINCT keyword.

36. Display the monthly Salary of Employees given annual salary.

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

```
INSERT INTO employees (id, name, annual_salary)
VALUES (1, 'Muskan', 60000);
```

```
INSERT INTO employees (id, name, annual_salary)
VALUES (2, 'Pallavi', 75000);
```

```
INSERT INTO employees (id, name, annual_salary)
```

```
VALUES (3, 'Raashi', 90000);
```

Employees

id	name	annual_salary
1	Muskan	60000
2	Pallavi	75000
3	Raashi	90000

```
SELECT id, name, annual_salary / 12 as monthly_salary
```

```
FROM employees;
```

Output:

id	name	monthly_salary
1	Muskan	5000
2	Pallavi	6250
3	Raashi	7500

37. Distinguish between nested subquery, correlated subquery, and join operation.

For distinguishing between these three, we have taken some example through which it would be better for you to understand the terms.

```
CREATE TABLE customerss (
```

```
customer_id INT,  
customer_name VARCHAR(50),  
customer_location VARCHAR(50)
```

);

INSERT INTO customerss (customer_id, customer_name, customer_location)

VALUES

(1, 'Shila', 'New York'),

(2, 'Palak', 'India'),

(3, 'Rajesh', 'New York');

CREATE TABLE orderss (

order_id INT,

customer_id INT,

total_amount INT

);

INSERT INTO orderss (order_id, customer_id, total_amount)

VALUES

(1, 1, 100),

(2, 1, 150),

(3, 2, 75),

(4, 3, 200),

(5, 3, 175);

order_id	customer_id	total_amount
1	1	100
2	1	150
3	2	75
4	3	200
5	3	175

Nested Subquery:

A nested subquery is a query that is embedded within another query. The inner query is performed first, and the outer query uses its results. The inner query is enclosed within parentheses and is usually placed in the WHERE or HAVING clause of the outer query. A nested subquery returns a single value or a list of values.

Example:

```
SELECT order_id, customer_id, total_amount
FROM orders
WHERE customer_id IN (
    SELECT customer_id
    FROM customers
    WHERE customer_location = 'New York'
);
```

Output:

order_id	customer_id	total_amount
1	1	100
2	1	150
4	3	200
5	3	175

Correlated Subquery:

It uses a value from the outer query. The inner query is performed once for each row of the outer query, and the result depends on the outer query's current row. A correlated subquery filters or joins data from two or more tables.

Example:

```
SELECT customer_name, (
    SELECT SUM(total_amount)
    FROM orderss
    WHERE orderss.customer_id = customerss.customer_id
) AS total_spent
FROM customerss;
```

Output:

customer_name	total_spent
Shila	250
Palak	75
Rajesh	375

Join Operation

When you have information about a single object or entity spread across multiple tables, you can use a join operation to combine this information into a single table. Join operations work by matching the data in a specific column between two or more tables and then combining the rows from these tables into a new table. Different types of join operations determine how data is matched and combined. You can retrieve a piece of complete information about an object or entity using join operations from multiple tables.

Example:

```
CREATE TABLE employees (
    employee_id INT,
```

```
employee_name VARCHAR(50),  
department_id INT  
);
```

```
INSERT INTO employees (employee_id, employee_name, department_id)
```

```
VALUES
```

```
(1, 'Juhi', 1),  
(2, 'Ruhi', 2),  
(3, 'Bharat', 2);
```

```
CREATE TABLE departments (
```

```
department_id INT,  
department_name VARCHAR(50)
```

```
);
```

```
INSERT INTO departments (department_id, department_name)
```

```
VALUES
```

```
(1, 'Sales'),  
(2, 'Marketing');
```

Departments

department_id	department_name
1	Sales
2	Marketing

Employees

employee_id	employee_name	department_id
1	Juhi	1
2	Ruhi	2
3	Bharat	2

```

SELECT employees.employee_name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
    
```

Output:

employee_name	department_name
Juhi	Sales
Ruhi	Marketing
Bharat	Marketing

38. What is a Non-Equi Join?

A non-equi join is a type of join operation in SQL where the join condition is based on a comparison operator other than equality, such as '>' (greater than), '>=' (greater than or equal to), '<' (less than), '<=' (less than or equal to), or '<>' (not equal to).

Example:

```
CREATE TABLE table1 (
```

```
    id INT,
```

```
    value INT
```

```
);
```

```
INSERT INTO table1 (id, value) VALUES (1, 10);
```

```
INSERT INTO table1 (id, value) VALUES (2, 20);
```

```
INSERT INTO table1 (id, value) VALUES (3, 30);
```

```
CREATE TABLE table2 (
```

```
    id INT,
```

```
    value INT
```

```
);
```

```
INSERT INTO table2 (id, value) VALUES (1, 15);
```

```
INSERT INTO table2 (id, value) VALUES (2, 25);
```

```
INSERT INTO table2 (id, value) VALUES (3, 35);
```

Table1

id	value
1	10
2	20
3	30

Table2

id	value
1	15
2	25
3	35

```

SELECT *
FROM table1
JOIN table2
ON table1.value > table2.value;

```

Output:

id	value	id	value
2	20	1	15
3	30	1	15
3	30	2	25

As you can see, this query returns all possible combinations of rows where the value in table1 is greater than the corresponding value in table2.

39. What are OLAP and OLTP?

OLAP(Online analytical processing) and **OLTP(Online Transaction Processing)** are two different types of database systems that are used for different purposes in the data processing. OLTP systems are mainly used for recording and processing transactions in real-time. These transactions are usually brief and involve updating, inserting, or deleting data in the database. OLTP systems are built to handle a high volume of concurrent transactions while ensuring data consistency, accuracy, and availability.

In contrast, OLAP systems are used for analytical processing, which involves complex queries and data aggregation for decision-making and business intelligence. OLAP systems typically store a large amount of historical data and allow for sophisticated data analysis, such as trend analysis, forecasting, and data mining. OLAP systems are designed for quick query performance and data retrieval, even when handling large amounts of data.

To sum up, OLTP systems are primarily used for real-time transaction processing, while OLAP systems are used for complex analysis of historical data to support decision-making and business intelligence.

40. What is a Self Join in SQL?

A self-join in SQL occurs when a table is joined with itself. This is typically used when a relationship exists within the same table and you need to compare rows within that table. For instance, if you have an employee table where each employee has a manager, you can use a self-join to find all employees with the same manager.

A self-join uses table aliases to distinguish between the original table and the copy of the table:

```
SELECT A.employee_name, B.employee_name AS manager_name
```

```
FROM employees A
```

```
JOIN employees B ON A.manager_id = B.employee_id;
```

Here, the table employees is joined with itself, and we can see which employee is managed by whom. The aliases A and B help identify the same table being used in different contexts.

41. Explain the use of the COALESCE function in SQL.

The **COALESCE** function in SQL is used to return the first non-null value in a list of expressions. It evaluates each expression one by one and returns the first expression that is not NULL. If all expressions evaluate to NULL, the COALESCE function returns NULL.

The function is particularly useful when dealing with nullable columns in databases where you want to substitute a default value if a NULL is encountered. For instance, you may want to return a default message if a user's name is not available:

```
SELECT COALESCE(first_name, 'Unknown') AS display_name  
FROM users;
```

In this example, if the first_name is NULL, the query will return 'Unknown'. The COALESCE function can accept multiple arguments, making it more versatile than ISNULL, which can only evaluate two arguments.

42. How do you delete duplicate rows in SQL?

To delete duplicate rows in SQL, you can use the **ROW_NUMBER()** function in a Common Table Expression (CTE) or subquery to assign unique row numbers to rows with duplicate values. Then, you can delete rows that have duplicate row numbers, keeping only one instance.

Here's a step-by-step approach using a CTE in SQL Server:

```
WITH CTE AS (  
    SELECT column_name,  
           ROW_NUMBER() OVER (PARTITION BY column_name ORDER BY column_name) AS  
           row_num  
    FROM table_name  
)  
DELETE FROM CTE WHERE row_num > 1;
```

Explanation:

- The ROW_NUMBER() function assigns a unique number to each row within a partition (in this case, for each unique column_name).

- The PARTITION BY clause specifies the column that contains duplicate values, and the ORDER BY clause determines how the rows should be numbered within each partition.
- Rows where row_num > 1 are considered duplicates and are deleted.

In MySQL, you can achieve the same by using a DELETE JOIN:

```
DELETE t1 FROM table_name t1
JOIN table_name t2
ON t1.id > t2.id AND t1.column_name = t2.column_name;
```

This approach deletes rows from t1 where the value of column_name matches another row in t2 but with a larger id, ensuring only one unique row is left.

43. What is a CTE (Common Table Expression) in SQL?

A **Common Table Expression (CTE)** is a temporary result set in SQL that is defined using the WITH keyword and can be referenced within a SELECT, INSERT, UPDATE, or DELETE query. CTEs make queries easier to read and maintain, especially when working with complex joins, recursive queries, or hierarchical data.

A CTE is particularly useful for breaking down complicated queries into simpler steps or reusing a common subquery in multiple parts of the main query. For example:

```
WITH CTE AS (
  SELECT employee_id, manager_id, salary
  FROM employees
  WHERE salary > 50000
)
SELECT * FROM CTE WHERE manager_id IS NOT NULL;
```

Here, the CTE named CTE filters employees who earn more than \$50,000. This result set is then queried in the main SELECT statement.

CTEs can also be recursive, meaning that the CTE can call itself to handle hierarchical data structures like organizational charts or tree-like data models.

44. Explain how to perform pagination in SQL.

Answer:

Pagination in SQL refers to the process of dividing large sets of data into smaller chunks or pages to improve performance and provide easier data navigation. Pagination is commonly used in web applications to display data in a user-friendly way across multiple pages.

In SQL, you can perform pagination using the LIMIT and OFFSET clauses (in MySQL and PostgreSQL), or with ROW_NUMBER() in SQL Server.

MySQL/PostgreSQL Example:

```
SELECT * FROM table_name  
ORDER BY column_name  
LIMIT 10 OFFSET 20;
```

Explanation:

- LIMIT 10 specifies the number of rows to return (10 rows per page).
- OFFSET 20 specifies the starting point for fetching records (skips the first 20 rows, i.e., starts from the 21st row).

SQL Server Example:

```
SELECT * FROM (  
    SELECT *, ROW_NUMBER() OVER (ORDER BY column_name) AS row_num  
    FROM table_name  
) AS T  
WHERE row_num BETWEEN 21 AND 30;
```

Explanation:

- ROW_NUMBER() assigns a unique row number to each row based on the specified ordering.

- The outer query selects rows where row_num falls within the specified range (in this case, rows 21 to 30).

Pagination is essential for handling large datasets and improving query performance, especially in applications where users need to navigate through many records.

45. How do you handle NULL values in SQL?

Handling NULL values in SQL is crucial because NULL represents missing, undefined, or unknown data. Operations involving NULL values can lead to unexpected results if not handled properly.

Here are a few common methods to handle NULL values in SQL:

IS NULL / IS NOT NULL: You can use the IS NULL or IS NOT NULL operators to filter records that have NULL values.

```
SELECT * FROM employees WHERE department IS NULL;
```

COALESCE Function: The COALESCE() function returns the first non-NULL value in a list of expressions.

```
SELECT COALESCE(phone_number, 'Not Available') FROM contacts;
```

IFNULL() or ISNULL(): These functions can be used to replace NULL values with a specified value.

```
SELECT ISNULL(salary, 0) FROM employees;
```

NULL-safe Comparisons: Comparing NULL values with the equality (=) operator will always return FALSE. Use IS NULL or IS NOT NULL to safely check for NULL values in conditions.

Handling NULL values ensures that your queries are more robust and return meaningful results, especially when performing calculations or aggregations.

46. What is the difference between IN and EXISTS in SQL?

Both IN and EXISTS are used in SQL to filter data based on the results of a subquery, but they differ in terms of execution and performance.

IN is used to compare a column to a set of values or the result of a subquery, and it returns TRUE if the column value matches any value in the list.

```
SELECT * FROM employees WHERE department_id IN (SELECT id FROM departments);
```

EXISTS is used to check whether a subquery returns any rows. It returns TRUE if the subquery returns at least one row.

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

Differences:

- **Performance:** EXISTS is often faster than IN for large datasets because EXISTS stops executing as soon as it finds a match, whereas IN checks every value in the subquery result.
- **Data Handling:** IN compares values, whereas EXISTS checks for the existence of rows in the subquery.

Use EXISTS when the subquery involves checking for the existence of data, and use IN when dealing with a small set of specific values.

47. How can you improve the performance of SQL queries?

Improving the performance of SQL queries is critical for optimizing database operations. Here are several techniques to enhance query performance:

Use Indexes: Indexing key columns speeds up query execution by reducing the amount of data the database engine needs to scan. However, avoid over-indexing as it can slow down insert, update, and delete operations.

Optimize Joins: Use the most efficient join type for your query (INNER JOIN, LEFT JOIN, etc.). Ensure that join conditions are indexed to avoid full table scans.

Avoid SELECT *: Instead of selecting all columns, specify only the columns you need. This reduces the amount of data transferred and processed by the database.

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

Use WHERE Clauses Effectively: Ensure that filters in WHERE clauses use indexed columns.

Avoid applying functions directly to columns in the WHERE clause as this can prevent the use of indexes.

```
SELECT * FROM orders WHERE order_date >= '2023-01-01';
```

Limit Results with LIMIT or TOP: When querying large tables, use LIMIT or TOP to fetch only the required number of rows, especially in paginated queries.

Use Query Caching: In databases that support caching, enable query caching for frequently executed queries to avoid running the same query multiple times.

Analyze Execution Plans: Use the EXPLAIN or EXPLAIN ANALYZE command to review the query execution plan and identify performance bottlenecks.

By applying these techniques, you can significantly reduce query execution time and improve overall database performance.

48. What is the difference between RANK() and DENSE_RANK() in SQL?

Both RANK() and DENSE_RANK() are window functions in SQL that assign ranks to rows within a partition based on a specified ordering. However, they handle ties (rows with the same rank) differently:

RANK():

Assigns the same rank to tied rows, but the rank of the next row is incremented by the number of tied rows. In other words, there can be gaps in the ranks.

Example:

Row 1: RANK = 1

Row 2: RANK = 2

Row 3: RANK = 2 (Tie)

Row 4: RANK = 4

DENSE_RANK():

Assigns the same rank to tied rows but ensures the next rank is consecutive (no gaps).

Example:

Row 1: DENSE_RANK = 1

Row 2: DENSE_RANK = 2

Row 3: DENSE_RANK = 2 (Tie)

Row 4: DENSE_RANK = 3

Row 1: RANK = 1

Row 2: RANK = 2

Row 3: RANK = 2 (Tie)

Row 4: RANK = 4

Both functions are useful when ranking rows based on some criteria, but DENSE_RANK() is typically used when you want continuous ranking without gaps.

49. What is a View in SQL, and why is it used?

A **View** in SQL is a virtual table that is the result of a pre-defined SQL query. It doesn't store data physically but provides a way to present data from one or more tables in a structured format. Views are primarily used to simplify complex queries, enforce security by restricting access to specific data, and provide a layer of abstraction to hide the complexity of the underlying table structures.

For example, you can create a view to display only the relevant columns and rows from a table:

```
CREATE VIEW employee_view AS
SELECT employee_name, department_name
FROM employees
JOIN departments ON employees.department_id = departments.id;
```

Benefits of using views:

- **Simplified Querying:** A view allows users to query complex joins or aggregations as if they were simple tables.

- **Security:** Views can restrict access to sensitive columns or rows, ensuring users see only what is necessary.
- **Data Abstraction:** Views hide the underlying table structure and any changes to it, simplifying data presentation.

50. What are Aggregate Functions in SQL? Provide examples.

Aggregate functions in SQL perform a calculation on a set of values and return a single value. They are often used with GROUP BY to summarize data. Common aggregate functions include:

COUNT(): Returns the number of rows in a result set.

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

SUM(): Returns the sum of numeric values in a column.

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

AVG(): Returns the average of numeric values in a column.

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

MAX(): Returns the maximum value in a column.

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

MIN(): Returns the minimum value in a column.

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

Aggregate functions are essential for reporting and analyzing data, such as calculating totals, averages, and counts across large datasets. They can be combined with GROUP BY to calculate aggregated values for each group of rows.

SQL Interview Questions for Experienced

51. What is a CTE (Common Table Expression) in SQL?

A **CTE (Common Table Expression)** is a temporary result set that can be referred to within a SELECT, INSERT, UPDATE, or DELETE statement. It is defined using the WITH clause and provides better readability for complex queries.

Example of CTE:

```
WITH EmployeeCTE AS (
    SELECT employee_id, first_name, salary
    FROM employees
    WHERE salary > 50000
)
SELECT * FROM EmployeeCTE;
```

CTEs improve query structure, making it easier to break down complex operations.

52. How does the MERGE statement work in SQL?

The **MERGE** statement in SQL allows you to perform INSERT, UPDATE, or DELETE operations in a single statement based on conditions. It is typically used for merging records from a source table into a target table.

Example:

```
MERGE INTO target_table AS T
USING source_table AS S
ON T.id = S.id
WHEN MATCHED THEN
    UPDATE SET T.name = S.name
WHEN NOT MATCHED THEN
    INSERT (id, name) VALUES (S.id, S.name);
```

This simplifies operations that would otherwise require multiple separate SQL statements.

53. What are Window Functions in SQL?

Window functions perform calculations across a set of table rows related to the current row. Unlike aggregate functions, they do not group the result into a single row but keep the row structure intact.

Example of ROW_NUMBER() as a window function:

```
SELECT employee_name, salary, ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rank  
FROM employees;
```

Window functions are often used in ranking, aggregating, and calculating running totals without affecting the row-level granularity of the result.

54. What is the difference between ROW_NUMBER(), RANK(), and DENSE_RANK() in SQL?

These are all **ranking window functions**, but they behave differently in handling ties:

- **ROW_NUMBER()**: Assigns a unique number to each row, regardless of ties.
- **RANK()**: Assigns the same rank to tied rows but leaves a gap in the ranking.
- **DENSE_RANK()**: Assigns the same rank to tied rows but does not leave any gaps.

Example:

```
SELECT employee_name, salary,  
ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,  
RANK() OVER (ORDER BY salary DESC) AS rank,  
DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank  
FROM employees;
```

55. Explain Recursive CTE in SQL.

A **Recursive CTE** allows a CTE to reference itself. This is useful for hierarchical or tree-like data structures, such as organizational charts or recursive relationships.

Example:

```
WITH RecursiveCTE (employee_id, manager_id, level) AS (  
    SELECT employee_id, manager_id, 1  
    FROM employees  
    WHERE manager_id IS NULL
```

UNION ALL

```
SELECT e.employee_id, e.manager_id, level + 1
FROM employees e
INNER JOIN RecursiveCTE r ON e.manager_id = r.employee_id
)
SELECT * FROM RecursiveCTE;
```

This CTE recursively retrieves hierarchical data, such as employees reporting to their managers.

56. What is the difference between UNION and UNION ALL?

- **UNION:** Combines the result sets of two or more SELECT statements but removes duplicates.
- **UNION ALL:** Combines the result sets of two or more SELECT statements without removing duplicates.

Example:

```
SELECT name FROM customers
```

```
UNION
```

```
SELECT name FROM suppliers;
```

UNION is slower due to the extra step of removing duplicates, while UNION ALL is faster because it includes all rows.

57. How do you optimize a SQL query with GROUP BY?

To optimize a GROUP BY query:

1. **Index the columns** used in the GROUP BY and WHERE clauses.
2. Use **aggregated columns** selectively.
3. Avoid using functions on columns in the GROUP BY clause.
4. If possible, filter data before using GROUP BY.

Example:

```
SELECT department, AVG(salary)  
FROM employees  
WHERE department IS NOT NULL  
GROUP BY department;
```

Indexes on department improve performance by allowing the query to access pre-sorted data.

58. What is a Cross Join in SQL?

A **Cross Join** returns the Cartesian product of two tables. This means every row from the first table is combined with every row from the second table.

```
SELECT *  
FROM products  
CROSS JOIN categories;
```

If the first table has 3 rows and the second has 4 rows, the result will contain 12 rows (3x4).

59. What is a Partitioned Table in SQL?

A **Partitioned Table** splits a large table into smaller, more manageable pieces while maintaining a logical single table view. It improves query performance and simplifies data management.

```
CREATE TABLE orders (  
    order_id INT,  
    order_date DATE,  
    amount DECIMAL(10, 2)  
)  
PARTITION BY RANGE (YEAR(order_date)) (  
    PARTITION p2019 VALUES LESS THAN (2020),
```

PARTITION p2020 VALUES LESS THAN (2021)

);

Partitioning makes it easier to query large datasets by accessing only relevant partitions.

60. How do you handle NULL values in SQL joins?

To handle NULL values in joins, you can:

- Use LEFT JOIN or RIGHT JOIN to include rows with NULL values from one table.
- Use COALESCE() or ISNULL() to replace NULL values with a default value in the result set.

Example:

```
SELECT e.employee_name, d.department_name  
FROM employees e  
LEFT JOIN departments d ON e.department_id = d.department_id;
```

Here, LEFT JOIN includes employees with NULL department IDs in the result.

61. What is an Index in SQL, and how does it improve performance?

An **Index** is a data structure that improves the speed of data retrieval operations on a table. Indexes are created on one or more columns of a table and allow the database to locate data without scanning the entire table.

Example:

```
CREATE INDEX idx_employee_name ON employees(employee_name);
```

Indexes improve the performance of SELECT queries but may slow down INSERT, UPDATE, and DELETE operations due to the overhead of maintaining the index.

62. What is a Covering Index?

A **Covering Index** is an index that contains all the columns needed to satisfy a query. If a query can be executed using only the index, without accessing the table, it is said to be "covered" by the index.

Example:

```
CREATE INDEX idx_covering ON employees (first_name, last_name, salary);
```

Here, the index covers queries that select only first_name, last_name, and salary from the employees table.

63. What is the difference between HAVING and WHERE in SQL?

- **WHERE:** Filters rows before aggregation. It cannot be used with aggregate functions like COUNT(), SUM(), etc.
- **HAVING:** Filters rows after aggregation. It is typically used with GROUP BY to filter aggregated data.

Example:

```
SELECT department, COUNT(*)
```

```
FROM employees
```

```
GROUP BY department
```

```
HAVING COUNT(*) > 10;
```

HAVING allows you to filter results based on the aggregate COUNT() function, whereas WHERE filters individual rows before the GROUP BY clause.

64. Explain the concept of Transaction Isolation Levels.

Transaction isolation levels determine how and when the changes made by one transaction become visible to other transactions. The SQL standard defines four levels of isolation:

1. **Read Uncommitted:** Lowest isolation level; allows dirty reads (reads uncommitted data).
2. **Read Committed:** Prevents dirty reads, but non-repeatable reads can still occur.
3. **Repeatable Read:** Prevents dirty and non-repeatable reads, but phantom reads can occur.

4. **Serializable:** Highest isolation level; prevents all types of inconsistencies, including phantom reads.

The higher the isolation level, the more consistent the data but at the cost of performance due to locking mechanisms.

65. What is the purpose of a WITH clause in SQL?

The **WITH** clause, also known as **CTE (Common Table Expression)**, is used to define a temporary result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement.

Example:

```
WITH EmployeeCTE AS (
    SELECT employee_id, first_name, salary
    FROM employees
)
SELECT * FROM EmployeeCTE WHERE salary > 50000;
```

The WITH clause improves query readability, especially for complex operations that involve subqueries.

66. What is a Composite Index in SQL?

A **Composite Index** is an index that is created on two or more columns of a table. It improves the performance of queries that filter data based on those multiple columns.

Example:

```
CREATE INDEX idx_emp_name_salary ON employees (first_name, last_name, salary);
```

This index will optimize queries that search or sort based on the combination of first_name, last_name, and salary. However, the order of columns in a composite index matters, as it can affect performance if not used in the right sequence.

67. How can you optimize a SQL query that uses JOIN operations?

To optimize a SQL query with JOIN:

1. **Index the columns** used in the JOIN condition.
2. Use **INNER JOIN** instead of **LEFT JOIN** when you don't need null records.
3. **Minimize the dataset** using the WHERE clause before applying JOIN.
4. Consider **partitioning large tables** to improve the performance of join operations.

Example:

```
SELECT e.first_name, d.department_name  
FROM employees e  
INNER JOIN departments d ON e.department_id = d.department_id  
WHERE e.salary > 50000;
```

Indexes on employee_id and department_id will improve query performance.

68. What is a Trigger in SQL and how is it used?

A **Trigger** is a database object that automatically executes a specified SQL code in response to certain events, such as INSERT, UPDATE, or DELETE. Triggers help enforce data integrity and business rules.

Example:

```
CREATE TRIGGER salary_update_trigger  
AFTER UPDATE ON employees  
FOR EACH ROW  
BEGIN  
IF NEW.salary < OLD.salary THEN  
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Salary cannot be reduced!';  
END IF;
```

END;

This trigger ensures that an employee's salary cannot be reduced during an update.

69. What are Materialized Views, and how do they differ from regular Views?

A **Materialized View** is a database object that stores the result of a query physically. Unlike a regular view, which is a virtual table that is generated dynamically upon query execution, a materialized view stores data to improve performance, especially for complex queries.

Example:

```
CREATE MATERIALIZED VIEW emp_dept_view AS  
SELECT e.first_name, d.department_name  
FROM employees e  
JOIN departments d ON e.department_id = d.department_id;
```

Materialized views are useful for improving performance but require periodic refreshing to keep the data current.

70. What is the difference between EXISTS and IN in SQL?

Both **EXISTS** and **IN** are used to filter data based on subqueries, but they operate differently:

- **EXISTS:** Checks if the subquery returns any rows. It is efficient when the subquery result set is large.
- **IN:** Compares values from the outer query with a list of values returned by the subquery. It performs better when the subquery returns a smaller result set.

Example:

-- Using EXISTS

```
SELECT e.first_name  
FROM employees e  
WHERE EXISTS (SELECT 1 FROM departments d WHERE e.department_id =  
d.department_id);
```

-- Using IN

```
SELECT e.first_name  
FROM employees e  
WHERE e.department_id IN (SELECT department_id FROM departments);
```

71. What is Database Normalization and why is it important?

Database Normalization is the process of organizing a database to reduce redundancy and improve data integrity. It involves dividing large tables into smaller, related tables and defining relationships between them. The key goals of normalization are:

1. Eliminate redundancy.
2. Ensure data consistency.
3. Facilitate easier maintenance.

The main normal forms include:

- **1NF**: Ensures that each column contains atomic values.
- **2NF**: Requires 1NF compliance and removes partial dependencies.
- **3NF**: Requires 2NF compliance and eliminates transitive dependencies.

Normalization optimizes database structure but may introduce performance issues with excessive joins.

72. What are Stored Procedures and why are they useful?

A **Stored Procedure** is a compiled group of SQL statements that can be executed as a single unit. Stored procedures are useful for:

1. **Code reusability**.
2. **Improved performance**, as they are precompiled.
3. **Security** by controlling access to data.

Example:

```
CREATE PROCEDURE UpdateSalary(IN emp_id INT, IN new_salary DECIMAL)
```

BEGIN

```
UPDATE employees SET salary = new_salary WHERE employee_id = emp_id;
```

END;

Stored procedures can accept input parameters and return output values, making them a powerful tool for automating tasks in SQL.

73. Explain the concept of Database Denormalization.

Denormalization is the process of combining tables or adding redundant data to a database to improve query performance. While normalization reduces redundancy, denormalization may reintroduce it to reduce the complexity of joins and speed up read queries.

Example: Instead of keeping employee and department data in separate tables, denormalization might store department information directly within the employee table.

Denormalization trades data redundancy for query performance, particularly in systems where read performance is more critical than storage efficiency.

74. What is the difference between Clustered and Non-Clustered Indexes?

- **Clustered Index:** Determines the physical order of data in a table. A table can only have one clustered index, and the data is stored based on this index.
- **Non-Clustered Index:** Does not affect the physical order of the data. A table can have multiple non-clustered indexes, and these indexes contain pointers to the actual data.

Example:

```
CREATE CLUSTERED INDEX idx_employee_id ON employees(employee_id);
```

```
CREATE NONCLUSTERED INDEX idx_employee_name ON employees(first_name, last_name);
```

Clustered indexes improve the performance of range queries, while non-clustered indexes optimize individual lookups.

75. How do you handle Deadlocks in SQL?

A **deadlock** occurs when two or more transactions hold locks on resources and each is waiting for the other to release the lock, causing a cycle. To handle deadlocks:

1. **Minimize locking time** by keeping transactions short.
2. **Use consistent locking order** in your application.
3. **Set proper isolation levels** to avoid unnecessary locking.

SQL Server automatically detects deadlocks and resolves them by rolling back one of the transactions. You can also manually resolve deadlocks by catching the error and retrying the transaction.

76. What is the purpose of the EXPLAIN command in SQL?

The **EXPLAIN** command is used to analyze and display the execution plan of a SQL query. It shows how the query optimizer plans to execute the query, including details like table scans, index usage, join operations, and cost estimates.

Example:

```
EXPLAIN SELECT * FROM employees WHERE salary > 50000;
```

By reviewing the execution plan, you can identify performance bottlenecks, such as full table scans, and take steps to optimize the query.

77. What are SQL Subqueries, and how are they used?

A **subquery** is a query nested inside another query. Subqueries can be used to return results that are used by the outer query. Subqueries can appear in SELECT, FROM, WHERE, or HAVING clauses.

Example:

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

This query returns employees whose salary is higher than the average salary, which is calculated by the subquery.

78. What are SQL Transactions and their properties?

A **Transaction** in SQL is a sequence of operations performed as a single logical unit. It ensures data integrity through **ACID properties**:

1. **Atomicity:** Ensures all operations in a transaction are completed or none are.
2. **Consistency:** Guarantees that the database remains in a consistent state after the transaction.
3. **Isolation:** Prevents concurrent transactions from interfering with each other.
4. **Durability:** Ensures that once a transaction is committed, it remains so, even in the event of a system failure.

Example:

```
BEGIN TRANSACTION;  
UPDATE employees SET salary = salary * 1.1 WHERE department_id = 1;  
COMMIT;
```

If any part of the transaction fails, you can use ROLLBACK to revert all changes.

79. How do you create and use a Pivot Table in SQL?

A **Pivot Table** in SQL allows you to transform data from rows into columns. This is useful for generating cross-tab reports.

Example:

```
SELECT department,  
       SUM(CASE WHEN gender = 'Male' THEN 1 ELSE 0 END) AS male_count,  
       SUM(CASE WHEN gender = 'Female' THEN 1 ELSE 0 END) AS female_count  
FROM employees  
GROUP BY department;
```

This query counts the number of males and females in each department, pivoting the gender data into separate columns.

80. What is the difference between DELETE and TRUNCATE in SQL?

- **DELETE:** Removes rows from a table one by one, and each row deletion is logged. It can be rolled back using a transaction.
- **TRUNCATE:** Removes all rows from a table quickly by deallocating the data pages. It is faster but cannot be rolled back in most databases.

Example:

```
DELETE FROM employees WHERE employee_id = 1001;
```

```
TRUNCATE TABLE employees;
```

DELETE is more flexible, while TRUNCATE is faster for large datasets.

PostgreSQL Interview Questions

81. What is PostgreSQL?

PostgreSQL is an open-source, object-relational database management system (ORDBMS) that emphasizes extensibility and compliance with SQL standards. It supports both SQL and JSON for relational and non-relational queries. PostgreSQL is widely known for its robustness, performance, and advanced features like complex queries, foreign keys, triggers, updatable views, and transaction integrity.

82. What is the difference between PostgreSQL and MySQL?

- **PostgreSQL** is an object-relational database management system (ORDBMS), whereas **MySQL** is a purely relational database management system (RDBMS).
- PostgreSQL supports advanced features like custom data types, table inheritance, and function overloading, making it highly extensible.
- MySQL has a simpler system architecture and is more focused on read-heavy applications, while PostgreSQL is designed for complex, write-heavy operations.
- PostgreSQL follows SQL standards more strictly than MySQL.

83. How do you create a new database in PostgreSQL?

To create a new database in PostgreSQL, you use the CREATE DATABASE command. For example:

```
CREATE DATABASE testdb;
```

This command will create a new database named testdb. You can connect to this database using the \c testdb command in the PostgreSQL command-line interface.

84. What is a CTE (Common Table Expression) in PostgreSQL?

A **Common Table Expression (CTE)** in PostgreSQL is a temporary result set defined within a SQL query. It is typically used to make queries more readable by breaking them into smaller parts. CTEs are created using the WITH clause. Example:

```
WITH sales_summary AS (
    SELECT department, SUM(sales) AS total_sales
    FROM sales
    GROUP BY department
)
```

```
SELECT * FROM sales_summary WHERE total_sales > 10000;
```

85. How does PostgreSQL handle indexing?

PostgreSQL supports various types of indexes, including **B-tree**, **Hash**, **GIN** (Generalized Inverted Index), and **GIST** (Generalized Search Tree). Indexes in PostgreSQL are automatically used by the query planner to optimize query performance. They help in speeding up the retrieval of rows from a database at the cost of additional storage space and slower inserts/updates. Example of creating an index:

```
CREATE INDEX idx_employee_id ON employees(employee_id);
```

86. What are TOAST tables in PostgreSQL?

TOAST (The Oversized-Attribute Storage Technique) is a mechanism in PostgreSQL for storing large data, such as big text or binary objects. PostgreSQL stores data in tuples, and if a tuple exceeds a certain size limit (usually 8KB), TOAST will automatically compress and store it in an external storage space to avoid performance issues.

87. What are the advantages of using JSONB in PostgreSQL?

JSONB is a binary representation of JSON data in PostgreSQL. It is optimized for efficient storage and faster query performance when working with JSON data. The advantages of JSONB over plain JSON include:

- Faster access and retrieval of JSON data.
- JSONB is stored in a decomposed binary format, allowing efficient indexing.
- Supports operators like @>, <@, and ? for advanced querying of JSON data.

88. How do you perform replication in PostgreSQL?

PostgreSQL supports multiple replication methods, including **Streaming Replication**, **Logical Replication**, and **Synchronous Replication**.

- **Streaming Replication**: Sends real-time changes from the primary server to a standby server.
- **Logical Replication**: Allows selective data replication (row-level) between databases.
- **Synchronous Replication**: Ensures that each transaction is confirmed on both primary and standby servers before committing.

Example of setting up streaming replication:

- Configure the primary server to enable wal_level to replica.
- Set up a standby server and configure the recovery.conf file.

89. How can you perform full-text search in PostgreSQL?

PostgreSQL offers built-in support for **full-text search** through the tsvector and tsquery data types. Full-text search is used to search for words and phrases within text fields, enabling more complex querying options. Example:

```
SELECT * FROM documents
```

```
WHERE to_tsvector('english', content) @@ to_tsquery('PostgreSQL & Database');
```

This query searches for documents containing both "PostgreSQL" and "Database" in their content.

90. What is the VACUUM command in PostgreSQL?

The **VACUUM** command in PostgreSQL reclaims storage occupied by dead tuples that result from UPDATE or DELETE operations. Without periodic vacuuming, PostgreSQL tables would bloat, leading to reduced performance. Example:

```
VACUUM FULL employees;
```

This command reclaims storage and optimizes the performance of the employees table.

SQL MCQs

91. Which of the following SQL statements is used to retrieve data from a database?

- a) SELECT
- b) GET
- c) RETRIEVE
- d) FETCH

Answer: a) SELECT

92. What does the acronym SQL stand for?

- a) Structured Query Language
- b) Simple Query Language
- c) Strong Query Language
- d) Secure Query Language

Answer: a) Structured Query Language

93. Which of the following SQL clauses is used to filter data returned by a query?

- a) WHERE
- b) FILTER
- c) ORDER BY
- d) GROUP BY

Answer: a) WHERE

94. Which of the following is used to create a new table in SQL?

- a) INSERT
- b) CREATE
- c) UPDATE
- d) ALTER

Answer: b) CREATE

95. Which SQL function is used to count the number of rows in a table?

- a) COUNT()
- b) SUM()
- c) AVG()
- d) MAX()

Answer: a) COUNT()

96. What is the correct syntax for an INNER JOIN in SQL?

- a) SELECT * FROM table1 JOIN table2 ON condition;
- b) SELECT * FROM table1 INNER JOIN table2 ON condition;
- c) SELECT * FROM table1 LEFT JOIN table2 ON condition;
- d) SELECT * FROM table1 FULL JOIN table2 ON condition;

Answer: b) SELECT * FROM table1 INNER JOIN table2 ON condition;

97. Which SQL clause is used to sort the result set?

- a) ORDER BY
- b) GROUP BY
- c) HAVING
- d) SELECT

Answer: a) ORDER BY

98. What is a primary key in SQL?

- a) A column that uniquely identifies each row in a table
- b) A column that allows duplicate values
- c) A foreign key in another table
- d) A key that is used to join two tables

Answer: a) A column that uniquely identifies each row in a table

99. Which SQL statement is used to delete data from a table?

- a) DELETE
- b) REMOVE

- c) TRUNCATE
- d) DROP

Answer: a) DELETE

100. In SQL, which of the following commands is used to modify the structure of a table?

- a) UPDATE
- b) ALTER
- c) MODIFY
- d) CHANGE

Answer: b) ALTER