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 c**omposite 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?

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

Example:

SELECT COUNT(*) FROM Orders;

2. 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 load**s 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.