

Scenario Based SQL Interview Questions

1. What does UNION do? What is the difference between UNION and UNION ALL?

`UNION` merges the contents of two structurally-compatible tables into a single combined table. The difference between `UNION` and `UNION ALL` is that `UNION` will omit duplicate records whereas `UNION ALL` will include duplicate records. It is important to note that the performance of `UNION ALL` will typically be better than `UNION`, since `UNION` requires the server to do the additional work of removing any duplicates. So, in cases where it is certain that there will not be any duplicates, or where having duplicates is not a problem, use of `UNION ALL` would be recommended for performance reasons.

2. Important Detail: Behaviour of NOT IN with NULL

When using `NOT IN` and the inner query contains a `NULL` value, the condition evaluates to `FALSE` or `UNKNOWN` for all rows. This is because comparisons involving `NULL` result in `UNKNOWN`, and `NOT IN` fails to exclude anything definitively.

The design of `NOT IN` ensures logical consistency:

- If there is even a single `NULL`, SQL cannot definitively conclude whether or not a value is in the list because `NULL` means "unknown."
- As a result, SQL treats the `NOT IN` condition as unresolvable, returning no rows.

How to Handle This Issue:

```
SELECT * FROM runners WHERE id NOT IN (SELECT winner_id FROM races WHERE winner_id IS NOT NULL);
```

3. If there are 10 records in the Emp table and 5 records in the Dept table, how many rows will be displayed in the result of the following SQL query:

Select * From Emp, Dept

The **Cartesian Join** (or Cross Join) occurs because no explicit `JOIN` condition or `ON` clause is specified in the query. Cartesian joins are the default behavior when no join condition is specified. They are mathematically equivalent to the Cartesian product of two sets. To explore all possible combinations of rows for debugging or verification purposes. Here's why this happens step by step:

- In this query, the tables `Emp` and `Dept` are simply listed with a comma (,), which is the syntax for a **Cartesian join** in SQL.
- A Cartesian join pairs **every row in the first table** (`Emp`) with **every row in the second table** (`Dept`).

4. Write a query to fetch values in table test_a that are and not in test_b without using the NOT keyword?

- To fetch the values in `test_a` that are **not present in test_b** without using the `NOT` keyword, you can use a **LEFT JOIN** combined with a `WHERE` clause that checks for `NULL` in the `test_b` table.

```
SELECT a.* FROM test_a a
```

```
LEFT JOIN test_b b ON a.id = b.id
```

```
WHERE b.id IS NULL;
```

- Alternative Using EXCEPT: If your database supports the EXCEPT operator, you can achieve the same result more succinctly:

```
SELECT id FROM test_a EXCEPT SELECT id FROM test_b;
```

5. Write a SQL query using UNION ALL (not UNION) that uses the WHERE clause to eliminate duplicates.

Why might you want to do this?

SQL Query Using UNION ALL with WHERE to Eliminate Duplicates

Why Use UNION ALL with a WHERE Clause Instead of UNION:

- **Performance:** UNION ALL is faster than UNION because it does not perform the expensive step of deduplication. Deduplication in UNION requires sorting and comparison, which can be costly for large datasets.
- **Control Over Deduplication:** Using the WHERE clause gives precise control over which rows to include, allowing fine-tuned elimination of duplicates based on specific conditions.

6. What is an execution plan? When would you use it? How would you view the execution plan? in SQL

What Is an Execution Plan in SQL: An **execution plan** is a roadmap that shows how a SQL query will be executed by the database engine. Execution plans are essential for understanding query performance and identifying bottlenecks in SQL statements.

It details the steps the database takes to access the data, including:

- The order in which tables are accessed.
- The types of operations performed (e.g., scans, joins, filters, sorts).
- The indexes or keys used.
- The estimated cost and number of rows for each operation

When Would You Use an Execution Plan:

- **Performance Tuning:** To optimize slow queries by identifying inefficiencies, such as full table scans or incorrect use of indexes. Analyse costly operations like large sorts, joins, or excessive disk I/O.
- **Troubleshooting:** To determine why a query is not using an index or returning unexpected results. To debug and refine query logic for better performance.
- **Database Design:** To test the effectiveness of indexes and constraints. To evaluate the impact of schema changes on query performance.
- **Query Optimization:** To compare execution plans of different query versions and choose the most efficient one.

7. List and explain each of the ACID properties that collectively guarantee that database transactions are processed reliably.?

ACID is an acronym that represents the key properties required for reliable database transactions. These properties ensure the integrity, consistency, and reliability of a database system during transactions, even in the presence of failures. Below is a detailed explanation of each property:

- **Atomicity:** Ensures that a transaction is treated as a single, indivisible unit. Either all operations within the transaction are completed successfully, or none are executed. Prevents partial updates to the database in case of system failures or errors.
Example: A bank transfer involves debiting an amount from one account and crediting it to another. If the system fails after debiting but before crediting, atomicity ensures that the debit is rolled back.
- **Consistency:** Guarantees that a transaction will bring the database from one valid state to another, maintaining all predefined rules, constraints, and integrity. Ensures the database remains in a valid state before and after the transaction.

Example: If a database constraint requires that the total balance of two accounts involved in a transfer remains constant, the system enforces this rule throughout the transaction.

- **Isolation:** Ensures that transactions are executed independently of one another, so concurrent transactions do not interfere and lead to inconsistent data. Prevents issues such as dirty reads, non-repeatable reads, and phantom reads.

Example: If two users try to update the same record simultaneously, isolation ensures that one transaction's changes are visible to others only after it is committed.

- **Durability:** Ensures that once a transaction is committed, its changes are permanent, even in the event of a system failure. Guarantees the reliability of committed data.

Example: After transferring money between accounts, the transaction is recorded in the database. Even if the system crashes, the transfer remains in effect upon recovery.

8. What is CTE in details with example. Also Explain me over clause in detail and Rows, Range with example.

A Common Table Expression (CTE) is a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, or DELETE statement. It is easier to read and maintain than subqueries and is particularly useful when dealing with complex queries.

Syntax:

sql

```
WITH cte_name (column1, column2, ...)
AS (
    -- SQL query
)
SELECT * FROM cte_name;
```

Key Features:

- Makes queries more readable by breaking them into smaller, manageable parts.
- Can be recursive (to perform tasks like hierarchical queries).
- Exists only during the execution of the SQL statement.

OVER() Clause: The OVER() clause frames define a window or subset of rows within a result set on which a window function operates. The frame is determined by specifying boundaries within the OVER() clause using ROWS or RANGE. These frames control how the calculation for each row is applied, affecting functions like SUM(), AVG(), ROW_NUMBER(), and more.

Syntax - <window_function>() OVER ([PARTITION BY column] [ORDER BY column])

- **PARTITION BY:** Divides the result set into partitions to apply the function to each partition.
- **ORDER BY:** Defines the order of rows within each partition.
- **Frame Definition:** Specifies the boundaries using:
 - **ROWS:** Defines a physical set of rows.
 - **RANGE:** Defines a logical set of rows based on value ranges.

Frame Definitions:

- **Default Frame:** If no explicit frame is specified, the default depends on the function:
 - Aggregates (SUM, AVG, etc.) default to RANGE UNBOUNDED PRECEDING AND CURRENT ROW.
 - Ranking functions (ROW_NUMBER, RANK, etc.) automatically operate on the entire partition.

- **Explicit Frame:** You can explicitly define the start and end of the frame using:
 - UNBOUNDED PRECEDING: The frame starts at the first row of the partition.
 - UNBOUNDED FOLLOWING: The frame ends at the last row of the partition.
 - CURRENT ROW: Includes only the current row in the calculation.
 - <n> PRECEDING: Includes n rows before the current row.
 - <n> FOLLOWING: Includes n rows after the current row.

Syntax:

sql

```
<window_function>()
OVER (
  [PARTITION BY column]
  [ORDER BY column]
  [ROWS | RANGE BETWEEN frame_start AND frame_end]
)
```

Clause	Description
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	Includes all rows from the start to the current row.
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	Includes all rows from the current row to the end.
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING	Includes one row before and one row after the current.
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	Includes all rows where values fall within this range.
RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	Includes all rows where values fall beyond the current row.
RANGE BETWEEN 10 PRECEDING AND 10 FOLLOWING	Includes rows where values are within ± 10 of the current value.

Recursive cte in detail with example: A Recursive CTE is a powerful feature in SQL that allows you to perform recursive queries—queries where the result of the query depends on the previous iteration. It’s useful for querying hierarchical data, like organizational structures, bill-of-materials, or directory structures.

A recursive CTE is typically divided into two parts:

- **Base Case:** The initial query that provides the first set of results.
- **Recursive Case:** The part of the query that refers to itself to retrieve subsequent results
- **Conditions:**
 - WITH RECURSIVE is used to define a recursive CTE
 - The **Base Case** initializes the query with a set of results
 - The **Recursive Case** joins the result of the previous iteration (cte_name) with the base table to produce additional results.
 - UNION ALL combines the results of the base and recursive queries. We use UNION ALL rather than UNION to avoid unnecessary deduplication.
- **Important Considerations:**
 - **Termination Condition:** Recursive queries must eventually stop. Without a proper termination condition, they can run indefinitely, which will result in a system crash or error.

- **Performance:** Recursive CTEs can be performance-intensive, especially if the recursion depth is large. Indexing the ManagerID column can help improve performance.
- **Recursion Depth Limitation:** Many SQL implementations limit the maximum recursion depth to prevent infinite loops. For example, SQL Server has a default recursion depth limit of 100. You can change this limit with the OPTION (MAXRECURSION n) clause.

Recursive CTE Syntax

sql

```
WITH RECURSIVE cte_name AS (
    -- Base case: Query to return the initial data
    SELECT column1, column2
    FROM table_name
    WHERE condition

    UNION ALL

    -- Recursive case: Query that references the CTE itself
    SELECT t.column1, t.column2
    FROM table_name t
    JOIN cte_name cte ON t.column1 = cte.column1
    WHERE condition
)
SELECT * FROM cte_name;
```

9. All date and time functions in sql with example?

- CURRENT_DATE / CURRENT_TIME / CURRENT_TIMESTAMP -

Example:

sql

```
SELECT CURRENT_DATE AS Today,
       CURRENT_TIME AS CurrentTime,
       CURRENT_TIMESTAMP AS CurrentTimestamp;
```

- DATEPART / EXTRACT – EXTRACT(part FROM date), DATEPART(part, date)

Example:

sql

```
-- MySQL
SELECT EXTRACT(YEAR FROM '2025-01-13') AS Year,
       EXTRACT(MONTH FROM '2025-01-13') AS Month,
       EXTRACT(DAY FROM '2025-01-13') AS Day;

-- SQL Server
SELECT DATEPART(YEAR, '2025-01-13') AS Year,
       DATEPART(MONTH, '2025-01-13') AS Month,
       DATEPART(DAY, '2025-01-13') AS Day;
```

- **DATE_ADD / DATE_SUB** - DATE_ADD(date, INTERVAL value unit) / DATE_SUB(date, INTERVAL value unit)

Example:

sql

```
-- MySQL
SELECT DATE_ADD('2025-01-13', INTERVAL 10 DAY) AS Plus10Days,
       DATE_SUB('2025-01-13', INTERVAL 10 DAY) AS Minus10Days;
```

- **DATEDIFF** - DATEDIFF(date1, date2)

Example:


sql

```
SELECT DATEDIFF('2025-01-13', '2024-12-31') AS DaysDifference;
```

- **TIMESTAMPDIFF** - Return the difference between two timestamps in a specified unit.

Example:

sql

 Copy code

```
SELECT TIMESTAMPDIFF(DAY, '2025-01-01', '2025-01-13') AS DayDifference,
       TIMESTAMPDIFF(HOUR, '2025-01-01 00:00:00', '2025-01-13 12:00:00') AS HourDifference;
```

- **STR_TO_DATE / CONVERT / CAST** - Convert a string to a date or vice versa.

Example:

sql


```
-- MySQL
SELECT STR_TO_DATE('13-01-2025', '%d-%m-%Y') AS ConvertedDate;

-- SQL Server
SELECT CONVERT(DATE, '01/13/2025', 101) AS ConvertedDate;
```

- **FORMAT / TO_CHAR** - Format a date or timestamp to a specific string format.

Example:

sql

 Copy code

```
-- MySQL
SELECT DATE_FORMAT('2025-01-13', '%W, %M %d, %Y') AS FormattedDate;

-- PostgreSQL
SELECT TO_CHAR(TIMESTAMP '2025-01-13 15:30:45', 'FMDay, Month DD, YYYY HH12:MI:SS AM') AS I
```

- **YEAR, MONTH, DAY, HOUR, MINUTE, SECOND** - Extract specific units of time.

Example:

sql

```
SELECT YEAR('2025-01-13') AS Year,
       MONTH('2025-01-13') AS Month,
       DAY('2025-01-13') AS Day,
       HOUR('15:30:45') AS Hour,
       MINUTE('15:30:45') AS Minute,
       SECOND('15:30:45') AS Second;
```

- **DAYNAME / MONTHNAME** - Retrieve the day name or month name from a date.

Example:

sql

```
SELECT DAYNAME('2025-01-13') AS DayName,
       MONTHNAME('2025-01-13') AS MonthName;
```

10. Explain view in sql with details with example. Also, all view types with example.

A **view** in SQL is a virtual table based on the result of a query. It does not store data physically; instead, it retrieves data from one or more tables dynamically when accessed. Views simplify complex queries, enhance security by restricting access to specific columns or rows, and improve maintainability.

Key Features of Views:

- **Virtual Table:** A view is a stored query, not a physical table.
- **Dynamic Content:** The data in a view change when the underlying table data changes.
- **Security:** Restrict access to sensitive data by exposing only necessary columns/rows.
- **Reusability:** Simplify repetitive complex queries by encapsulating them in a view.

Types of Views in SQL:

- **Simple View:** Derived from a single table, No aggregation, joins, or complex logic, Allows updates.

Example:

sql

```
CREATE VIEW HR_Employees AS
SELECT EmpID, Name
FROM Employees
WHERE Department = 'HR';
```

- **Complex View:** Derived from multiple tables, Includes joins, aggregations, or subqueries, Typically read-only.

Example:

sql

```
-- Table: Departments
CREATE TABLE Departments (
    DeptID INT,
    DeptName VARCHAR(50)
);

INSERT INTO Departments VALUES (1, 'HR'), (2, 'Finance'), (3, 'IT');

-- Create a complex view
CREATE VIEW EmployeeDetails AS
SELECT e.Name, e.Salary, d.DeptName
FROM Employees e
JOIN Departments d ON e.Department = d.DeptName;

-- Query the view
SELECT * FROM EmployeeDetails;
```


- **Materialized View:** Stores data physically (not virtual), Requires manual or scheduled refresh, Optimized for performance in large datasets.

Example (Oracle/SQL Server):

sql

```
CREATE MATERIALIZED VIEW SalesSummary
AS
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Department;
```

- **Indexed View:** A materialized view with an index for faster access, Useful for large and frequently accessed data.

Example (SQL Server):

sql

```
CREATE VIEW IndexedView
WITH SCHEMABINDING
AS
SELECT Department, COUNT(*) AS EmpCount
FROM Employees
GROUP BY Department;

-- Create index on the view
CREATE UNIQUE CLUSTERED INDEX IX_IndexedView ON IndexedView(Department);
```

- **Dynamic View:** Adjusts based on dynamic conditions.

Example:

sql

```
CREATE VIEW DynamicSalaryView AS
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

- **Partitioned View:** Combines data from multiple tables using UNION ALL, Helps manage large datasets by splitting them across tables.

Example:

sql

Copy code

```
-- Tables
CREATE TABLE Employees_Part1 (EmpID INT, Name VARCHAR(100), Salary DECIMAL(10, 2));
CREATE TABLE Employees_Part2 (EmpID INT, Name VARCHAR(100), Salary DECIMAL(10, 2));

-- Insert data
INSERT INTO Employees_Part1 VALUES (1, 'Alice', 50000);
INSERT INTO Employees_Part2 VALUES (2, 'Bob', 60000);

-- Partitioned view
CREATE VIEW AllEmployees AS
SELECT * FROM Employees_Part1
UNION ALL
SELECT * FROM Employees_Part2;

-- Query the view
SELECT * FROM AllEmployees;
```

11. Explain me WINDOWS FUNCTIONS in detail with example?

Window Functions are powerful tools in SQL used to perform calculations across a set of rows related to the current row, without collapsing the result set into a single row like aggregate functions. These functions allow detailed analysis while preserving row-level details.

Key Features of Window Functions:

- **Does Not Collapse Rows:** Unlike GROUP BY, window functions retain individual rows in the result.
- **Works on Windows (Partitions):** Operates on a "window" or subset of rows defined by the OVER clause.
- **Flexible Analysis:** Enables tasks like ranking, running totals, moving averages, and cumulative sums.

Syntax of a Window Function

sql

```
function_name (expression)
OVER (
    [PARTITION BY column]
    [ORDER BY column]
    [ROWS or RANGE clause]
)
```


- **function_name:** The specific window function, such as SUM(), AVG(), ROW_NUMBER(), etc.
- **PARTITION BY:** Divides the dataset into partitions; the function is applied separately to each partition.
- **ORDER BY:** Specifies the order of rows within each partition for the calculation.
- **ROWS or RANGE:** Defines the "window frame" relative to the current row.

Types of Window Functions:

- **Aggregate Functions:** Used with a window to calculate cumulative metrics like SUM, AVG, COUNT, MIN, MAX.

Total Sales for Each Region

sql

 Copy code


```
SELECT
    Employee,
    Region,
    Sales,
    SUM(Sales) OVER (PARTITION BY Region) AS TotalSalesByRegion
FROM Sales;
```

Employee	Region	Sales	TotalSalesByRegion
Alice	East	500	1800
Bob	East	700	1800
Eva	East	600	1800
Charlie	West	300	1600
David	West	400	1600
Frank	West	900	1600

- **Ranking Functions:** Functions like ROW_NUMBER, RANK, DENSE_RANK, and NTILE assign ranks to rows within partitions.

Rank Employees by Sales Within Their Region

sql

 Copy code

```
SELECT
    Employee,
    Region,
    Sales,
    RANK() OVER (PARTITION BY Region ORDER BY Sales DESC) AS RankBySales
FROM Sales;
```

Employee	Region	Sales	RankBySales
Bob	East	700	1
Eva	East	600	2
Alice	East	500	3
Frank	West	900	1
David	West	400	2
Charlie	West	300	3

- **Value Functions:** Functions like LAG, LEAD, FIRST_VALUE, and LAST_VALUE retrieve specific rows.

Find the Previous Sales (LAG) and Next Sales (LEAD)

sql

Copy code

```
SELECT
  Employee,
  Region,
  Sales,
  LAG(Sales) OVER (PARTITION BY Region ORDER BY Sales) AS PreviousSales,
  LEAD(Sales) OVER (PARTITION BY Region ORDER BY Sales) AS NextSales
FROM Sales;
```

Employee	Region	Sales	PreviousSales	NextSales
Alice	East	500	NULL	600
Eva	East	600	500	700
Bob	East	700	600	NULL
Charlie	West	300	NULL	400
David	West	400	300	900
Frank	West	900	400	NULL

12. What are Stored Procedures? Explain in detail with examples. Also include real time example

A **Stored Procedure** is a precompiled collection of one or more SQL statements that are stored on a database server. It allows users to execute complex SQL logic using a single call. Stored procedures improve performance, reusability, and maintainability of SQL code by enabling the centralization of frequently used queries or business logic.

Key Features of Stored Procedures:

- **Encapsulation:** Groups multiple SQL statements in a single unit.
- **Improved Performance:** Precompiled execution enhances performance.
- **Reusability:** Once created, they can be reused across applications or users.
- **Security:** Permissions can be granted to execute the procedure without exposing the underlying database objects.
- **Error Handling:** Includes logic for exception handling to manage errors effectively.

Syntax of a Stored Procedure


sql

Copy code

```
CREATE PROCEDURE procedure_name
AS
BEGIN
  -- SQL statements
END;
```

With Parameters:

sql


 Copy code

```
CREATE PROCEDURE procedure_name (  
    @parameter_name data_type [IN | OUT | INOUT],  
    ...  
)  
AS  
BEGIN  
    -- SQL statements  
END;
```

Example 1: Basic Stored Procedure

Requirement: Retrieve all employees from the "employees" table.


sql

 Copy code

```
CREATE PROCEDURE GetAllEmployees  
AS  
BEGIN  
    SELECT * FROM employees;  
END;
```

To Execute:

sql


 Copy code

```
EXEC GetAllEmployees;
```

Example 2: Stored Procedure with Input Parameters

Requirement: Retrieve employee details by their department.


sql

 Copy code

```
CREATE PROCEDURE GetEmployeesByDepartment  
    @DepartmentName NVARCHAR(50)  
AS  
BEGIN  
    SELECT *  
    FROM employees  
    WHERE department = @DepartmentName;  
END;
```

To Execute:

sql


 Copy code

```
EXEC GetEmployeesByDepartment @DepartmentName = 'HR';
```

Example 3: Stored Procedure with Output Parameters

Requirement: Get the count of employees in a specific department.


sql

 Copy code

```
CREATE PROCEDURE GetEmployeeCountByDepartment
    @DepartmentName NVARCHAR(50),
    @EmployeeCount INT OUTPUT
AS
BEGIN
    SELECT @EmployeeCount = COUNT(*)
    FROM employees
    WHERE department = @DepartmentName;
END;
```

To Execute:

sql

 Copy code


```
DECLARE @Count INT;
EXEC GetEmployeeCountByDepartment @DepartmentName = 'HR', @EmployeeCount = @Count OUTPUT;
PRINT @Count;
```



Example 4: Stored Procedure with Multiple Parameters

Requirement: Add a new employee record.


sql

 Copy code

```
CREATE PROCEDURE AddNewEmployee
    @EmployeeName NVARCHAR(100),
    @Department NVARCHAR(50),
    @Salary FLOAT
AS
BEGIN
    INSERT INTO employees (name, department, salary)
    VALUES (@EmployeeName, @Department, @Salary);
END;
```

To Execute:

sql


 Copy code

```
EXEC AddNewEmployee @EmployeeName = 'John Doe', @Department = 'Finance', @Salary = 75000;
```

Real-Time Example: Payroll Calculation

Requirement: Calculate the total salary expense for a given department.


sql

 Copy code

```
CREATE PROCEDURE CalculateTotalSalary
    @DepartmentName NVARCHAR(50),
    @TotalSalary FLOAT OUTPUT
AS
BEGIN
    SELECT @TotalSalary = SUM(salary)
    FROM employees
    WHERE department = @DepartmentName;
END;
```

To Execute:

sql

 Copy code


```
DECLARE @Total FLOAT;
EXEC CalculateTotalSalary @DepartmentName = 'IT', @TotalSalary = @Total OUTPUT;
PRINT @Total;
```

Real-Time Application

Scenario: Dynamic Reporting System for Monthly Sales

Task: Create a stored procedure to generate a monthly sales report.

sql

 Copy code

```
CREATE PROCEDURE GenerateMonthlySalesReport
    @StartDate DATE,
    @EndDate DATE
AS
BEGIN
    SELECT product_id,
           SUM(quantity) AS TotalQuantitySold,
           SUM(total_price) AS TotalRevenue
    FROM sales
    WHERE sale_date BETWEEN @StartDate AND @EndDate
    GROUP BY product_id
    ORDER BY TotalRevenue DESC;
END;
```

To Execute:

sql

Copy code

```
EXEC GenerateMonthlySalesReport @StartDate = '2025-01-01', @EndDate = '2025-01-31';
```

13. When I use CTE and View. Explain me in details

When to Use a CTE:

- **Breaking Down Complex Queries:** When a query involves multiple nested subqueries, using a CTE can simplify it by breaking it into readable parts.
- **Temporary Query-Specific Needs:** When you need a temporary result set that is not reused outside the query.
Example: Calculating aggregates and using them only in the query at hand.
- **Recursive Queries:** CTEs are a must when you need to perform recursive operations, such as traversing hierarchical data.
- **Ad-Hoc Queries:** When you need to analyse data temporarily or test logic without storing it.

When to Use a View:

- **Reusable Queries:** If you need to use the same query logic across multiple applications or queries, a view eliminates duplication.
- **Encapsulation and Simplification:** When you want to encapsulate complex joins, calculations, or aggregations into a single virtual table. This simplifies query writing for end-users.
- **Abstraction:** To provide a layer of abstraction, e.g., exposing a simplified schema to end-users while hiding underlying complexity.
- **Security:** Views can restrict access to sensitive columns by exposing only selected columns.
- **Indexed Materialized Views (Performance):** In some databases, views can be materialized, improving performance for repeated queries by storing results.

14. How to handle NULL value in SQL?

Handling **NULL** values in SQL is crucial for data integrity, query accuracy, and preventing unexpected results.

Here's a comprehensive guide on how to manage NULL values:

- **Understanding NULL in SQL:** **NULL** represents missing or unknown data. It is not the same as 0, an empty string ("), or a blank space. Operations involving NULL generally result in NULL (e.g., NULL + 10 results in NULL).
- **Detecting NULL Values:** Use the **IS NULL** or **IS NOT NULL** operators to filter rows with or without NULL values.

Example:

sql

```
SELECT *  
FROM Employees  
WHERE ManagerID IS NULL; -- Finds employees without a manager
```

sql

```
SELECT *  
FROM Employees  
WHERE ManagerID IS NOT NULL; -- Finds employees with a manager
```


- Replacing NULL Values: Use **COALESCE** or **IFNULL** to replace NULL values with default values.

Example: Using **COALESCE**

sql

```
SELECT EmployeeID,  
       COALESCE(Bonus, 0) AS Bonus -- Replaces NULL with 0  
FROM Salaries;
```

Example: Using **IFNULL** (MySQL)

sql

```
SELECT EmployeeID,  
       IFNULL(PhoneNumber, 'Not Provided') AS ContactNumber  
FROM Employees;
```

- Handling NULL in Aggregates: Most aggregate functions ignore NULL values. If you want to include them, handle them explicitly.

Example:

sql

```
SELECT  
    COUNT(*) AS TotalEmployees,      -- Includes NULLs  
    COUNT(ManagerID) AS WithManager, -- Excludes NULLs  
    SUM(COALESCE(Salary, 0)) AS TotalSalary -- Handles NULL salaries  
FROM Employees;
```

- Conditional Logic with NULL: Use CASE expressions or logical functions to handle NULL conditions.

Example: Assigning Labels Based on NULL

sql

```
SELECT EmployeeID,  
       CASE  
           WHEN ManagerID IS NULL THEN 'No Manager'  
           ELSE 'Has Manager'  
       END AS ManagerStatus  
FROM Employees;
```

- Using NULL-Safe Equality: Use <=> (NULL-safe equal) in MySQL to compare NULL values directly.

Example:

sql

```
SELECT *
FROM Employees
WHERE ManagerID <=> NULL; -- TRUE if ManagerID is NULL
```

- Using Default Values for Inserts: Define default values in table creation to avoid NULL values.

Example:

sql

```
CREATE TABLE Products (
    ProductID INT,
    Price DECIMAL(10, 2) DEFAULT 0 -- Default value for NULL
);
```

15. How to perform SELF JOIN and when it's more effective? Explain with example

A **Self Join** is a join where a table is joined with itself. It's useful for comparing rows within the same table. The table is assigned different aliases to distinguish its instances during the join. By using a **Self Join**, you can address complex queries that involve relationships within a single table effectively.

When to Use a Self-Join:

- Hierarchical Data Representation: To analyze relationships such as employees and their managers or categories and subcategories.
- Comparing Rows Within the Same Table: Finding duplicate records or rows that meet specific relational criteria.
- Data Transformation: For tasks like identifying successive rows or calculating differences between rows.

Syntax for Self Join

sql

```
SELECT A.column1, B.column2
FROM TableName A
INNER JOIN TableName B
ON A.column = B.column;
```


Example 1: Employee-Manager Relationship

Table: Employees

EmployeeID	EmployeeName	ManagerID
1	Alice	NULL
2	Bob	1
3	Carol	1
4	Dave	2

Query: Get Employee and Their Manager

sql

 Copy code

```
SELECT
    e.EmployeeName AS Employee,
    m.EmployeeName AS Manager
FROM Employees e
LEFT JOIN Employees m
ON e.ManagerID = m.EmployeeID;
```



Output:

Employee	Manager
Alice	NULL
Bob	Alice
Carol	Alice
Dave	Bob

16. What is ERM (Entity Relationship Model) in SQL

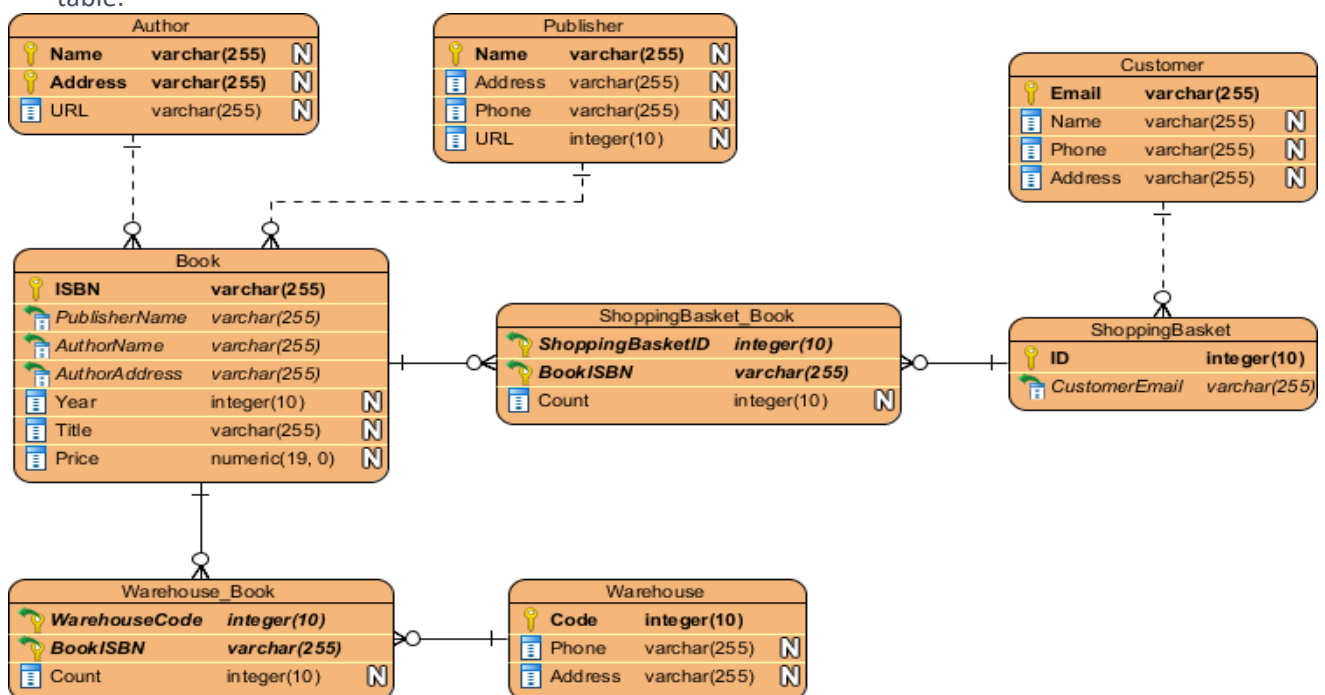
ERM (Entity Relationship Model) in SQL refers to a conceptual framework used for structuring and designing databases. While ERM itself isn't a feature of SQL, it is a key step in database design that influences how databases are created, queried, and managed using SQL. The Entity-Relationship (ER) Model is used to visually map out the relationships between entities (tables) in a database.

What is the Entity Relationship Model (ERM): The Entity Relationship Model (ERM) is a blueprint for database design that helps define how data is structured and how relationships between different data elements are represented. It uses diagrams known as Entity-Relationship Diagrams (ERD) to illustrate the entities, their attributes, and their relationships with each other.

Key Components of ERM:

- **Entities:** Entities are objects or things within the domain that have a distinct existence in the database (e.g., customers, products, employees). These typically become tables in a relational database.
Example: Customers, Orders, Employees.

- **Attributes:** Attributes describe properties or characteristics of an entity (e.g., name, address, date of birth).
Example: For an Employee entity, attributes could include EmployeeID, FirstName, LastName, HireDate.
- **Relationships:** Define how two or more entities are related to each other. They can be one-to-one, one-to-many, or many-to-many
Example: An Order is placed by a Customer, which is a one-to-many relationship (one customer can place multiple orders)
- **Primary Keys (PK):** A primary key is a unique identifier for each record in a table (entity). It ensures that each row in a table is uniquely identifiable.
Example: CustomerID in the Customers table.
- **Foreign Keys (FK):** A foreign key is an attribute in one table that links to the primary key of another table, establishing a relationship between the two entities.
Example: CustomerID in the Orders table refers to the primary key CustomerID in the Customers table.



17. What is database schema and different types of it. Explain in detail with example?

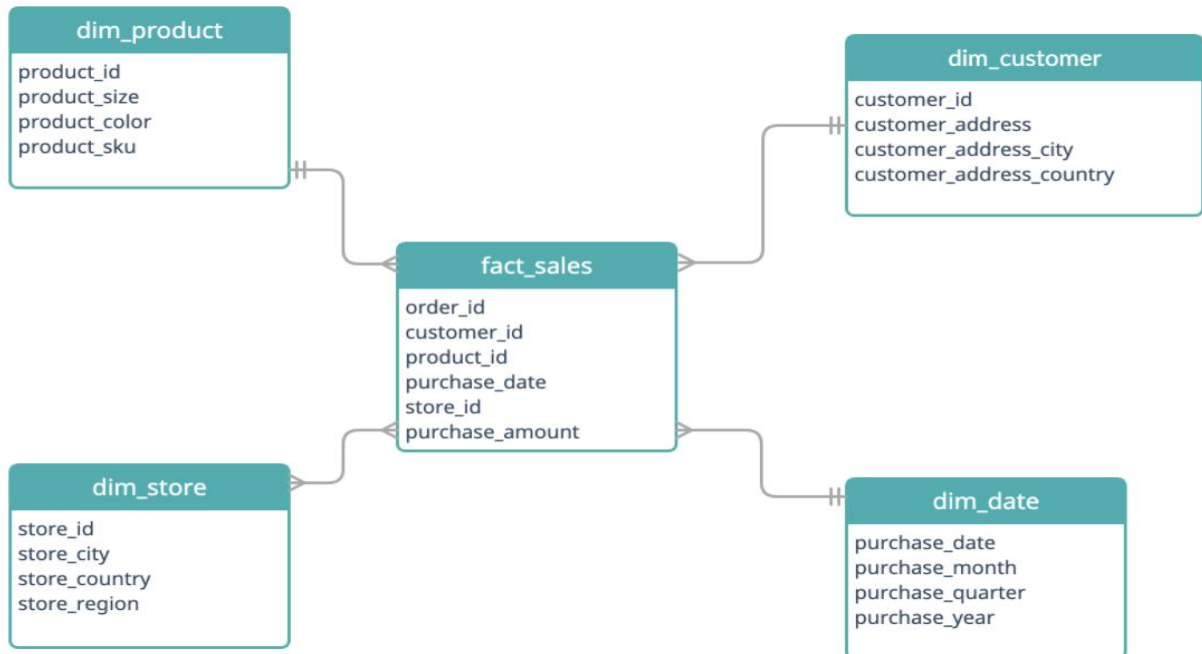
A **database schema** is a blueprint or architecture of how data is organized within a database. It defines the structure of the database, including tables, columns, relationships, views, indexes, constraints, and other elements. Schemas ensure consistency and provide a logical organization of data, making it easier to understand and work with the database.

A database schema is critical for organizing and managing data effectively. The choice of schema type depends on the specific requirements, whether it's for transactional systems, analytical systems, or simpler flat-file storage. Each schema type has unique benefits and use cases, and understanding these helps in designing efficient databases.

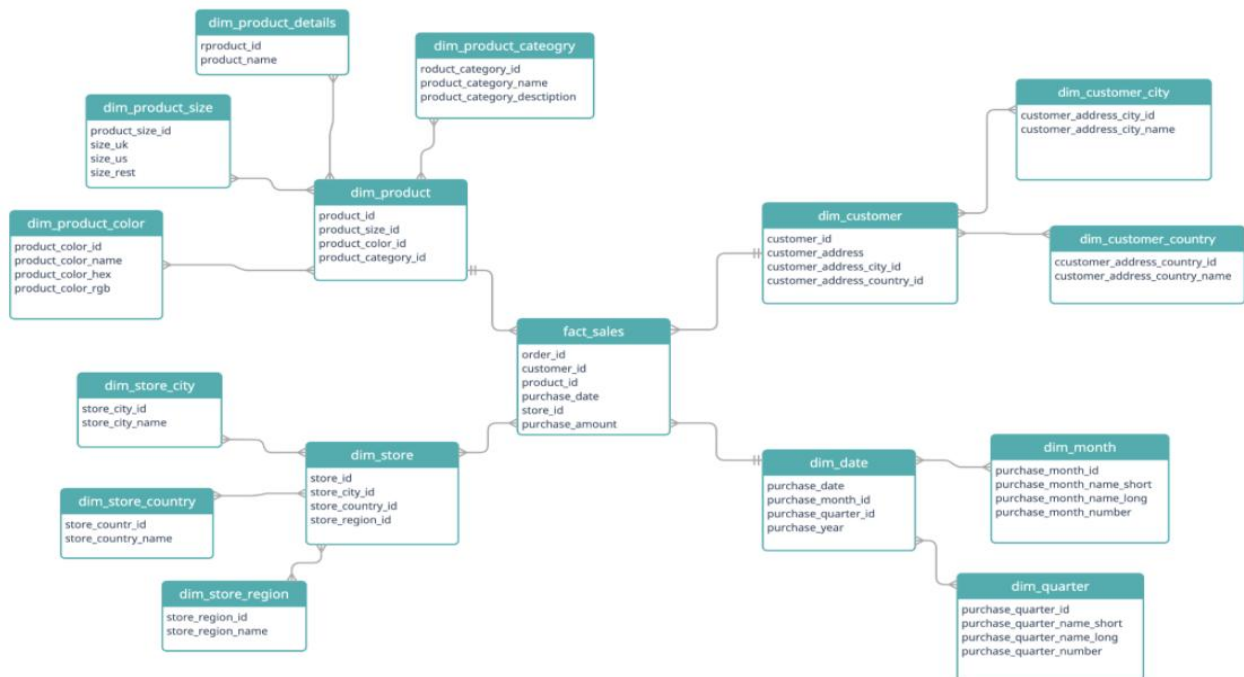
Types of Database Schemas:

- **Physical Schema:** Describes how data is physically stored in the database, including details about file storage, indexes, partitions, and other low-level configurations. Focused on performance and storage.

- **Logical Schema:** Represents the logical structure of the database. It includes tables, columns, relationships, and constraints but abstracts away the physical storage details. Defines how data is logically organized and accessible.
- **Star Schema (Dimensional Modeling):** A type of schema commonly used in data warehousing where a central fact table is connected to multiple dimension tables. To optimize querying for analytical purposes.



- **Snowflake Schema (Dimensional Modeling):** An extension of the star schema where dimension tables are normalized into multiple related tables. Reduces redundancy in dimension data.



- **Flat Schema:** A schema where all the data is stored in a single table without any relationships or normalization.

18. How to identify and optimizing slow running queries in SQL?

Optimizing slow-running queries in SQL is critical for improving database performance. Optimizing slow queries involves a mix of identifying the bottlenecks, understanding the database's execution plan, and applying techniques like indexing, query rewriting, and schema adjustments. Regular monitoring and maintenance are essential to ensure consistent performance.

Identifying Slow Queries:

- **Use SQL Query Profiling Tools:**

- **EXPLAIN or EXPLAIN PLAN:** These commands show the execution plan of a query, revealing how SQL processes it.

Example in MySQL:

sql

```
EXPLAIN SELECT * FROM Orders WHERE CustomerID = 101;
```

- **Execution Time Analysis:** Use SHOW PROFILE in MySQL to measure query performance.

sql

```
SET profiling = 1;
SELECT * FROM Orders WHERE CustomerID = 101;
SHOW PROFILE FOR QUERY 1;
```

- **Query Performance Logs:** Enable slow query logs in the database.

MySQL:

sql

Copy

```
SET GLOBAL slow_query_log = 1;
SET GLOBAL long_query_time = 2; -- Logs queries taking more than 2 seconds.
```

- **Database Monitoring Tools:** Use tools like pgAdmin (PostgreSQL), MySQL Workbench, or third-party solutions (e.g., New Relic, SolarWinds) to monitor slow queries.

Common Causes of Slow Queries:

- **Missing Indexes:** Queries requiring full table scans due to lack of appropriate indexes.
- **Suboptimal Joins:** Joining large tables without indexes on the join columns.
- **Complex Subqueries:** Nested subqueries that can often be replaced with joins or Common Table Expressions (CTEs).
- **Inefficient Filters:** Using non-sargable filters (e.g., LIKE '%value%', FUNCTION(column)).
- **Large Dataset Operations:** Queries without proper limits or pagination.

Optimizing Slow Queries:

- **Indexing:** Add indexes on columns used in WHERE, JOIN, GROUP BY, and ORDER BY clauses. Include columns used in SELECT to avoid accessing the main table.

sql


```
CREATE INDEX idx_customer_id ON Orders(CustomerID);
```

sql

```
CREATE INDEX idx_customer_date ON Orders(CustomerID, OrderDate);
```

- **Optimize Joins:** Use indexes on join columns. Avoid unnecessary joins.
- **Rewrite Queries:** Replace subqueries with joins.

sql

 Copy code

```
-- Subquery
SELECT CustomerID FROM Orders WHERE OrderID IN (SELECT OrderID FROM Shipments);

-- Optimized with JOIN
SELECT DISTINCT O.CustomerID
FROM Orders O
JOIN Shipments S ON O.OrderID = S.OrderID;
```

- **Use CTEs or Temp Tables:** Break down complex queries.

sql

```
WITH CustomerOrders AS (
    SELECT CustomerID, SUM(OrderAmount) AS TotalAmount
    FROM Orders
    GROUP BY CustomerID
)
SELECT * FROM CustomerOrders WHERE TotalAmount > 1000;
```

- **Pagination for Large Data:** Limit the number of rows fetched. **Avoid SELECT *** - Specify only the required columns

sql

```
SELECT * FROM Orders ORDER BY OrderDate DESC LIMIT 50 OFFSET 0;
```

- **Partitioning:** Split large tables into smaller partitions.

sql

```
CREATE TABLE Orders2023 PARTITION BY RANGE (OrderDate) (  
    PARTITION p1 VALUES LESS THAN ('2023-06-01'),  
    PARTITION p2 VALUES LESS THAN ('2023-12-31')  
);
```

- **Caching:** Cache frequently accessed data at the application or database level.

Materialized views (PostgreSQL, Oracle):

sql

```
CREATE MATERIALIZED VIEW mv_top_customers AS  
SELECT CustomerID, SUM(OrderAmount) AS TotalAmount  
FROM Orders  
GROUP BY CustomerID;
```

- **Batch Processing:** Process large operations in smaller batches.

sql

```
DELETE FROM Orders WHERE OrderDate < '2020-01-01' LIMIT 1000;
```

19. Describe how to measure performance of SQL queries.

Measuring the performance of SQL queries is essential to ensure efficiency, optimize resource utilization, and improve the overall responsiveness of a database system. Here's a detailed breakdown of how to measure SQL query performance:

Analyze Query Execution Plans: Execution plans provide a step-by-step breakdown of how the SQL engine executes a query.

- **Tools:**

- In SQL Server: Use `SET STATISTICS PROFILE ON` or view the "Actual Execution Plan."
- In MySQL: Use `EXPLAIN` or `EXPLAIN ANALYZE`.
- In PostgreSQL: Use `EXPLAIN` or `EXPLAIN ANALYZE`.

- **Key Metrics:**

- Cost of operations (e.g., joins, scans, aggregations).
- Index usage (whether indexes are being utilized or not).
- Execution order of operations.

- **Example:**

sql


Copy code

```
EXPLAIN SELECT * FROM Orders WHERE OrderDate = '2025-01-01';
```


Measure Query Execution Time: Check how long a query takes to execute.

- **Methods:**
 - Use built-in timing functions in database systems:
 - MySQL: `SHOW PROFILE` or `SET profiling = 1`.
 - PostgreSQL: Use `EXPLAIN ANALYZE`.
 - SQL Server: Use `SET STATISTICS TIME ON`.
- **Key Metric:** Total execution time (in milliseconds or seconds).
- **Example:**

sql

 Copy code

```
SET profiling = 1;
SELECT * FROM Orders WHERE CustomerID = 123;
SHOW PROFILE FOR QUERY 1;
```


Monitor Resource Utilization: Observe how a query utilizes CPU, memory, and disk I/O

- **Tools:**
 - SQL Server: Activity Monitor or SQL Profiler.
 - MySQL: Performance Schema.
 - PostgreSQL: `pg_stat_statements`.
- **Key Metrics:**
 - CPU time.
 - Disk reads/writes (logical and physical I/O).
 - Memory usage.

Check Query Statistics: Gather statistics on rows processed and operations performed

- **Tools:**
 - SQL Server: `SET STATISTICS IO ON` and `SET STATISTICS TIME ON`.
 - MySQL: Query Profiler.
 - PostgreSQL: `pg_stat_activity` and `pg_stat_statements`.
- **Key Metrics:**
 - Number of rows read, returned, or written.
 - Number of logical and physical reads.
- **Example (SQL Server):**

sql

 Copy code

```
SET STATISTICS IO ON;  
SELECT * FROM Products WHERE CategoryID = 5;
```

Use Database Monitoring Tools: Employ third-party or built-in database tools to track performance

- **Examples:**
 - SQL Server: SQL Server Profiler, Azure SQL Analytics.
 - MySQL: MySQL Enterprise Monitor.
 - PostgreSQL: pgAdmin, AWS CloudWatch for RDS.

20. How do you rewrite query to improve its performance?

Rewriting SQL queries to improve performance involves optimizing their structure, minimizing resource consumption, and ensuring they take full advantage of the database's features.

Use Proper Indexing: Queries scanning large tables due to missing indexes. Add indexes on frequently queried columns, especially in WHERE, JOIN, and GROUP BY clauses.

Avoid SELECT * (Specify Columns Explicitly): Retrieving all columns unnecessarily increases data transfer and memory usage. Select only required columns.

Use JOINS Instead of Subqueries: Subqueries can be inefficient and hard to optimize. Replace subqueries with JOIN clauses where possible.

Use EXISTS Instead of IN: IN can be slower when dealing with large datasets. Use EXISTS for better performance

Optimize GROUP BY with Aggregations: Aggregations over large datasets can be slow. Ensure indexes support the GROUP BY column and avoid unnecessary computations.

Limit the Use of Wildcards: Wildcards like % at the beginning of a string prevent index usage. Use wildcards judiciously or avoid them entirely.

Avoid Fetching Unnecessary Rows: Retrieving more rows than required impacts performance. Use LIMIT, OFFSET, or TOP to fetch only the required rows.

21. Are JOINS Better Than Subqueries?

JOINS are generally faster and better suited for combining related data. **Subqueries** are helpful for simplifying specific conditions or handling aggregated data. The choice depends on the query complexity, dataset size, and performance requirements.

Performance	
JOINS	Subqueries
Faster for large datasets as they are optimized by database engines.	Can be slower due to repeated execution of inner queries.
Executes once and fetches combined results.	May execute the inner query multiple times for each row.

When to Use	
JOINS	Subqueries
Combining Data: Joining related data across multiple tables.	Logic Isolation: When isolating complex filtering logic is helpful.
Relational Data: When tables have defined relationships.	Aggregations: Filtering based on calculated or aggregated data.
Performance Critical: Preferred for performance and scalability.	Simplifying Queries: Useful for breaking down complex queries.

22. Difference between OLTP and OLAP. explain me with examples

The main difference between **OLTP (Online Transaction Processing)** and **OLAP (Online Analytical Processing)** lies in their purpose, use cases, and design. By understanding these differences, you can better align the database architecture with the specific needs of your application or analysis.

OLTP (Online Transaction Processing): OLTP systems are designed to manage and execute real-time transactional operations. These systems are primarily used for day-to-day operations in businesses, where speed and accuracy are crucial for handling multiple, concurrent transactions.

Key Characteristics of OLTP Systems:

- **Transactional Nature:** Focuses on transactions like adding, updating, or deleting records.
- **Normalized Data:** Ensures that data is stored efficiently, avoiding redundancy and improving consistency.
- **High Volume:** Can handle thousands or millions of simple transactions per second.
- **Data Integrity:** Enforces constraints like primary keys, foreign keys, and validations to maintain data integrity.

Example Scenarios of OLTP:

- **Banking Systems:**

- Recording deposits, withdrawals, and fund transfers.

- Example Query:

```
sql
```

```
INSERT INTO Transactions (AccountID, Amount, TransactionType)
VALUES (12345, 500, 'Deposit');
```

- **E-commerce Platforms:**

- Adding an item to a customer's shopping cart.

- Example Query:

```
sql
```

```
INSERT INTO Cart (CustomerID, ProductID, Quantity)
VALUES (98765, 45678, 2);
```

Advantages of OLTP:

- **Fast Processing:** Designed to execute short and quick queries efficiently.
- **Data Accuracy:** Transactions are ACID-compliant, ensuring atomicity, consistency, isolation, and durability.
- **Real-Time Access:** Enables businesses to respond to events as they occur.

Limitations of OLTP:

- **Not Ideal for Analysis:** Due to normalized schemas and real-time updates, OLTP systems are not optimized for running complex analytical queries.
- **Storage Overhead:** Strict normalization can lead to multiple tables joins, which increases complexity.

OLAP (Online Analytical Processing): OLAP systems are designed to perform complex queries and aggregations on historical data. They are used in business intelligence (BI) applications to provide insights that aid in decision-making.

Key Characteristics of OLTP Systems:

- **Analytical Nature:** Focuses on aggregating and analyzing large volumes of data
- **Denormalized Data:** Data is often structured in star or snowflake schemas to simplify and speed up querying.
- **Historical Data:** Stores data collected over months or years, often in summarized or aggregated form.
- **Batch Processing:** Queries are read-intensive and can process large datasets.

Example Scenarios of OLTP:

- **Sales Analysis:**

- Determining total sales by region and year.
- Example Query:

```
sql
```

```
SELECT Region, SUM(SalesAmount) AS TotalSales
FROM SalesFact
WHERE Year = 2024
GROUP BY Region;
```

- **Marketing Campaign Insights:**

- Evaluating the impact of a marketing campaign over time.
- Example Query:

```
sql
```

```
SELECT CampaignID, AVG(SalesGrowth) AS AverageGrowth
FROM CampaignPerformance
WHERE Year BETWEEN 2022 AND 2023
GROUP BY CampaignID;
```

Advantages of OLTP:

- **Complex Analysis:** Enables multidimensional analysis of data (e.g., slicing and dicing)
- **Decision Support:** Provides insights for strategic decision-making
- **Historical Perspective:** Helps analyze trends over time

Limitations of OLTP:

- **Slow Data Updates:** Not designed for real-time data updates
- **Storage Requirements:** Historical and aggregated data require significant storage

Key Differences Between OLTP and OLAP

Feature	OLTP	OLAP
Purpose	Transaction Management	Analytical Processing
Data Model	Normalized (3NF)	Denormalized (Star/Snowflake Schema)
Query Type	Simple, transaction-based queries	Complex, aggregate queries
Data	Real-Time	Historical
Performance	Optimized for fast read/write transactions	Optimized for read-intensive queries
Users	Operational Staff (e.g., cashiers, agents)	Analysts, Executives
Examples	Banking Systems, E-commerce Platforms	Data Warehouses, Business Intelligence Tools

23. What is user defined functions. Explain in detail with examples

A **User-Defined Function (UDF)** is a custom function created by the user to extend the capabilities of SQL or programming languages. It is used to encapsulate logic that can be reused across queries or programs. User-Defined Functions (UDFs) are powerful tools for encapsulating logic and promoting code reuse in SQL and programming. However, they should be used judiciously to avoid performance bottlenecks, especially in large-scale data processing. By understanding their types and applications, you can leverage UDFs to streamline development and enhance query efficiency.

Types of UDFs:

- **Scalar Functions:** Return a single value.

Create a UDF to calculate the square of a number:

sql [Copy code](#)

```
CREATE FUNCTION CalculateSquare (@Number INT)
RETURNS INT
AS
BEGIN
    RETURN @Number * @Number;
END;
```

Usage:

sql [Copy code](#)

```
SELECT dbo.CalculateSquare(4) AS SquareResult; -- Output: 16
```

- **Table-Valued Function Example:** Return a table

Create a UDF to return employees from a specific department:

sql [Copy code](#)

```
CREATE FUNCTION GetEmployeesByDepartment (@DeptID INT)
RETURNS TABLE
AS
RETURN (
    SELECT EmployeeID, Name, DepartmentID
    FROM Employees
    WHERE DepartmentID = @DeptID
);
```

Usage:

sql [Copy code](#)

```
SELECT *
FROM dbo.GetEmployeesByDepartment(101);
```

- **Multi-Statement Table-Valued Function Example:** Allow multiple statements and return a table.

Create a UDF to return all employees with their salary after a 10% bonus:

```
sql Copy code

CREATE FUNCTION GetEmployeesWithBonus ()
RETURNS @EmployeeBonus TABLE (
    EmployeeID INT,
    Name VARCHAR(100),
    Salary DECIMAL(10, 2),
    BonusSalary DECIMAL(10, 2)
)
AS
BEGIN
    INSERT INTO @EmployeeBonus
    SELECT
        EmployeeID,
        Name,
        Salary,
        Salary * 1.1 AS BonusSalary
    FROM Employees;

    RETURN;
END;
```

Usage:

```
sql Copy code

SELECT *
FROM dbo.GetEmployeesWithBonus();
```

Best Practices for UDFs:

- **Optimize Performance:** Avoid using scalar UDFs in WHERE or JOIN clauses; prefer inline TVFs
- **Keep It Simple:** Avoid overcomplicating logic inside UDFs.
- **Index Usage:** Ensure that UDFs do not prevent the optimizer from using indexes effectively.
- **Document Logic:** Clearly describe the purpose and parameters of the UDF for maintainability.

24. Difference between functions and Stored procedures in sql

Key Takeaways:

- **Functions** are best for encapsulating calculations and logic that do not affect database state.
- **Stored Procedures** are better suited for performing tasks, managing transactions, and handling complex workflows.
- The choice depends on the use case—if you need reusable logic in queries, use a **function**. For broader tasks, data manipulation, or process workflows, use a **stored procedure**.

Aspect	Function	Stored Procedure
Purpose	Designed for computations and returning a value or table.	Designed for performing tasks like data modification, logic execution, or batch processing.
Return Value	Must return a value (scalar or table).	May or may not return a value. Can return multiple result sets.
Usage in Queries	Can be used in <code>SELECT</code> , <code>WHERE</code> , <code>GROUP BY</code> , and <code>HAVING</code> clauses.	Cannot be used directly in queries like <code>SELECT</code> .
Parameters	Only <code>IN</code> parameters are allowed.	Supports <code>IN</code> , <code>OUT</code> , and <code>INOUT</code> parameters.
Transactions	Cannot use <code>TRY-CATCH</code> or manage transactions within.	Can manage transactions using <code>BEGIN</code> , <code>COMMIT</code> , and <code>ROLLBACK</code> .
Side Effects	Functions cannot modify database state (e.g., no <code>INSERT</code> , <code>UPDATE</code> , or <code>DELETE</code>).	Stored procedures can modify the database state by performing <code>INSERT</code> , <code>UPDATE</code> , or <code>DELETE</code> operations.
Error Handling	Limited error handling capabilities.	Robust error handling using <code>TRY-CATCH</code> blocks.
Compiled	Compiled every time they are called.	Stored in a compiled state in the database, leading to better performance.
Output	Returns a single value, a table, or scalar values.	Can return multiple result sets or output parameters.
Execution	Called as part of an SQL expression (e.g., <code>SELECT dbo.function_name()</code>).	Executed using <code>EXEC</code> or <code>EXECUTE</code> .
Example Use Case	Commonly used for calculations, transformations, and reusable logic in queries.	Commonly used for batch processing, data manipulation, or complex business logic.

25. What are some common SQL coding practice you follow?

Adhering to best practices when writing SQL improves code readability, maintainability, and performance.

Code Formatting:

- **Use Consistent Indentation:**

- Proper indentation makes the code more readable.

sql

Copy Edit

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE department_id = 10
ORDER BY last_name;
```


- **Write Keywords in Uppercase:**

- SQL keywords (`SELECT`, `WHERE`, `JOIN`, etc.) should be in uppercase for better distinction.

- **Use Meaningful Aliases:**

- Use short but descriptive aliases and avoid ambiguity.

sql

Copy Edit

```
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

Query Optimization:

- **Select Only Necessary Columns:**

- Avoid `SELECT *` to reduce unnecessary data retrieval.

sql

Copy Edit

```
SELECT employee_id, first_name FROM employees; -- Better than SELECT *
```

- **Use Proper Indexing:**

- Create indexes on columns used in `WHERE`, `JOIN`, and `GROUP BY` clauses.

- **Filter Early:**

- Apply `WHERE` conditions to limit the number of rows processed.

sql

Copy Edit

```
SELECT employee_id FROM employees WHERE department_id = 10;
```

- **Avoid Nested Subqueries:**

- Use `JOIN` or `WITH` clauses instead for better performance.

sql

Copy Edit

```
-- Avoid
SELECT * FROM employees WHERE department_id IN (SELECT department_id FROM departments

-- Better
SELECT e.*
FROM employees e
JOIN departments d ON e.department_id = d.department_id
WHERE d.location_id = 1700;
```

Naming Conventions:

- **Use Descriptive Table and Column Names:** Avoid generic names like table1, col1. Use employee_id instead of id.
- **Follow Naming Patterns:** Use snake_case or camelCase consistently.

Use Comments:

- **Add Comments for Complex Logic:**

```
sql                                                                    Copy Edit

-- Fetch employees who have been with the company for over 10 years
SELECT employee_id, hire_date
FROM employees
WHERE hire_date < DATEADD(YEAR, -10, GETDATE());
```

Error Handling:

- **Handle NULLs Explicitly:**
 - Use COALESCE or ISNULL to avoid unexpected behavior.

```
sql                                                                    Copy Edit

SELECT COALESCE(salary, 0) AS salary FROM employees;
```

- **Validate Data:**
 - Use constraints like NOT NULL, CHECK, and DEFAULT in table definitions.

Modularization:

- **Use Views or CTEs:**
 - Break down complex queries into manageable parts.

```
sql                                                                    Copy Edit

WITH DepartmentSales AS (
    SELECT department_id, SUM(sales) AS total_sales
    FROM orders
    GROUP BY department_id
)
SELECT d.department_id, d.total_sales
FROM DepartmentSales d
WHERE d.total_sales > 10000;
```

- **Use Functions for Reusable Logic:**
 - Encapsulate repetitive calculations in a user-defined function.

Testing and Debugging:

- **Test with Real Data:** Ensure queries handle edge cases (e.g., empty data, large datasets).
- **Use EXPLAIN or Execution Plans:** Analyze query performance and optimize accordingly.

Avoid Anti-Patterns:

- **Avoid Cartesian Joins:**

- Ensure `JOIN` conditions are always defined.

```
sql                                                                    Copy Edit

-- Bad: Cartesian Join
SELECT * FROM employees, departments;

-- Good
SELECT e.*, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

- **Avoid Hardcoding Values:**

- Use parameters for flexibility and maintainability.

Documentation:

- **Document Query Logic:** Include details about input, output, and assumptions.
- **Version Control:** Maintain a version history of changes in queries for collaboration.

Additional Scenario Based SQL Interview Questions (Chat GPT)

26. How can you ensure the portability of SQL scripts across different database systems?
27. What methods do you use for version controlling SQL scripts?
28. How do you document SQL code effectively?
29. What is the process of Extract, Transform, Load (ETL)?
30. How do you import/export data from/to a flat file using SQL?
31. Explain the steps for a basic ETL process in a data warehousing environment.
32. How do you cleanse and format data using SQL queries?
33. What tools do you use for automating data import/export routines?
34. How would you model a many-to-many relationship in SQL?
35. Describe how to manage hierarchical data in SQL.
36. How do you use SQL in financial applications for risk and portfolio analysis?
37. What steps do you take to troubleshoot a failed SQL query?
38. What methods do you employ to ensure data integrity?
39. Explain how to use SQL for predictive analysis and machine learning purposes.