SQL 50 Interview Questions – Zero Analyst

1. What is the purpose of the GROUP BY clause in SQL? Provide an example.

The **GROUP BY** clause in SQL is used to group rows that have the same values in specified columns into aggregate data. This is commonly used with aggregate functions like COUNT(), SUM(), AVG(), MAX(), and MIN() to perform calculations on each group of data.

Example: Suppose we have a table Orders with columns CustomerID and OrderAmount. We want to find the total amount of orders placed by each customer.

```
SELECT CustomerID, SUM(OrderAmount) AS TotalOrderAmount

FROM Orders

GROUP BY CustomerID;
```

This query groups the data by CustomerID and calculates the sum of OrderAmount for each customer.

2. Explain the difference between an INNER JOIN and a LEFT JOIN with examples.

An **INNER JOIN** returns only the rows that have matching values in both tables. A **LEFT JOIN** (or LEFT OUTER JOIN) returns all the rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for columns from the right table.

Example of INNER JOIN: Suppose we have two tables, Customers and Orders. We want to retrieve all customers who have placed an order.

```
SELECT Customers.CustomerID, Customers.CustomerName, Orders.OrderID

FROM Customers

INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

This query returns only the customers who have placed at least one order.

Example of LEFT JOIN: Now, if we want to retrieve all customers, along with their orders (if any), we use a LEFT JOIN.

```
SELECT Customers.CustomerID, Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

This query returns all customers, and for those who have not placed any orders, the OrderID will be NULL.

3. Discuss the role of the WHERE clause in SQL queries and provide examples of its usage.

The **WHERE** clause in SQL is used to filter records that meet specific conditions. It is typically used in SELECT, UPDATE, DELETE, and INSERT INTO statements to specify which records should be affected.

Example of WHERE clause in a SELECT statement: Suppose we have a table Employees and we want to find all employees who work in the "Sales" department.

```
SELECT EmployeeID, EmployeeName, Department

FROM Employees

WHERE Department = 'Sales';
```

This query returns only the employees whose department is "Sales."

Example of WHERE clause in a DELETE statement: Suppose we want to delete all orders that were placed before January 1, 2023, from the Orders table.

```
DELETE FROM Orders

WHERE OrderDate < '2023-01-01';
```

This query deletes only the orders placed before the specified date.

4. Explain the concept of database transactions and the ACID properties.

A **database transaction** is a sequence of operations performed as a single logical unit of work. A transaction must be either fully completed or fully rolled back. The ACID properties ensure the reliability of database transactions.

- **Atomicity:** Ensures that all operations within a transaction are completed; if any part of the transaction fails, the entire transaction is rolled back.
- **Consistency:** Ensures that a transaction brings the database from one valid state to another, maintaining database invariants.
- **Isolation:** Ensures that concurrently executed transactions do not affect each other's execution.
- **Durability:** Ensures that once a transaction is committed, it remains so, even in the case of a system failure.

Example: Consider a banking system where a transfer of funds from one account to another is a transaction. The operations include debiting one account and crediting another. If any of these operations fail, the entire transaction should be rolled back to maintain consistency.

```
BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;

UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;

COMMIT;
```

If any of these UPDATE operations fail, the transaction is rolled back to its initial state.

5. Describe the benefits of using subqueries in SQL and provide a scenario where they would be useful.

A **subquery** (or inner query) is a query nested inside another query. Subqueries are useful for breaking down complex queries, improving readability, and sometimes improving performance by allowing the filtering of results at an earlier stage.

Scenario: Suppose we have a table Employees and we want to find all employees who earn more than the average salary of their department.

```
SELECT EmployeeID, EmployeeName, Salary

FROM Employees

WHERE Salary > (

  SELECT AVG(Salary)

  FROM Employees

  WHERE Department = Employees.Department

);
```

In this example, the subquery calculates the average salary for each department, and the outer query selects employees whose salary is greater than this average.

6. Discuss the differences between the CHAR and VARCHAR data types in SQL.

The **CHAR** and **VARCHAR** data types in SQL are used to store character strings, but they differ in how they store and manage the data.

- **CHAR:** Fixed-length character data type. If the data stored is shorter than the specified length, it will be padded with spaces to reach the defined length.
- **VARCHAR:** Variable-length character data type. It only uses as much space as needed to store the actual data, without padding.

Example: Suppose we define a CHAR(10) and a VARCHAR(10) field.

```
CREATE TABLE ExampleTable (

  FixedLengthField CHAR(10),
```

```
    VariableLengthField VARCHAR(10)
);


INSERT INTO ExampleTable (FixedLengthField, VariableLengthField)
VALUES ('Test', 'Test');
```

In this case, FixedLengthField will store "Test" as "Test " (with 6 spaces), while VariableLengthField will store "Test" as "Test" without extra spaces.

7. Explain the purpose of the ORDER BY clause in SQL queries and provide examples.

The **ORDER BY** clause in SQL is used to sort the result set of a query by one or more columns. Sorting can be done in ascending (ASC) or descending (DESC) order.

Example: Suppose we have a table Employees and we want to list all employees sorted by their Salary in descending order.

```
SELECT EmployeeID, EmployeeName, Salary
FROM Employees
ORDER BY Salary DESC;
```

This query returns the list of employees sorted by salary from highest to lowest.

You can also sort by multiple columns:

```
SELECT EmployeeID, EmployeeName, Department, Salary
FROM Employees
ORDER BY Department ASC, Salary DESC;
```

This query sorts employees first by department (ascending) and then by salary (descending) within each department.

8. Describe the importance of data integrity constraints such as NOT NULL, UNIQUE, and CHECK constraints in SQL databases.

Data integrity constraints in SQL ensure the accuracy and consistency of data within a database. The most common constraints include:

- **NOT NULL:** Ensures that a column cannot contain a NULL value, enforcing mandatory data entry.
- **UNIQUE:** Ensures that all values in a column are unique, preventing duplicate entries.
- **CHECK:** Ensures that all values in a column meet a specific condition or criteria.

Example: Suppose we have a table Products with constraints:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100) NOT NULL,
    Price DECIMAL(10, 2) CHECK (Price > 0),
    SKU VARCHAR(50) UNIQUE
);
```

In this table, ProductName cannot be NULL, Price must be greater than 0, and SKU must be unique.

9. Discuss the advantages and disadvantages of using stored procedures.

Advantages of Stored Procedures:

- **Performance:** Stored procedures are precompiled and optimized, reducing the need for parsing and execution time.
- **Reusability:** They can be reused across different applications or parts of an application, promoting code reuse.

- **Security:** They provide an additional layer of security by encapsulating the SQL code, reducing the risk of SQL injection.
- **Maintainability:** Centralized logic in stored procedures makes it easier to maintain and update database-related logic.

Disadvantages of Stored Procedures:

- **Complexity:** Stored procedures can become complex and hard to debug, especially as they grow in size and logic.
- **Portability:** They are often specific to a database management system, making it harder to migrate to a different system.
- **Versioning:** Managing versions of stored procedures can be challenging, especially in large projects with multiple developers.

10. How can you handle NULL values in SQL?

NULL values in SQL represent the absence of a value. Handling NULL values effectively is crucial to avoid unexpected results in queries.

- **IS NULL / IS NOT NULL:** Used to filter records with or without NULL values.
- **COALESCE:** Returns the first non-NULL value in a list of expressions.
- **IFNULL / ISNULL:** Database-specific functions that return a specified value if the expression is NULL.
- **NVL:** A function used in Oracle to replace NULL with a specified value.

Example: Suppose we want to display employee names and their bonuses. If the bonus is NULL, we want to show it as 0.

```
SELECT EmployeeName, COALESCE(Bonus, 0) AS Bonus
FROM Employees;
```

This query uses COALESCE to replace any NULL values in the Bonus column with 0.

11. Discuss the role of the COMMIT and ROLLBACK statements in SQL transactions.

In SQL, **COMMIT** and **ROLLBACK** are used to manage transactions:

- **COMMIT:** Saves all changes made during the current transaction to the database. Once committed, the changes are permanent and visible to other users.
- **ROLLBACK:** Undoes all changes made during the current transaction, reverting the database to its previous state before the transaction began.

Example: Consider a banking system where you are transferring money from one account to another. If any step fails, you can roll back the entire transaction.

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 1;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 2;

-- If both updates succeed, commit the transaction
COMMIT;

-- If any update fails, roll back the transaction
ROLLBACK;
```

12. Explain the purpose of the LIKE operator in SQL and provide examples of its usage.

The **LIKE** operator in SQL is used to search for a specified pattern in a column. It is commonly used with the WHERE clause to filter records.

Example: Suppose we want to find all employees whose names start with "J":

```
SELECT EmployeeName

FROM Employees

WHERE EmployeeName LIKE 'J%';
```

This query returns all employee names that start with the letter "J".
The % wildcard represents any sequence of characters.

Another example is finding names that end with "son":

```
SELECT EmployeeName

FROM Employees

WHERE EmployeeName LIKE '%son';
```

This query returns all employee names that end with "son".

13. Describe the concept of normalization forms (1NF, 2NF, 3NF) and why they are important in database design.

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. The main normalization forms are:

- **1NF (First Normal Form):** Ensures that the table has a primary key and that all columns contain atomic (indivisible) values.
- **2NF (Second Normal Form):** Achieved when a table is in 1NF and all non-key columns are fully dependent on the primary key, eliminating partial dependencies.
- **3NF (Third Normal Form):** Achieved when a table is in 2NF and all columns are dependent only on the primary key, eliminating transitive dependencies.

Importance: Normalization helps in eliminating data redundancy, ensuring data consistency, and simplifying database maintenance.

14. Discuss the differences between a clustered and non-clustered index in SQL.

Clustered Index: A clustered index determines the physical order of data in a table. There can be only one clustered index per table, as the data rows themselves are stored in the index order.

Non-Clustered Index: A non-clustered index is a separate structure that points to the data in the table. It does not alter the physical order of the data and can be created in multiple numbers on a single table.

Example: Creating a clustered index on a primary key:

```
CREATE CLUSTERED INDEX IDX_EmployeeID

ON Employees(EmployeeID);
```

Creating a non-clustered index on the LastName column:

```
CREATE NONCLUSTERED INDEX IDX_LastName

ON Employees(LastName);
```

15. Explain the concept of data warehousing and how it differs from traditional relational databases.

A **Data Warehouse** is a specialized system used for storing, retrieving, and analyzing large volumes of historical data from various sources. It is designed for query and analysis rather than transaction processing.

Key Differences:

- **Purpose:** Traditional relational databases are optimized for day-to-day transaction processing (OLTP), while data warehouses are optimized for data analysis and reporting (OLAP).
- **Data Structure:** Relational databases typically store normalized data, whereas data warehouses often use denormalized or star/snowflake schemas for faster query performance.
- **Data Volume:** Data warehouses are designed to handle much larger volumes of data than traditional databases.
- **Time Horizon:** Data warehouses store historical data for analysis over long periods, while relational databases often store current, real-time data.

Example: A retail company might use a data warehouse to analyze customer purchasing patterns over several years, helping in decision-making and strategy planning.

16. Describe the benefits of using database triggers and provide examples of their usage.

Database triggers are special stored procedures that automatically execute in response to certain events on a particular table or view. They are useful for enforcing business rules, validating data, and auditing changes.

Benefits of using triggers:

- **Automatic Execution:** Triggers automatically respond to specified events such as INSERT, UPDATE, or DELETE, ensuring certain actions are taken without manual intervention.
- **Data Integrity:** Triggers can enforce complex integrity constraints that might not be possible with standard SQL constraints alone.
- **Auditing:** Triggers can log changes to the database, providing an audit trail of who made changes and when.
- **Consistency:** They help maintain consistent and valid data across the database by ensuring related actions are taken when certain changes occur.

Example: Suppose we want to automatically update a LastModified column whenever a row in the Employees table is updated:

```
CREATE TRIGGER trg_UpdateLastModified

ON Employees

AFTER UPDATE

AS

BEGIN

   UPDATE Employees

   SET LastModified = GETDATE()

   WHERE EmployeeID IN (SELECT DISTINCT EmployeeID FROM inserted);

END;
```

17. Discuss the concept of database concurrency control and how it is achieved in SQL databases.

Concurrency control in databases is a mechanism to manage simultaneous operations without conflicting with each other, ensuring data consistency and integrity in a multi-user environment.

Concurrency control is achieved through:

- **Locking:** Locks are placed on data being accessed by a transaction, preventing other transactions from making conflicting changes.
- **Isolation Levels:** SQL provides different isolation levels (e.g., Read Uncommitted, Read Committed, Repeatable Read, Serializable) that determine the level of visibility one transaction has over the data changes made by another transaction.

- **Optimistic Concurrency:** Assumes that conflicts are rare and checks for conflicts only at the time of committing a transaction.
- **Pessimistic Concurrency:** Assumes that conflicts are likely and locks data when a transaction starts to ensure no other transactions can alter it.

18. Explain the role of the SELECT INTO statement in SQL and provide examples of its usage.

The **SELECT INTO** statement in SQL is used to create a new table and insert the result set of a query into it. It's commonly used for making backups of tables or for creating temporary tables to hold query results.

Example: Creating a backup of the Employees table:

```
SELECT * INTO EmployeesBackup
FROM Employees;
```

This query creates a new table called EmployeesBackup and populates it with all the data from the Employees table.

19. What is the difference between a correlated and a non-correlated subquery?

A **correlated subquery** is a subquery that references columns from the outer query. It is executed once for each row processed by the outer query, making it dependent on the outer query.

A **non-correlated subquery** is independent of the outer query and can be executed separately. It is evaluated once, and the result is used by the outer query.

Example of a correlated subquery:

```
SELECT e1.EmployeeName
FROM Employees e1
WHERE e1.Salary > (SELECT AVG(e2.Salary) FROM Employees e2 WHERE e2.DepartmentID = e1.DepartmentID);
```

This query selects employees whose salary is above the average salary in their respective departments.

Example of a non-correlated subquery:

```
SELECT EmployeeName
FROM Employees
WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'HR');
```

This query selects employees who work in the HR department.

20. How do you perform data migration in SQL?

Data migration in SQL involves transferring data from one database to another or between different environments (e.g., development to production). It can be done using various methods:

- **Using INSERT INTO ... SELECT:** Copy data from one table to another, possibly in a different database.
- **Using data export/import tools:** Many database management systems provide tools for exporting data to a file (e.g., CSV) and importing it into another database.
- **Using ETL (Extract, Transform, Load) tools:** Specialized tools designed for complex data migrations, often involving data transformation.

Example of simple data migration:

```
INSERT INTO ProductionDB.Employees (EmployeeID, EmployeeName, Salary)
SELECT EmployeeID, EmployeeName, Salary
FROM DevelopmentDB.Employees;
```

This query migrates data from the Employees table in the Development database to the Production database.

21. Describe the differences between a database view and a materialized view in SQL.

View: A database view is a virtual table that is created by a SQL query. It does not store data physically; instead, it fetches the data from the underlying tables whenever the view is queried. Views are typically used to simplify complex queries, provide security by restricting access to certain columns or rows, and present data in a specific format.

Materialized View: A materialized view, on the other hand, stores the result of the query physically on disk. It is essentially a snapshot of the data at a specific point in time and can be refreshed periodically. Materialized views are used to improve query performance, especially in cases where the underlying data is large and complex, and frequent querying is needed.

Key Differences:

- **Storage:** Views do not store data, while materialized views do.
- **Performance:** Queries on materialized views are faster since the data is precomputed and stored, while views require real-time computation.
- **Refresh:** Materialized views can be refreshed to reflect changes in the underlying data, whereas views always reflect the latest data.

22. Discuss the advantages of using parameterized queries in SQL applications.

Parameterized queries are SQL queries in which placeholders are used for parameters, and the actual values are supplied at runtime. This approach has several advantages:

- **Prevents SQL Injection:** By separating the SQL logic from the data, parameterized queries prevent attackers from injecting malicious SQL code.
- **Improves Performance:** Parameterized queries allow the database to cache and reuse query execution plans, leading to better performance.
- **Increases Code Reusability:** The same query can be reused with different parameter values, reducing code duplication.
- **Enhances Maintainability:** Parameterized queries make it easier to manage and debug SQL code, as the logic is clear and separated from the data.

23. Write a query to retrieve all employees who have a salary greater than $100,000.

Example Query:

```
SELECT EmployeeID, EmployeeName, Salary

FROM Employees

WHERE Salary > 100000;
```

This query retrieves the EmployeeID, EmployeeName, and Salary of all employees who earn more than $100,000.

24. Create a query to display the total number of orders placed in the last month.

Example Query:

```
SELECT COUNT(*) AS TotalOrders

FROM Orders

WHERE OrderDate >= DATEADD(MONTH, -1, GETDATE())
```

AND OrderDate < GETDATE();

This query counts the total number of orders placed in the last month.

25. Write a query to find the average order value for each customer.

Example Query:

SELECT CustomerID, AVG(OrderTotal) AS AverageOrderValue

FROM Orders

GROUP BY CustomerID;

This query calculates the average order value for each customer by grouping the orders by CustomerID.

26. Create a query to count the number of distinct products sold in the past week.

Example Query:

SELECT COUNT(DISTINCT ProductID) AS DistinctProductsSold

FROM Sales

WHERE SaleDate >= DATEADD(WEEK, -1, GETDATE());

This query counts the number of distinct products that were sold in the past week.

27. What is SQL and its significance in data analysis?

SQL (Structured Query Language) is a standard programming language used to manage and manipulate relational databases. It allows users to query, insert, update, delete, and retrieve data stored in a database. SQL is significant in data analysis because:

- **Data Retrieval:** SQL allows analysts to extract specific data from large datasets efficiently.
- **Data Manipulation:** It provides tools to modify data, such as updating records or deleting unwanted information.
- **Aggregation and Calculation:** SQL offers powerful functions for summarizing data, such as calculating averages, totals, and counts.
- **Data Integration:** SQL can join data from multiple tables, enabling complex data analysis across different datasets.
- **Scalability:** SQL queries can handle large datasets, making it a valuable tool for big data analysis.

28. Differentiate between SQL and MySQL.

SQL (Structured Query Language) is a standardized language used to manage and manipulate relational databases. It is a language that all relational database management systems (RDBMS) understand, and it defines how to interact with the data stored in databases.

MySQL, on the other hand, is an open-source relational database management system (RDBMS) that uses SQL as its language for querying and managing databases. It is a software application used to store, retrieve, and manage data using SQL commands.

Key Differences:

- **SQL:** A language used to interact with relational databases.
- **MySQL:** A database management system that uses SQL to perform operations on its data.
- **SQL:** Is a standard, and many different RDBMS like MySQL, PostgreSQL, and SQL Server implement it.
- **MySQL:** Is one of the many implementations of the SQL standard, with specific features and optimizations.

29. Explain the difference between SQL joins: INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN.

SQL joins are used to combine rows from two or more tables based on a related column between them.

- **INNER JOIN:** Returns only the rows that have matching values in both tables. If there is no match, the row is excluded.
- **LEFT JOIN (or LEFT OUTER JOIN):** Returns all rows from the left table and the matching rows from the right table. If there is no match, the result is NULL on the side of the right table.
- **RIGHT JOIN (or RIGHT OUTER JOIN):** Returns all rows from the right table and the matching rows from the left table. If there is no match, the result is NULL on the side of the left table.
- **FULL JOIN (or FULL OUTER JOIN):** Returns all rows when there is a match in either the left or right table. Rows with no match in either table will have NULL values in the columns from the table without the match.

Example Queries:

```
-- INNER JOIN
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;


-- LEFT JOIN
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
LEFT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

```
-- RIGHT JOIN
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
RIGHT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;


-- FULL JOIN
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
FULL OUTER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

30. What are the primary components of a SQL query?

The primary components of a SQL query include:

- **SELECT:** Specifies the columns to be retrieved from the database.
- **FROM:** Indicates the table(s) from which the data should be fetched.
- **WHERE:** Filters the rows based on specified conditions.
- **GROUP BY:** Groups the result set by one or more columns.
- **HAVING:** Filters the grouped rows based on a condition, typically used with aggregate functions.
- **ORDER BY:** Sorts the result set by one or more columns.
- **JOIN:** Combines rows from two or more tables based on a related column between them.

These components are often combined to create powerful queries that extract, manipulate, and present data as needed.

31. Define the terms: table, row, and column in SQL.

Table: In SQL, a table is a collection of data organized in rows and columns. Each table represents an entity, and the columns represent the attributes of that entity. For example, a table named Employees might store employee data with columns like EmployeeID, Name, and Department.

Row: A row in a SQL table represents a single record or entry in the table. Each row contains data corresponding to the columns defined in the table. For example, a row in the Employees table might represent a single employee, with data for each column like EmployeeID, Name, and Department.

Column: A column in a SQL table represents an attribute or field of the entity that the table represents. Each column has a name and a data type, and it holds the same type of data for all rows in the table. For example, the Name column in the Employees table might store the names of all employees.

32. How do you comment out lines in SQL?

SQL allows you to comment out lines of code, which means that those lines are ignored when the query is executed. There are two ways to add comments in SQL:

- **Single-line Comments:** Use -- to comment out a single line. Anything after -- on that line will be ignored by the SQL interpreter.

```
-- This is a single-line comment
```

- **Multi-line Comments:** Use /* to start and */ to end a comment that spans multiple lines.

```
/*
    This is a multi-line comment.
    It can span multiple lines.
*/
```

33. What is the purpose of the SELECT statement?

The **SELECT** statement is used to retrieve data from one or more tables in a database. It allows you to specify the columns you want to retrieve and apply filters, sorting, grouping, and other operations to customize the result set. The SELECT statement is the most commonly used SQL command and forms the basis of most queries.

Example Query:

```
SELECT EmployeeID, Name, Department
FROM Employees
WHERE Department = 'IT';
```

This query retrieves the EmployeeID, Name, and Department of all employees who work in the IT department.

34. How do you retrieve all columns from a table using SQL?

To retrieve all columns from a table, you can use the **SELECT *** syntax. The asterisk (*) is a wildcard that represents all columns in the table.

Example Query:

```
SELECT * FROM Employees;
```

This query retrieves all columns for all rows in the Employees table.

35. What is a WHERE clause used for in SQL?

The **WHERE** clause is used to filter rows in a SQL query based on a specified condition. It allows you to retrieve only those rows that meet the criteria defined in the WHERE clause. This clause is commonly used in SELECT, UPDATE, and DELETE statements.

Example Query:

```
SELECT EmployeeID, Name, Department
```

FROM Employees

WHERE Salary > 50000;

This query retrieves the EmployeeID, Name, and Department of employees whose salary is greater than $50,000.

36. Explain the difference between the WHERE and HAVING clauses.

The **WHERE** and **HAVING** clauses are used to filter records in SQL, but they are applied at different stages of query execution.

- **WHERE Clause:** The WHERE clause is used to filter records before any grouping or aggregation occurs. It is applied to individual rows in the table and is used in SELECT, UPDATE, and DELETE statements.
- SELECT EmployeeID, Name
- FROM Employees

WHERE Department = 'IT';

- **HAVING Clause:** The HAVING clause is used to filter records after grouping and aggregation have taken place. It is applied to groups of rows created by the GROUP BY clause and is used only in SELECT statements.
- SELECT Department, COUNT(EmployeeID) AS NumberOfEmployees
- FROM Employees
- GROUP BY Department

HAVING COUNT(EmployeeID) > 10;

37. How do you eliminate duplicate records in a SQL query?

To eliminate duplicate records in a SQL query, you can use the **DISTINCT** keyword. This keyword ensures that only unique rows are returned in the result set.

Example Query:

SELECT DISTINCT Department

FROM Employees;

This query retrieves a list of unique departments from the Employees table.

38. What is the difference between COUNT(*) and COUNT(column_name) in SQL?

The **COUNT(*)** and **COUNT(column_name)** functions are used to count rows in a table, but they differ in what they count:

- **COUNT(*):** Counts the total number of rows in a table, including rows with NULL values in any column. It does not ignore any rows.
- SELECT COUNT(*) AS TotalRows

FROM Employees;

- **COUNT(column_name):** Counts the number of non-NULL values in a specific column. It ignores rows where the specified column has a NULL value.
- SELECT COUNT(Salary) AS NonNullSalaries

FROM Employees;

39. Explain the difference between GROUP BY and ORDER BY clauses.

The **GROUP BY** and **ORDER BY** clauses are used to organize data in SQL queries, but they serve different purposes:

- **GROUP BY Clause:** Groups rows that have the same values in specified columns into aggregated data. It is often used with aggregate functions like COUNT, SUM, AVG, etc., to perform calculations on each group.
- SELECT Department, COUNT(EmployeeID) AS NumberOfEmployees

- FROM Employees

GROUP BY Department;

- **ORDER BY Clause:** Sorts the result set by one or more columns in ascending or descending order. It does not affect the grouping of rows, only the order in which they are displayed.
- SELECT EmployeeID, Name
- FROM Employees

ORDER BY Salary DESC;

40. How do you limit the number of records returned by a SQL query?

To limit the number of records returned by a SQL query, you can use the **LIMIT** clause in databases like MySQL or PostgreSQL, or the **TOP** clause in SQL Server. For Oracle, you can use the **ROWNUM** pseudo-column or the **FETCH FIRST** clause.

- **MySQL/PostgreSQL:** Use LIMIT.
- SELECT * FROM Employees

LIMIT 10;

- **SQL Server:** Use TOP.

SELECT TOP 10 * FROM Employees;

- **Oracle:** Use ROWNUM or FETCH FIRST.
- -- Using ROWNUM
- SELECT * FROM Employees
- WHERE ROWNUM <= 10;
-
- -- Using FETCH FIRST
- SELECT * FROM Employees

FETCH FIRST 10 ROWS ONLY

41. What are Common Table Expressions (CTEs) in SQL, and how are they used?

Common Table Expressions (CTEs) are temporary result sets that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. They are defined using the WITH keyword and are useful for breaking down complex queries into simpler parts.

CTEs can be used for:

- **Improving Readability:** They help to make queries more readable and easier to understand by allowing the use of named result sets.
- **Recursion:** CTEs can be recursive, which allows for querying hierarchical or tree-structured data.

Example Query:

WITH DepartmentCounts AS (

SELECT Department, COUNT(EmployeeID) AS EmployeeCount

FROM Employees

GROUP BY Department

)

SELECT Department

FROM DepartmentCounts

WHERE EmployeeCount > 10;

This query first creates a CTE named DepartmentCounts that calculates the number of employees in each department and then selects departments with more than 10 employees.

42. How do we select the 3rd highest salary from the Employees table?

To select the 3rd highest salary from a table, you can use a subquery with ORDER BY and LIMIT or ROW_NUMBER() function, depending on the SQL database you are using.

Example Queries:

Using LIMIT (MySQL/PostgreSQL):

```
SELECT DISTINCT Salary
FROM Employees
ORDER BY Salary DESC
LIMIT 1 OFFSET 2;
```

- **Using ROW_NUMBER() (SQL Server/Oracle):**

```
WITH RankedSalaries AS (
  SELECT Salary, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS Rank
  FROM Employees
)
SELECT Salary
FROM RankedSalaries
WHERE Rank = 3;
```

These queries retrieve the 3rd highest distinct salary from the Employees table.

43. Explain the difference between a view and a table.

A **view** and a **table** are both database objects, but they have key differences:

- **Table:** A table is a physical database object that stores data in rows and columns. It represents an actual storage structure in the database and holds persistent data.

```
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  Name VARCHAR(100),
  Salary DECIMAL(10, 2)
);
```

- **View:** A view is a virtual table based on the result of a query. It does not store data physically but provides a way to present data from one or more tables. Views are used to simplify complex queries or restrict access to certain data.

```
CREATE VIEW HighSalaryEmployees AS
SELECT EmployeeID, Name, Salary
FROM Employees
WHERE Salary > 50000;
```

44. What is the purpose of the GROUP BY clause in SQL? Provide an example.

The **GROUP BY** clause is used to group rows that have the same values in specified columns into aggregated data. It is commonly used with aggregate functions like COUNT, SUM, AVG, etc., to perform calculations on each group.

Example Query:

```
SELECT Department, COUNT(EmployeeID) AS NumberOfEmployees
FROM Employees
GROUP BY Department;
```

This query groups the employees by department and counts the number of employees in each department.

45. Explain the difference between INNER JOIN and LEFT JOIN with examples.

The **INNER JOIN** and **LEFT JOIN** clauses are used to combine rows from two or more tables based on a related column, but they handle unmatched rows differently.

- **INNER JOIN:** Returns only the rows with matching values in both tables. If there is no match, the row is excluded from the result set.

  SELECT Orders.OrderID, Customers.CustomerName

  FROM Orders

  INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

- **LEFT JOIN (or LEFT OUTER JOIN):** Returns all rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for columns from the right table.

  SELECT Orders.OrderID, Customers.CustomerName

  FROM Orders

  LEFT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

46. Discuss the role of the WHERE clause in SQL queries and provide examples of its usage.

The **WHERE** clause is used in SQL to filter records based on specific conditions. It determines which rows will be included in the result set of a SELECT query, or affected by UPDATE and DELETE statements.

Example Queries:

- **Filter Rows in SELECT Query:**
- SELECT * FROM Employees
- WHERE Salary > 50000;

This query retrieves all employees with a salary greater than $50,000.

- **Update Rows Based on Condition:**

UPDATE Employees

SET Salary = Salary * 1.1

WHERE PerformanceRating = 'Excellent';

This query increases the salary by 10% for employees with an 'Excellent' performance rating.

- **Delete Rows Based on Condition:**

DELETE FROM Employees

WHERE RetirementDate < CURDATE();

This query deletes employees whose retirement date is earlier than the current date.

47. Explain the concept of database transactions and the ACID properties.

A **database transaction** is a sequence of one or more SQL operations executed as a single unit of work. Transactions ensure data integrity by either completing all operations successfully or rolling back changes in case of an error.

The **ACID properties** are critical for maintaining the reliability of transactions:

- **Atomicity:** Ensures that all operations within a transaction are completed successfully; otherwise, the transaction is rolled back.
- **Consistency:** Ensures that a transaction transforms the database from one valid state to another, preserving database integrity.

- **Isolation:** Ensures that transactions are executed in isolation from one another, so concurrent transactions do not interfere with each other.
- **Durability:** Ensures that once a transaction is committed, its changes are permanent and survive system failures.

48. Describe the benefits of using subqueries in SQL and provide a scenario where they would be useful.

Subqueries are queries nested within another query and can be used to perform complex filtering and calculations. They provide several benefits:

- **Enhanced Query Flexibility:** Subqueries allow for more flexible and dynamic queries by enabling nested queries that use the result of one query as input for another.
- **Complex Filtering:** They can be used to filter results based on aggregated or computed values.
- **Data Isolation:** Subqueries help isolate and simplify parts of complex queries, improving readability and maintainability.

Example Scenario:

```
SELECT EmployeeID, Name
FROM Employees
WHERE Salary > (
   SELECT AVG(Salary)
   FROM Employees
);
```

This query retrieves employees whose salary is above the average salary of all employees.

49. Discuss the differences between the CHAR and VARCHAR data types in SQL.

Both **CHAR** and **VARCHAR** are used to store character data, but they have key differences:

- **CHAR:**
    - Fixed-length string.
    - Pad shorter values with spaces to match the defined length.
    - More efficient for storing fixed-length data but can waste storage space for variable-length data.

```
CREATE TABLE Employees (
   EmployeeID INT,
   Name CHAR(10)
);
```

- **VARCHAR:**
    - Variable-length string.
    - Only uses space for the actual length of the string, plus a small amount of overhead.
    - More efficient for storing variable-length data but can have slightly higher overhead.

```
CREATE TABLE Employees (
   EmployeeID INT,
   Name VARCHAR(50)
);
```

50. Explain the purpose of the ORDER BY clause in SQL queries and provide examples.

The **ORDER BY** clause is used to sort the result set of a SQL query by one or more columns. It can sort data in ascending (ASC) or descending (DESC) order.

Example Queries:

- **Sort by Single Column (Ascending):**

SELECT * FROM Employees

ORDER BY Salary ASC;

This query retrieves all employees sorted by their salary in ascending order.

- **Sort by Single Column (Descending):**

SELECT * FROM Employees

ORDER BY Salary DESC;

This query retrieves all employees sorted by their salary in descending order.

- **Sort by Multiple Columns:**

SELECT * FROM Employees

ORDER BY Department ASC, Salary DESC;

This query retrieves employees sorted by department in ascending order, and within each department, sorted by salary in descending order.

51. Describe the importance of data integrity constraints such as NOT NULL, UNIQUE, and CHECK constraints in SQL databases.

Data integrity constraints are essential for maintaining the accuracy and consistency of data within a database. They enforce rules that ensure the data adheres to specific standards and business requirements.

- **NOT NULL:** Ensures that a column cannot have NULL values, which means that every row must have a value for this column. This constraint is crucial for mandatory fields where missing data would lead to incomplete records.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100) NOT NULL

);

- **UNIQUE:** Ensures that all values in a column (or a set of columns) are unique across the table. This constraint is important for columns that require distinct values, such as email addresses or usernames.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Email VARCHAR(100) UNIQUE

);

- **CHECK:** Ensures that values in a column meet a specific condition. This constraint helps enforce business rules, such as valid ranges for numerical values or specific formats for text.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Salary DECIMAL(10, 2),

CHECK (Salary > 0)

);

52. Discuss the advantages and disadvantages of using stored procedures.

Stored procedures are precompiled SQL statements that are stored in the database and can be executed with a single call. They offer several advantages and some disadvantages:

- **Advantages:**
    - **Performance:** Stored procedures are precompiled, which can improve execution performance compared to ad-hoc SQL queries.
    - **Reusability:** They can be reused across different applications and parts of an application, reducing redundancy.
    - **Security:** Stored procedures can help protect against SQL injection attacks by parameterizing queries and restricting direct access to database tables.
    - **Maintainability:** Encapsulating complex logic within stored procedures can simplify code maintenance and updates.
- **Disadvantages:**
    - **Complexity:** Managing and debugging stored procedures can become complex, especially in large systems with many procedures.
    - **Portability:** Stored procedures may be specific to a particular database management system, which can affect portability across different database systems.
    - **Overhead:** Excessive use of stored procedures can lead to performance overhead and reduce the flexibility of applications if not managed properly.

Commonly SQL Interview Questions

1. Write a query to delete duplicate rows from a table.

To delete duplicate rows, you first need to identify the duplicates and then remove them. Here's a common approach using a Common Table Expression (CTE) with the ROW_NUMBER() function:

```
WITH CTE AS (
   SELECT *,
       ROW_NUMBER() OVER (PARTITION BY column1, column2, ... ORDER BY (SELECT 0)) AS rn
   FROM TableName
)
DELETE FROM CTE
WHERE rn > 1;
```

Replace column1, column2, ... with the columns that define the uniqueness of a row, and TableName with your table name.

2. Write a query to retrieve the names of employees who work in the same department as 'John'.

```
SELECT e2.Name
FROM Employees e1
JOIN Employees e2 ON e1.DepartmentID = e2.DepartmentID
WHERE e1.Name = 'John';
```

This query joins the Employees table with itself to find employees in the same department as the employee named 'John'.

3. Write a query to display the second highest salary from the Employee table.

You can use a subquery to find the second highest salary. Here's one way to achieve this:

```
SELECT MAX(Salary) AS SecondHighestSalary
```

```
FROM Employees
WHERE Salary < (
   SELECT MAX(Salary)
   FROM Employees
);
```

This query finds the maximum salary that is less than the highest salary, effectively giving you the second highest salary.

4. Write a query to find all customers who have made more than two orders.

```
SELECT CustomerID
FROM Orders
GROUP BY CustomerID
HAVING COUNT(OrderID) > 2;
```

This query groups the orders by CustomerID and uses the HAVING clause to filter customers who have made more than two orders.

5. Write a query to count the number of orders placed by each customer.

```
SELECT CustomerID, COUNT(OrderID) AS NumberOfOrders
FROM Orders
GROUP BY CustomerID;
```

This query counts the number of orders for each CustomerID by grouping the results and using the COUNT function.

6. Write a query to retrieve the list of employees who joined in the last 3 months.

```
SELECT *
FROM Employees
WHERE JoinDate > DATEADD(MONTH, -3, GETDATE());
```

This query retrieves employees whose JoinDate is within the last 3 months. Replace JoinDate with the actual column name and adjust the function if using a different SQL dialect (e.g., use SYSDATE for Oracle).

7. Write a query to find duplicate records in a table and count the number of duplicates for each unique record.

```
WITH DuplicateRecords AS (
   SELECT column1, column2, ..., COUNT(*)
   FROM TableName
   GROUP BY column1, column2, ...
   HAVING COUNT(*) > 1
)
SELECT *, COUNT(*) AS DuplicateCount
FROM TableName
JOIN DuplicateRecords
ON TableName.column1 = DuplicateRecords.column1
AND TableName.column2 = DuplicateRecords.column2
GROUP BY column1, column2, ...;
```

Replace column1, column2, ... with the columns that determine duplicates and TableName with your table name.

8. Write a query to list all products that have never been sold.

```
SELECT p.ProductID, p.ProductName

FROM Products p

LEFT JOIN Orders o ON p.ProductID = o.ProductID

WHERE o.OrderID IS NULL;
```

This query uses a LEFT JOIN between Products and Orders to find products with no corresponding orders.

9. Write a query to update the salary of employees based on their performance rating.

```
UPDATE Employees

SET Salary = CASE

   WHEN PerformanceRating = 'Excellent' THEN Salary * 1.10

   WHEN PerformanceRating = 'Good' THEN Salary * 1.05

   ELSE Salary

END;
```

This query updates the salary of employees based on
their PerformanceRating. Adjust the percentage increases as needed.

10. Write a query to find all employees who earn more than the average salary.

```
SELECT *

FROM Employees

WHERE Salary > (

   SELECT AVG(Salary)

   FROM Employees

);
```

This query retrieves employees whose salary is greater than the average salary calculated for all employees.

11. Write a query to retrieve the list of employees who joined in the last 3 months.

```
SELECT *

FROM Employees

WHERE JoinDate > DATEADD(MONTH, -3, GETDATE());
```

This query selects all employees who joined within the last 3 months.
The DATEADD function is used to calculate the date 3 months ago. Adjust the function according to your SQL dialect if needed.

12. Write a query to identify the top 10 customers who have not placed an order in the last year.

```
WITH RecentOrders AS (

   SELECT DISTINCT CustomerID

   FROM Orders

   WHERE OrderDate > DATEADD(YEAR, -1, GETDATE())

)

SELECT c.CustomerID, c.CustomerName

FROM Customers c

LEFT JOIN RecentOrders ro ON c.CustomerID = ro.CustomerID

WHERE ro.CustomerID IS NULL

ORDER BY c.CustomerID
```

```
LIMIT 10;
```

This query first identifies customers who have placed orders in the last year using a CTE. It then selects the top 10 customers who have not placed any orders in that period.

13. Create a query to compute the year-over-year growth rate of revenue for each product category.

```
WITH RevenueByYear AS (
   SELECT
      ProductCategory,
      YEAR(OrderDate) AS Year,
      SUM(Revenue) AS TotalRevenue
   FROM Orders
   GROUP BY ProductCategory, YEAR(OrderDate)
)
SELECT
   r1.ProductCategory,
   r1.Year AS CurrentYear,
   r1.TotalRevenue AS CurrentYearRevenue,
   r2.Year AS PreviousYear,
   r2.TotalRevenue AS PreviousYearRevenue,
   ((r1.TotalRevenue - r2.TotalRevenue) / r2.TotalRevenue * 100) AS
GrowthRate
FROM RevenueByYear r1
LEFT JOIN RevenueByYear r2
ON r1.ProductCategory = r2.ProductCategory
AND r1.Year = r2.Year + 1;
```

This query calculates the year-over-year revenue growth for each product category. It uses a CTE to summarize revenue by year and category, then joins the table with itself to compare revenue between years.

14. Write a query to join three tables and filter the results to show only records that exist in exactly two of the tables.

```
SELECT t1.ID
FROM Table1 t1
JOIN Table2 t2 ON t1.ID = t2.ID
LEFT JOIN Table3 t3 ON t1.ID = t3.ID
GROUP BY t1.ID
HAVING COUNT(DISTINCT CASE
   WHEN t2.ID IS NOT NULL THEN 'Table2'
   WHEN t3.ID IS NOT NULL THEN 'Table3'
   ELSE 'Other'
END) = 2;
```

This query joins three tables and uses a GROUP BY clause with a HAVING clause to ensure that only IDs appearing in exactly two tables are returned.

15. Write a query to calculate the retention rate of customers over a given time period.

```
WITH CustomerActivity AS (
```

```
    SELECT CustomerID, MIN(OrderDate) AS FirstOrderDate
    FROM Orders
    GROUP BY CustomerID
),
RetainedCustomers AS (
    SELECT DISTINCT CustomerID
    FROM Orders
    WHERE OrderDate > DATEADD(MONTH, -6, GETDATE())  -- Adjust the time period as needed
)
SELECT
    COUNT(DISTINCT ca.CustomerID) AS TotalCustomers,
    COUNT(DISTINCT rc.CustomerID) AS RetainedCustomers,
    (COUNT(DISTINCT rc.CustomerID) * 100.0 / COUNT(DISTINCT ca.CustomerID)) AS RetentionRate
FROM CustomerActivity ca
LEFT JOIN RetainedCustomers rc ON ca.CustomerID = rc.CustomerID;
```

This query calculates the retention rate by comparing the total number of customers with those who have placed orders within a specified time period. Adjust the DATEADD function to reflect the desired retention period.

What is a CROSS Join? (Asked in JP Morgan Interview)

Also known as a cartesian join. Used to generate combinations from each row of first table with each row of second table.

Example, if 1st table have 3 rows and 2nd table have 4 rows, this join will return 3*4 that is 12 rows in output.

Query:

```
SELECT * FROM Table1
CROSS JOIN Table2
```

Example use case: Table1 have list of colours and Table2 have list of patterns. So, this join will return all possible colour-combinations.