Deloitte Data Analyst Interview Questions (0-3 Years) 9-13 LPA

1. What is SQL, and how is it used in data analysis?

Structured Query Language (SQL) is a powerful language used to interact with relational databases. It allows users to **retrieve, manipulate, and manage structured data efficiently**. SQL is essential in data analysis because it enables analysts to extract meaningful insights from large datasets stored in databases.

How SQL is Used in Data Analysis?

1. **Data Extraction:** SQL helps in retrieving relevant data using queries (e.g., SELECT * FROM sales_data WHERE year = 2024;).
2. **Data Cleaning & Transformation:** Analysts use SQL functions like CASE, COALESCE, TRIM, CAST, etc., to handle missing values, format data, and remove duplicates.
3. **Data Aggregation:** SQL allows summarizing data using functions like SUM(), AVG(), COUNT(), MAX(), and MIN().
4. **Joins & Merging Tables:** SQL enables combining data from multiple tables using JOIN operations, helping analysts gather insights from different sources.
5. **Filtering & Sorting Data:** The WHERE, ORDER BY, and GROUP BY clauses allow refining data based on specific conditions.
6. **Creating Views & Reports:** SQL can create **views** and stored procedures to automate reporting processes and improve performance.

---

2. Explain the Difference Between Primary and Foreign Keys in SQL

In relational databases, **Primary Key** and **Foreign Key** play a crucial role in establishing relationships between tables.

Primary Key:

A **Primary Key** is a column (or a set of columns) in a table that uniquely identifies each record.

- It **must contain unique values** and **cannot have NULL values**.
- Each table can have only **one primary key**.
- Ensures data integrity and prevents duplicate records.

✅ **Example:**

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

Here, Emp_ID is the **Primary Key**, ensuring each employee has a unique identifier.

Foreign Key:

A **Foreign Key** is a column in one table that **refers to the Primary Key** in another table, creating a relationship between them.

- It ensures **referential integrity** by enforcing constraints (i.e., no orphan records).
- Can have **duplicate values and NULLs** (if allowed).
- A table **can have multiple foreign keys**.

✅ **Example:**

```
CREATE TABLE Orders (
  Order_ID INT PRIMARY KEY,
  Emp_ID INT,
  Order_Date DATE,
  FOREIGN KEY (Emp_ID) REFERENCES Employees(Emp_ID)
);
```

Here, Emp_ID in the **Orders** table is a **Foreign Key** referencing Emp_ID in the **Employees** table. This ensures that an order is always linked to a valid employee.

---

3. What is a JOIN in SQL, and How Does it Work?

A **JOIN** in SQL is used to **combine rows from two or more tables** based on a related column. It helps retrieve meaningful insights by connecting different datasets.

Types of SQL JOINs:

1. **INNER JOIN** – Returns only matching records from both tables.
2. **LEFT JOIN (LEFT OUTER JOIN)** – Returns all records from the left table and matching records from the right table.
3. **RIGHT JOIN (RIGHT OUTER JOIN)** – Returns all records from the right table and matching records from the left table.
4. **FULL JOIN (FULL OUTER JOIN)** – Returns all records when there is a match in either table.
5. **CROSS JOIN** – Returns the Cartesian product of two tables.
6. **SELF JOIN** – A table joins itself using different aliases.

✅ **Example:** INNER JOIN

```
SELECT Employees.Emp_ID, Employees.Name, Orders.Order_ID, Orders.Order_Date
FROM Employees
INNER JOIN Orders ON Employees.Emp_ID = Orders.Emp_ID;
```

- This retrieves **only employees who have placed orders**.
- The condition ON Employees.Emp_ID = Orders.Emp_ID ensures matching records are selected.

✅ **Example:** LEFT JOIN

```
SELECT Employees.Emp_ID, Employees.Name, Orders.Order_ID, Orders.Order_Date
FROM Employees
LEFT JOIN Orders ON Employees.Emp_ID = Orders.Emp_ID;
```

- This retrieves **all employees**, including those who haven't placed orders.

Why are JOINS Important?

- Helps **combine data from multiple tables** efficiently.
- Reduces **data redundancy** and ensures normalization.
- Enables complex **data analysis and reporting**.

---

4. Describe the Various Types of JOINs and Provide Examples

SQL **JOINs** are used to combine data from two or more tables based on a related column. Here's a detailed breakdown of each type of **JOIN** with examples:

---

1 .INNER JOIN

- Returns **only matching rows** from both tables.
- If there is **no match**, the row is excluded.

✅ **Example:**

```
SELECT Employees.Emp_ID, Employees.Name, Orders.Order_ID, Orders.Order_Date
FROM Employees
INNER JOIN Orders ON Employees.Emp_ID = Orders.Emp_ID;
```

📌 **Scenario:** This retrieves **only employees who have placed orders**.

---

2 .LEFT JOIN (LEFT OUTER JOIN)

- Returns **all records from the left table** and **matching records from the right table**.
- If no match is found, NULL values appear for the right table's columns.

✅ **Example:**

```
SELECT Employees.Emp_ID, Employees.Name, Orders.Order_ID, Orders.Order_Date
FROM Employees
LEFT JOIN Orders ON Employees.Emp_ID = Orders.Emp_ID;
```

📌 **Scenario:** Retrieves **all employees**, including those who **haven't placed orders**.

---

3 .RIGHT JOIN (RIGHT OUTER JOIN)

- Returns **all records from the right table** and **matching records from the left table**.
- If no match is found, NULL values appear for the left table's columns.

✅ **Example:**

```
SELECT Employees.Emp_ID, Employees.Name, Orders.Order_ID, Orders.Order_Date
FROM Employees
RIGHT JOIN Orders ON Employees.Emp_ID = Orders.Emp_ID;
```

📌 **Scenario:** Retrieves **all orders**, including those that **aren't linked to any employee**.

---

4 .FULL JOIN (FULL OUTER JOIN)

- Returns **all records from both tables**, whether they match or not.
- NULLs appear where there is no match.

✅ **Example:**

```
SELECT Employees.Emp_ID, Employees.Name, Orders.Order_ID, Orders.Order_Date
FROM Employees
FULL JOIN Orders ON Employees.Emp_ID = Orders.Emp_ID;
```

📌 **Scenario:** Retrieves **all employees and orders**, including unmatched ones.

---

5 .CROSS JOIN

- Returns the **Cartesian product** of two tables (i.e., every row in the first table joins with every row in the second table).
- **No JOIN condition is needed.**

✅ **Example:**

```
SELECT Employees.Name, Orders.Order_ID
```

FROM Employees

CROSS JOIN Orders;

📌 **Scenario:** If Employees has 5 rows and Orders has 4 rows, the result will have **5 × 4 = 20 rows**.

---

6 .SELF JOIN

- A table joins **with itself** using aliases.
- Often used for hierarchical or relational comparisons (e.g., employees and their managers).

✅ **Example:**

SELECT E1.Name AS Employee, E2.Name AS Manager

FROM Employees E1

JOIN Employees E2 ON E1.Manager_ID = E2.Emp_ID;

📌 **Scenario:** Fetches employees along with their managers.

---

5. What is the Difference Between WHERE and HAVING Clauses?

Feature	WHERE Clause	HAVING Clause
Purpose	Filters rows before grouping	Filters aggregated/grouped data
Used With	SELECT, UPDATE, DELETE	GROUP BY queries
Can Use Aggregate Functions?	❌ No	✅ Yes
Example	WHERE Salary > 50000	HAVING AVG(Salary) > 50000

✅ **Example of WHERE:** (Before Aggregation)

SELECT * FROM Employees

WHERE Department = 'IT';

📌 **Filters all employees from the 'IT' department before any aggregation.**

✅ **Example of HAVING:** (After Aggregation)

```
SELECT Department, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY Department
HAVING AVG(Salary) > 50000;
```

📌 **Finds departments where the average salary is greater than 50,000.**

---

6. How Do You Use GROUP BY in SQL, and Why is it Important?

What is GROUP BY?

The GROUP BY statement is used to **group rows with the same values** in specified columns and **apply aggregate functions** like SUM(), COUNT(), AVG(), MIN(), and MAX().

Why is GROUP BY Important?

- **Summarizes data** based on key attributes.
- **Enables aggregation** on large datasets.
- **Used in reporting and analytics.**

✅ **Example 1:** Count Employees in Each Department

```
SELECT Department, COUNT(Emp_ID) AS Employee_Count
FROM Employees
GROUP BY Department;
```

📌 **Groups employees by department and counts them.**

✅ **Example 2:** Find Total Sales by Each Customer

```
SELECT Customer_ID, SUM(Total_Amount) AS Total_Spent
FROM Orders
GROUP BY Customer_ID;
```

📌 **Groups orders by Customer ID and calculates the total spending per customer.**

✅ **Example 3:** Average Salary by Job Role (Using HAVING)

```
SELECT Job_Title, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY Job_Title
HAVING AVG(Salary) > 70000;
```

📌 **Groups employees by job title and filters only those with an average salary above 70,000.**

---

7. What are Subqueries, and When Would You Use Them?

A **subquery** (or **nested query**) is a SQL query embedded inside another query. It is executed **first**, and its result is used by the **main (outer) query**.

When to Use Subqueries?

- **Filtering Data Dynamically:** Using results from another query in WHERE or HAVING.
- **Complex Data Retrieval:** When direct JOINs are not feasible or when data needs pre-processing before the main query.
- **Aggregation in Filtering:** Applying aggregate functions within WHERE.
- **Replacing Temporary Tables:** To improve readability without storing intermediate results.

Types of Subqueries

1. **Single-Row Subqueries** – Return one value.
2. **Multi-Row Subqueries** – Return multiple values (used with IN, ANY, ALL).
3. **Correlated Subqueries** – Depend on the outer query for execution.

✅ **Example 1: Subquery in WHERE Clause** (Finding employees earning above the average salary)

```
SELECT Name, Salary
```

FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

📌 **The subquery calculates the average salary, and the main query retrieves employees earning above that value.**

✅ **Example 2: Subquery in FROM Clause** (Using a subquery as a virtual table)

SELECT Department, Avg_Salary

FROM (SELECT Department, AVG(Salary) AS Avg_Salary FROM Employees GROUP BY Department) AS DeptSalary

WHERE Avg_Salary > 50000;

📌 **The subquery groups employees by department, and the outer query filters departments with an average salary above 50,000.**

✅ **Example 3: Correlated Subquery** (Finding employees with the highest salary in each department)

SELECT Name, Salary, Department

FROM Employees E1

WHERE Salary = (SELECT MAX(Salary) FROM Employees E2 WHERE E1.Department = E2.Department);

📌 **The subquery finds the max salary per department and compares it for each employee.**

---

8. Explain the Concept of Indexing and Its Impact on Query Performance

What is an Index?

An **index** is a database structure that improves query speed by allowing **faster data retrieval**. It works like a book index—pointing directly to the required data instead of scanning the entire table.

Impact of Indexing on Performance

✅ **Benefits of Indexing:**

- **Speeds up SELECT queries** by reducing row scans.
- **Enhances WHERE, JOIN, ORDER BY, and GROUP BY performance.**
- **Reduces disk I/O**, improving database efficiency.

⚠️ **Drawbacks of Indexing:**

- **Increases storage usage** as indexes require space.
- **Slows down INSERT, UPDATE, DELETE** operations because indexes need to be updated.

Types of Indexes

1. **Primary Index** – Automatically created on a table's **Primary Key.**
2. **Unique Index** – Prevents duplicate values in a column.
3. **Composite Index** – Created on multiple columns to speed up queries involving multiple filters.
4. **Full-Text Index** – Used for searching text-based data efficiently.
5. **Clustered Index** – Determines the **physical order of data** in a table (only one per table).
6. **Non-Clustered Index** – Stores index separately, allowing multiple indexes per table.

✅ **Example: Creating an Index**

```
CREATE INDEX idx_employee_name ON Employees(Name);
```

📌 **This index speeds up searches on the 'Name' column.**

✅ **Example: Composite Index for Faster Filtering**

```
CREATE INDEX idx_emp_dept_salary ON Employees(Department, Salary);
```

📌 **This improves performance for queries filtering by Department and Salary.**

---

9. What is Normalization, and Why is It Important in Database Design?

What is Normalization?

Normalization is the **process of structuring a relational database** to eliminate redundancy and ensure data integrity. It divides large tables into smaller ones and establishes **relationships using foreign keys**.

Why is Normalization Important?

✅ **Advantages:**

- **Reduces Data Redundancy:** Prevents duplicate data storage.
- **Enhances Data Integrity:** Ensures consistency across tables.
- **Improves Query Performance:** Smaller, structured tables improve efficiency.
- **Easier Maintenance:** Updates affect fewer records.

⚠️ **Disadvantages:**

- **Complex Queries:** More JOINs may slow down query performance.
- **Increased Design Complexity:** Requires planning for relationships.

Normalization Forms

Normal Form	Description	Example
1NF (First Normal Form)	No duplicate rows; each cell holds a single value.	Remove repeating groups in tables.
2NF (Second Normal Form)	1NF + All non-key attributes depend on the primary key.	Remove partial dependencies.
3NF (Third Normal Form)	2NF + No transitive dependencies.	Remove columns that depend on non-primary key attributes.
BCNF (Boyce-Codd NF)	3NF + Every determinant is a candidate key.	Stronger version of 3NF.

✅ **Example of Normalization Process**

🔴 **Unnormalized Table (Repeating Data, 1NF Violation)**

Order_ID	Customer_Name	Product	Quantity
101	Alice	Laptop	1

Order_ID	Customer_Name	Product	Quantity
101	Alice	Mouse	2
102	Bob	Keyboard	1

🔵 **1NF (Atomic Values, Separate Rows for Each Product)**

Order_ID	Customer_Name	Product	Quantity
101	Alice	Laptop	1
101	Alice	Mouse	2
102	Bob	Keyboard	1

🔵 **2NF (Separate Customers and Orders Table, No Partial Dependencies)**

✅ **Customers Table**

Customer_ID	Customer_Name
1	Alice
2	Bob

✅ **Orders Table**

Order_ID	Customer_ID
101	1
102	2

✅ **Order_Details Table**

Order_ID	Product	Quantity
101	Laptop	1
101	Mouse	2
102	Keyboard	1

🔵 **3NF (Ensuring No Transitive Dependency)**

- If Customer_Name depends on Customer_ID, and Customer_ID is the primary key, then no transitive dependency exists.
- Now, Customers, Orders, and Order_Details are separate entities, **eliminating redundancy and improving data integrity**.

---

10. Describe the Different Normal Forms and Their Significance

Normalization is a **database design process** that reduces **data redundancy** and ensures **data integrity** by organizing tables efficiently. It is divided into different **Normal Forms (NFs)** based on increasing levels of optimization.

🔹 **1NF (First Normal Form)**

✅ **Rules:**

- Each column contains **atomic (indivisible) values** (no multiple values in a single cell).
- Each row is **unique** (has a primary key).

🔵 **Example (Violating 1NF)**

Order_ID	Customer_Name	Products
101	Alice	Laptop, Mouse
102	Bob	Keyboard

🔵 **Fix (Following 1NF)**

Order_ID	Customer_Name	Product
101	Alice	Laptop
101	Alice	Mouse
102	Bob	Keyboard

📌 **Significance:** Eliminates duplicate data and ensures each column has atomic values.

🔹 **2NF (Second Normal Form)**

✅ **Rules:**

- Must be in **1NF**.
- All **non-key attributes** should depend **entirely on the primary key** (no **partial dependencies**).

🔵 **Example (Violating 2NF)**

Order_ID (PK)	Customer_Name	Product	Product_Category
101	Alice	Laptop	Electronics
102	Bob	Keyboard	Electronics

📌 **Problem:** Product_Category depends on Product, not on Order_ID.

🔵 **Fix (Following 2NF)**

✅ **Orders Table**

Order_ID (PK)	Customer_Name
101	Alice
102	Bob

✅ **Products Table**

Product_ID (PK)	Product	Product_Category
P1	Laptop	Electronics
P2	Keyboard	Electronics

✅ **Order_Details Table (Bridge Table)**

Order_ID (FK)	Product_ID (FK)
101	P1
102	P2

📌 **Significance:** Eliminates **partial dependencies**, ensuring all attributes are fully dependent on the **entire primary key**.

🔹 **3NF (Third Normal Form)**

✅ **Rules:**

- Must be in **2NF**.
- **No transitive dependencies** (Non-key attributes should depend only on the primary key).

🔵 **Example (Violating 3NF)**

Employee_ID (PK)	Employee_Name	Department_ID	Department_Name
1	Alice	10	Sales
2	Bob	20	HR

📌 **Problem:** Department_Name depends on Department_ID, not directly on Employee_ID.

🔵 **Fix (Following 3NF)**

✅ **Employees Table**

Employee_ID (PK)	Employee_Name	Department_ID (FK)
1	Alice	10
2	Bob	20

✅ **Departments Table**

Department_ID (PK)	Department_Name
10	Sales
20	HR

📌 **Significance:** Removes **transitive dependencies**, ensuring **only direct dependencies exist**.

🔹 **BCNF (Boyce-Codd Normal Form)**

✅ **Rules:**

- Must be in **3NF**.
- Every **determinant** (attribute that determines another attribute) should be a **candidate key**.

🔵 **Example (Violating BCNF)**

Student_ID (PK)	Course	Instructor
101	Math	Prof. A
102	Science	Prof. B
103	Math	Prof. A

📌 **Problem:** Instructor depends on Course, not Student_ID.

🔵 **Fix (Following BCNF)**

✅ **Students Table**

Student_ID (PK)	Course_ID (FK)
101	C1
102	C2
103	C1

✅ **Courses Table**

Course_ID (PK)	Course	Instructor
C1	Math	Prof. A
C2	Science	Prof. B

📌 **Significance:** Ensures that **all functional dependencies are handled properly** by making **every determinant a candidate key**.

---

11. What is Denormalization, and When Would You Consider It?

🔹 **What is Denormalization?**

Denormalization is the process of **reintroducing redundancy** into a normalized database to **improve performance** by reducing JOINs.

🔹 **When to Use Denormalization?**

- **To speed up read-heavy queries** (reporting, dashboards).
- **To minimize complex JOINs**, especially in large datasets.
- **When real-time analysis requires faster access.**

✅ **Example: Before Denormalization (Normalized)**
Separate tables for Customers, Orders, and Products.

SELECT Customers.Name, Orders.Order_Date, Products.Product_Name

FROM Customers

JOIN Orders ON Customers.Customer_ID = Orders.Customer_ID

JOIN Products ON Orders.Product_ID = Products.Product_ID;

📌 **Issue:** Multiple JOINs slow down the query.

✅ **After Denormalization (Fewer Tables, Faster Reads)**
A single table containing all data.

SELECT Name, Order_Date, Product_Name

FROM Customers_Orders_Products;

📌 **Trade-off:** Faster read queries but higher **storage and update complexity**.

---

12. How Do You Handle NULL Values in SQL?

🔹 **What is a NULL Value?**

NULL represents **missing, undefined, or unknown data** in SQL.

🔹 **Handling NULL Values**

✅ **1. Checking for NULL (IS NULL or IS NOT NULL)**

SELECT * FROM Employees WHERE Manager_ID IS NULL;

📌 Finds employees **without a manager**.

✅ **2. Replacing NULL (COALESCE or IFNULL)**

```
SELECT Name, COALESCE(Phone, 'No Phone Number') AS Contact_Info FROM Customers;
```

📌 **Replaces NULL with a default value**.

✅ **3. Conditional Handling (CASE Statement)**

```
SELECT Name,
    CASE
      WHEN Salary IS NULL THEN 'Salary Not Assigned'
      ELSE Salary
    END AS Salary_Info
FROM Employees;
```

📌 **Assigns a custom message** when Salary is NULL.

✅ **4. Aggregation with NULL (COUNT, SUM, AVG)**

```
SELECT COUNT(*) FROM Employees WHERE Department IS NOT NULL;
```

📌 **Ignores NULL values in aggregations**.

✅ **5. Filtering NULL Values (NULLIF Function)**

```
SELECT NULLIF(Salary, 0) FROM Employees;
```

📌 **Returns NULL if Salary is 0**, useful to **avoid division by zero errors**.

🔹 **Summary**

Concept	Description	Key Benefit
Normalization	Organizing data to remove redundancy	Ensures consistency
Denormalization	Merging tables for performance	Faster read queries
Handling NULLs	Techniques to manage missing data	Prevents query errors

13. What Are Window Functions, and How Are They Used?

🔹 **What Are Window Functions?**

Window functions **perform calculations across a set of table rows related to the current row** without collapsing the rows into a single output (unlike aggregate functions).

🔹 **Key Components of Window Functions**

- **PARTITION BY** → Divides data into groups (optional).
- **ORDER BY** → Defines order within each partition (optional).
- **FRAME CLAUSE** → Defines the subset of rows for the calculation.

🔹 **Common Window Functions**

✅ **1. Ranking Functions**

- RANK(), DENSE_RANK(), ROW_NUMBER()

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

📌 **Ranks employees by salary within each department** (same rank if salary is the same).

✅ **2. Aggregate Window Functions**

- SUM(), AVG(), MIN(), MAX(), COUNT()

```
SELECT Employee_ID, Department, Salary,
    SUM(Salary) OVER (PARTITION BY Department) AS Total_Department_Salary
FROM Employees;
```

📌 **Calculates the total salary for each department without grouping rows**.

✅ **3. Running Totals and Moving Averages**

```
SELECT Order_ID, Customer_ID, Order_Amount,
    SUM(Order_Amount) OVER (ORDER BY Order_Date ROWS BETWEEN 4 PRECEDING
AND CURRENT ROW) AS Moving_Avg
```

FROM Orders;

📌 **Calculates a moving average over the last 5 rows**.

🚀 **Use Case:** Window functions are widely used in **analytics, reporting, and ranking** operations.

---

14. Explain the Difference Between UNION and UNION ALL

Feature	UNION	UNION ALL
Duplicates	Removes duplicates	Keeps duplicates
Performance	Slower (extra step to remove duplicates)	Faster (no duplicate check)
Use Case	When unique records are needed	When performance matters

✅ **Example: Using UNION (Removes Duplicates)**

```
SELECT Customer_ID FROM Customers_2024
UNION
SELECT Customer_ID FROM Customers_2023;
```

📌 **Only unique Customer_IDs appear**.

✅ **Example: Using UNION ALL (Keeps Duplicates)**

```
SELECT Customer_ID FROM Customers_2024
UNION ALL
SELECT Customer_ID FROM Customers_2023;
```

📌 **Duplicates are kept (faster performance)**.

🚀 **When to Use?**

- **Use UNION** if you need unique values.
- **Use UNION ALL** if performance is critical and duplicates are acceptable.

---

15. How Can You Optimize SQL Queries for Better Performance?

✅ **1. Use Proper Indexing**

- Create indexes on frequently searched columns.
- Avoid indexing columns with **low cardinality** (few unique values).

```
CREATE INDEX idx_customer ON Orders(Customer_ID);
```

📌 **Speeds up lookups on Customer_ID in Orders table.**

✅ **2. Use EXPLAIN to Analyze Query Execution**

- Helps identify slow joins, missing indexes, or performance bottlenecks.

```
EXPLAIN ANALYZE
SELECT * FROM Orders WHERE Order_Date > '2024-01-01';
```

📌 **Shows how SQL engine processes the query.**

✅ **3. Avoid SELECT ***

- Retrieve only necessary columns instead of fetching all data.

```
SELECT Order_ID, Customer_Name FROM Orders;
```

📌 **Reduces memory and improves performance.**

✅ **4. Use Joins Efficiently**

- Prefer **INNER JOIN** over **OUTER JOIN** if possible.
- Avoid **CROSS JOIN** unless needed.

```
SELECT Customers.Name, Orders.Order_Date
FROM Customers
INNER JOIN Orders ON Customers.Customer_ID = Orders.Customer_ID;
```

📌 **Optimized join fetching only necessary data.**

✅ **5. Use EXISTS Instead of IN for Subqueries**

```
SELECT * FROM Customers WHERE EXISTS (
  SELECT 1 FROM Orders WHERE Orders.Customer_ID = Customers.Customer_ID
);
```

📌 **EXISTS is more efficient than IN in large datasets**.

✅ **6. Use LIMIT and OFFSET for Pagination**

```
SELECT * FROM Orders ORDER BY Order_Date DESC LIMIT 10 OFFSET 20;
```

📌 **Fetches only required rows instead of loading the full dataset**.

✅ **7. Optimize GROUP BY and ORDER BY**

- Use **indexed columns** in GROUP BY.
- Avoid ordering large datasets unnecessarily.

```
SELECT Department, AVG(Salary)
FROM Employees
GROUP BY Department;
```

📌 **Groups data efficiently using indexed columns**.

16. What is a CTE (Common Table Expression), and How is it Used?

🔹 **What is a CTE?**

A **Common Table Expression (CTE)** is a **temporary named result set** that can be referenced within a SQL query. It improves readability, modularity, and performance when working with complex queries.

✅ **Syntax:**

```
WITH CTE_Name AS (
  SELECT Column1, Column2
  FROM Table_Name
  WHERE Condition
)
SELECT * FROM CTE_Name;
```

🔹 **Why Use CTEs?**

1. **Improves Query Readability** – Breaks down complex queries into smaller sections.

2. **Avoids Repetition** – Can be referenced multiple times in the query.
3. **Enhances Performance** – Optimizes recursive queries.
4. **Supports Recursion** – Useful for hierarchical data (e.g., organization structures).

✅ **Example 1: Simple CTE**

```
WITH High_Salary AS (
  SELECT Employee_ID, Name, Salary
  FROM Employees
  WHERE Salary > 70000
)
SELECT * FROM High_Salary;
```

📌 **Filters employees earning more than 70,000 and allows reuse of the result set.**

✅ **Example 2: Recursive CTE (Hierarchical Data)**

```
WITH EmployeeHierarchy AS (
  SELECT Employee_ID, Manager_ID, Name, 1 AS Level
  FROM Employees
  WHERE Manager_ID IS NULL
  UNION ALL
  SELECT e.Employee_ID, e.Manager_ID, e.Name, eh.Level + 1
  FROM Employees e
  JOIN EmployeeHierarchy eh ON e.Manager_ID = eh.Employee_ID
)
SELECT * FROM EmployeeHierarchy;
```

📌 **Fetches hierarchical reporting structure using recursion.**

---

17. Describe the ACID Properties in the Context of SQL Transactions

🔹 **What Are ACID Properties?**

ACID properties ensure database transactions are **reliable, consistent, and fault-tolerant**.

ACID Property	Description	Example
Atomicity	A transaction is **all or nothing** (either fully completed or rolled back).	If transferring money from **Account A → Account B**, both debit and credit must occur; otherwise, rollback.
Consistency	Database must remain **valid** before and after a transaction.	Foreign keys prevent inserting invalid references.
Isolation	Transactions execute **independently** without interfering.	Prevents race conditions in multi-user environments.
Durability	Committed changes **persist** even after system failures.	A power outage won't erase a committed transaction.

✅ **Example: ACID in Action**

```
BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE Account_ID = 1;
UPDATE Accounts SET Balance = Balance + 500 WHERE Account_ID = 2;

IF @@ERROR <> 0
   ROLLBACK;
ELSE
   COMMIT;
```

📌 **Ensures money is only transferred if both debit and credit operations succeed.**

---

18. What Are Stored Procedures, and What Are Their Advantages?

🔹 **What Is a Stored Procedure?**

A **Stored Procedure** is a **precompiled SQL script** that can be executed multiple times with parameters.

✅ **Syntax:**

```
CREATE PROCEDURE GetEmployeeDetails @EmpID INT
AS
BEGIN
  SELECT * FROM Employees WHERE Employee_ID = @EmpID;
END;
```

📌 **Executes predefined SQL logic when called.**

🔹 **Advantages of Stored Procedures**

1. **Performance Optimization** – Precompiled and executed faster.
2. **Code Reusability** – Can be used across multiple applications.
3. **Security** – Prevents SQL injection by using parameterized queries.
4. **Reduced Network Traffic** – Only procedure name & parameters are sent instead of full query.
5. **Encapsulation of Business Logic** – Centralizes complex logic within the database.

✅ **Example: Calling a Stored Procedure**

```
EXEC GetEmployeeDetails @EmpID = 101;
```

📌 **Fetches details of Employee ID 101.**

19. How Do You Implement Error Handling in SQL?

🔹 **Why Is Error Handling Important?**

Error handling ensures that SQL transactions are executed **safely and reliably**, preventing partial updates and maintaining database integrity.

🔹 **Error Handling Techniques in SQL Server**

✅ **1. Using TRY...CATCH** (SQL Server)

```
BEGIN TRY
  BEGIN TRANSACTION;

  UPDATE Accounts
  SET Balance = Balance - 500
  WHERE Account_ID = 1;

  UPDATE Accounts
  SET Balance = Balance + 500
  WHERE Account_ID = 2;

  COMMIT TRANSACTION;
END TRY
BEGIN CATCH
  ROLLBACK TRANSACTION;
  PRINT 'An error occurred: ' + ERROR_MESSAGE();
END CATCH;
```

📌 **Ensures that if any update fails, the entire transaction is rolled back.**

✅ **2. Using @@ERROR (Older SQL Versions)**

```
DECLARE @Error INT;

BEGIN TRANSACTION;
```

```
UPDATE Orders SET Quantity = Quantity - 1 WHERE Order_ID = 100;
SET @Error = @@ERROR;

IF @Error <> 0
BEGIN
   ROLLBACK TRANSACTION;
   PRINT 'Error: ' + CAST(@Error AS VARCHAR);
END
ELSE
   COMMIT TRANSACTION;
```

📌 **Checks for errors after each statement and rolls back if needed.**

✅ **3. Using RAISERROR for Custom Errors**

```
IF NOT EXISTS (SELECT 1 FROM Customers WHERE Customer_ID = 500)
   RAISERROR ('Customer ID not found!', 16, 1);
```

📌 **Raises a user-defined error when a condition is met.**

🚀 **Best Practices:** Always use transactions, try-catch blocks, and custom error messages for robustness.

---

20. What Is the Difference Between DELETE and TRUNCATE Commands?

Feature	DELETE	TRUNCATE
Operation Type	DML (Data Manipulation Language)	DDL (Data Definition Language)

Feature	DELETE	TRUNCATE
Removes Specific Rows?	Yes, with WHERE condition	No, removes all rows
Rollback Possible?	Yes, supports ROLLBACK	No rollback (unless inside a transaction)
Resets Identity Column?	No	Yes (resets AUTO_INCREMENT)
Performance	Slower (logs each row deletion)	Faster (minimal logging)

✅ **Example: DELETE with Condition**

DELETE FROM Employees WHERE Department = 'HR';

📌 **Deletes only employees in the HR department.**

✅ **Example: TRUNCATE (Removes All Data)**

TRUNCATE TABLE Employees;

📌 **Deletes all records and resets identity values.**

🚀 **When to Use?**

- Use **DELETE** when you need selective deletions.
- Use **TRUNCATE** for faster, complete table clearing.

---

21. Explain the Concept of Database Views and Their Use Cases

🔹 **What Is a View?**

A **view** is a **virtual table** based on a SQL query, providing a simplified representation of data without storing it physically.

✅ **Syntax:**

CREATE VIEW Employee_View AS

SELECT Employee_ID, Name, Department, Salary

FROM Employees

WHERE Salary > 60000;

📌 **Filters employees earning more than 60,000.**

🔹 **Why Use Views?**

1. **Encapsulation** – Hides complex SQL logic.
2. **Security** – Restricts access to sensitive columns.
3. **Reusability** – Used in multiple queries without rewriting logic.
4. **Data Consistency** – Ensures the latest data is always available.

✅ **Example: Querying a View**

SELECT * FROM Employee_View;

📌 **Returns employees earning more than 60,000 dynamically.**

🔹 **Types of Views**

- **Simple View** – Based on a single table (no aggregation).
- **Complex View** – Uses joins, aggregates, or subqueries.
- **Updatable View** – Allows INSERT, UPDATE, DELETE if based on a single table.
- **Read-Only View** – Prevents modifications if based on multiple tables.

22. How Do You Perform Pattern Matching in SQL?

🔹 **Pattern Matching in SQL with LIKE Operator**

The **LIKE** operator is used to search for a specific pattern in a text column.

✅ **Wildcard Characters in LIKE**

Symbol	Meaning	Example
%	Matches **any number** of characters (including none)	'A%' → Starts with A
_	Matches **exactly one** character	'A_' → A followed by any one character

Symbol	Meaning	Example
[ ]	Matches any character inside brackets	'[A-C]%' → Starts with A, B, or C
[^]	Matches any character **not** inside brackets	'[^A]%' → Doesn't start with A

✅ **Example 1: Find Names Starting with 'J'**

```
SELECT * FROM Employees WHERE Name LIKE 'J%';
```

📌 **Finds employees whose names start with 'J'.**

✅ **Example 2: Find Emails Ending with 'gmail.com'**

```
SELECT * FROM Customers WHERE Email LIKE '%@gmail.com';
```

📌 **Fetches all Gmail users.**

✅ **Example 3: Find 4-Letter Names Starting with 'A'**

```
SELECT * FROM Students WHERE Name LIKE 'A___';
```

📌 **Finds names that start with 'A' and have exactly 4 letters.**

🚀 **Advanced:** Use **REGEXP** in MySQL for complex patterns.

```
SELECT * FROM Employees WHERE Name REGEXP '^A.*son$';
```

📌 **Finds names that start with 'A' and end with 'son'.**

---

23. What Are Aggregate Functions, and Can You Provide Examples?

🔹 **What Are Aggregate Functions?**

Aggregate functions **perform calculations on a group of rows** and return a single value.

✅ **Common Aggregate Functions**

Function	Description	Example
COUNT()	Counts rows	COUNT(*) → Total rows

Function	Description	Example
SUM()	Adds up values	SUM(Salary) → Total salary
AVG()	Finds average value	AVG(Salary) → Avg salary
MAX()	Finds maximum value	MAX(Salary) → Highest salary
MIN()	Finds minimum value	MIN(Salary) → Lowest salary

✅ **Example 1: Count Employees in Each Department**

```
SELECT Department, COUNT(*) AS Total_Employees
FROM Employees
GROUP BY Department;
```

📌 **Counts employees per department.**

✅ **Example 2: Calculate Average Salary per Job Role**

```
SELECT Job_Title, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY Job_Title;
```

📌 **Finds average salary for each job role.**

✅ **Example 3: Find Department with Highest Salary**

```
SELECT Department, MAX(Salary) AS Highest_Salary
FROM Employees
GROUP BY Department;
```

📌 **Fetches the highest salary in each department.**

🚀 **Best Practices:** Always use GROUP BY with aggregate functions to group data logically.

---

24. Describe the Process of Importing and Exporting Data in SQL

🔹 **Importing Data into SQL**

1. **Using LOAD DATA INFILE (MySQL)**

```
LOAD DATA INFILE '/path/to/file.csv'
INTO TABLE Employees
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

📌 **Imports CSV data into the Employees table.**

2. **Using BULK INSERT (SQL Server)**

```
BULK INSERT Employees
FROM 'C:\data\employees.csv'
WITH (FORMAT = 'CSV', FIRSTROW = 2);
```

📌 **Loads large data efficiently.**

3. **Using IMPORT WIZARD (SQL Server, MySQL, PostgreSQL)**
   - 📌 **Step-by-step UI-based data import for beginners.**

🔹 **Exporting Data from SQL**

1. **Using SELECT INTO OUTFILE (MySQL)**

```
SELECT * FROM Employees
INTO OUTFILE '/path/to/export.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';
```

📌 **Exports table data to a CSV file.**

2. **Using bcp Command (SQL Server)**

```
bcp Employees out "C:\data\employees.csv" -c -t, -T -S ServerName
```

📌 **Exports data from SQL Server in CLI.**

3. **Using EXPORT WIZARD (SQL Server, MySQL, PostgreSQL)**
   📌 **Graphical interface for data export.**

🚀 **Best Practices:**

- Always **validate** imported data for accuracy.
- Ensure **data formats match** before import/export.
- Use INDEXING after import to **boost performance**.

22. How Do You Perform Pattern Matching in SQL?

Pattern matching in SQL is done using the **LIKE** operator and **wildcard characters** to search for specific patterns in string columns.

✅ **Wildcard Characters in LIKE Operator**

Wildcard	Description	Example
%	Matches **any sequence** of characters (including none)	'A%' → Starts with A
_	Matches **exactly one** character	'A_' → A followed by any one character
[ ]	Matches **any character** inside brackets	'[A-C]%' → Starts with A, B, or C
[^]	Matches any character **not** inside brackets	'[^A]%' → Doesn't start with A

✅ **Example 1: Find Customers Whose Name Starts with 'J'**

```
SELECT * FROM Customers WHERE Name LIKE 'J%';
```

📌 **Finds names like "John", "James", "Jennifer".**

✅ **Example 2: Find Emails Ending with 'gmail.com'**

```
SELECT * FROM Users WHERE Email LIKE '%@gmail.com';
```

📌 **Fetches all Gmail users.**

✅ **Example 3: Find 5-Letter Names Starting with 'A'**

```
SELECT * FROM Employees WHERE Name LIKE 'A____';
```

📌 **Finds names like "Alice", "Abdul".**

✅ **Advanced Pattern Matching Using REGEXP (MySQL & PostgreSQL)**

SELECT * FROM Employees WHERE Name REGEXP '^A.*son$';

📌 **Finds names that start with 'A' and end with 'son'.**

🚀 **Use Case:** When filtering large datasets for customer names, email domains, or product codes.

---

23. What Are Aggregate Functions, and Can You Provide Examples?

🔹 **What Are Aggregate Functions?**

Aggregate functions **compute a single value** from a set of values. They are commonly used with GROUP BY.

✅ **Common Aggregate Functions**

Function	Description	Example
COUNT()	Counts the number of rows	COUNT(*) → Total rows
SUM()	Adds values	SUM(Sales) → Total sales
AVG()	Finds the average	AVG(Salary) → Avg salary
MAX()	Returns max value	MAX(Salary) → Highest salary
MIN()	Returns min value	MIN(Salary) → Lowest salary

✅ **Example 1: Count Employees in Each Department**

SELECT Department, COUNT(*) AS Total_Employees

FROM Employees

GROUP BY Department;

📌 **Counts employees per department.**

✅ **Example 2: Calculate Average Salary per Job Role**

```
SELECT Job_Title, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY Job_Title;
```

📌 **Finds average salary for each job role.**

✅ **Example 3: Find Department with Highest Salary**

```
SELECT Department, MAX(Salary) AS Highest_Salary
FROM Employees
GROUP BY Department;
```

📌 **Fetches the highest salary in each department.**

🚀 **Use Case:** Helps in **sales analysis**, **performance metrics**, and **financial reporting**.

---

24. Describe the Process of Importing and Exporting Data in SQL

🔹 **Importing Data into SQL**

1 .**Using LOAD DATA INFILE (MySQL)**

```
LOAD DATA INFILE '/path/to/file.csv'
INTO TABLE Employees
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

📌 **Efficient for importing large datasets.**

2 .**Using BULK INSERT (SQL Server)**

```
BULK INSERT Employees
FROM 'C:\data\employees.csv'
WITH (FORMAT = 'CSV', FIRSTROW = 2);
```

📌 **Best for structured file imports.**

3 .**Using IMPORT WIZARD (SQL Server, MySQL, PostgreSQL)**

📌 **Graphical UI-based data import method.**

🔹 **Exporting Data from SQL**

1 .**Using SELECT INTO OUTFILE (MySQL)**

```
SELECT * FROM Employees
INTO OUTFILE '/path/to/export.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';
```

📌 **Exports table data to CSV.**

2 .**Using bcp Command (SQL Server)**

```
bcp Employees out "C:\data\employees.csv" -c -t, -T -S ServerName
```

📌 **Exports data from SQL Server via CLI.**

3 .**Using EXPORT WIZARD (SQL Server, MySQL, PostgreSQL)**

📌 **Graphical interface for exporting data.**

🚀 **Best Practices:**

✓ **Validate data** before importing/exporting.

✓ Use **indexing** after import for **performance optimization**.

✓ **Back up databases** before large-scale imports.

28. Explain the Concept of Transactions and Their Importance

🔹 **What is a Transaction in SQL?**

A **transaction** is a sequence of SQL operations that are executed as a **single unit of work**. A transaction ensures that either **all operations are completed successfully**, or **none of them are applied** to maintain database integrity.

✅ **Transactions Follow ACID Properties:**

Property	Description
Atomicity	Ensures **all operations** in a transaction succeed or none at all (all-or-nothing).
Consistency	Ensures database remains in a **valid state** before and after the transaction.
Isolation	Ensures transactions don't **interfere** with each other.
Durability	Ensures committed transactions are **permanently saved** even if system crashes.

🔹 **How to Use Transactions in SQL?**

Most SQL databases support transactions using the following commands:

1 .**Begin Transaction** – Starts a transaction.

2 .**Commit** – Saves all changes permanently.

3 .**Rollback** – Undoes all changes if something goes wrong.

✅ **Example: Transferring Money Between Two Accounts**

```
START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE Account_ID = 101;

UPDATE Accounts SET Balance = Balance + 500 WHERE Account_ID = 102;

COMMIT;  -- Finalize transaction if no errors
```

📌 **Ensures money is deducted from one account and added to another only if both operations succeed.**

✅ **Using ROLLBACK in Case of Errors**

```
START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE Account_ID = 101;

UPDATE Accounts SET Balance = Balance + 500 WHERE Account_ID = 102;
```

```
-- If an error occurs, rollback
IF @@ERROR <> 0
   ROLLBACK;
ELSE
   COMMIT;
```

📌 **Ensures no partial money transfer happens if any error occurs.**

🚀 **Why Are Transactions Important?**

✓ Prevents **data corruption** due to partial updates.

✓ Ensures **financial accuracy** in banking apps.

✓ Avoids **data inconsistencies** in multi-step operations.

---

29. How Do You Manage User Permissions and Security in SQL?

🔹 **Why is SQL Security Important?**

SQL security ensures **only authorized users** can access or modify the database, protecting **sensitive data** from unauthorized access or breaches.

🔹 **Managing User Permissions in SQL**

✅ **1. Creating Users**

```
CREATE USER 'john_doe'@'localhost' IDENTIFIED BY 'SecurePassword';
```

📌 **Creates a new database user with a password.**

✅ **2. Granting Privileges**

```
GRANT SELECT, INSERT ON SalesDB.* TO 'john_doe'@'localhost';
```

📌 **Allows user to view and insert data in the SalesDB database.**

✅ **3. Revoking Privileges**

```
REVOKE INSERT ON SalesDB.* FROM 'john_doe'@'localhost';
```

📌 **Removes insert permission from the user.**

✅ **4. Checking User Permissions**

```
SHOW GRANTS FOR 'john_doe'@'localhost';
```

📌 **Displays all permissions assigned to a user.**

✅ **5. Removing Users**

```
DROP USER 'john_doe'@'localhost';
```

📌 **Deletes a user from the database.**

🔹 **Additional Security Measures**

✓ Use **ROLE-BASED ACCESS CONTROL (RBAC)** for **granular permission management**.
✓ **Encrypt** sensitive data (e.g., passwords, credit card info).
✓ **Regularly audit** and monitor database access logs.

🚀 **Real-world Use Case:** Protects **financial records**, **customer information**, and **company data** from unauthorized modifications.

---

30. Can You Provide an Example of a Complex SQL Query You've Written and Explain Its Components?

🔹 **Complex Query: Finding Top 3 Highest Revenue Products by Category**

```
WITH RankedProducts AS (
  SELECT
    p.Category,
    p.ProductName,
    SUM(s.Quantity * s.Price) AS TotalRevenue,
    RANK() OVER (PARTITION BY p.Category ORDER BY SUM(s.Quantity * s.Price) DESC) AS Rank
  FROM Sales s
  JOIN Products p ON s.ProductID = p.ProductID
  WHERE s.SaleDate BETWEEN '2024-01-01' AND '2024-12-31'
```

```
    GROUP BY p.Category, p.ProductName
)
SELECT Category, ProductName, TotalRevenue
FROM RankedProducts
WHERE Rank <= 3
ORDER BY Category, Rank;
```

🔍 **Explanation of Components**

Component	Explanation
WITH RankedProducts AS (...)	Defines a **Common Table Expression (CTE)** for ranking products.
SUM(s.Quantity * s.Price) AS TotalRevenue	Calculates total revenue for each product.
RANK() OVER (PARTITION BY p.Category ORDER BY SUM(s.Quantity * s.Price) DESC) AS Rank	Ranks products within each category based on revenue.
WHERE s.SaleDate BETWEEN '2024-01-01' AND '2024-12-31'	Filters sales data for the year 2024.
GROUP BY p.Category, p.ProductName	Groups data by product and category for aggregation.
WHERE Rank <= 3	Retrieves only the **top 3 highest revenue products** per category.

🚀 **Use Case:**
This query is used in **e-commerce analytics** to identify **top-selling products per category**.