Top 24 SQL Interview Questions & Answers

1. **How to use aggregate functions as window functions?**

Answer: To use aggregate functions as window functions in SQL, you can use the OVER() clause. Window functions allow you to perform calculations across a set of table rows that are somehow related to the current row. Unlike aggregate functions, window functions do not cause rows to become grouped into a single output row—they still return multiple rows.

Syntax for Window Functions

```
SELECT column_name,
    AGGREGATE_FUNCTION(column_name) OVER (PARTITION BY column_name ORDER BY column_name) AS alias_name
FROM table_name;
```

Common Aggregate Functions Used as Window Functions

1. SUM(): Calculates the sum of values.
2. AVG(): Calculates the average of values.
3. COUNT(): Counts the number of rows or non-null values.
4. MIN(): Finds the minimum value.
5. MAX(): Finds the maximum value.

2. What is the difference between COUNT(*) and COUNT(column_name) when used on a table?

Answer:

1. **COUNT(*)**
    - Purpose: COUNT(*) counts all the rows in a table, including rows with NULL values in any column.
    - Usage: Use COUNT(*) when you want to count every row in the result set, regardless of whether it contains NULL values.

- Example: If a table has 5 rows, COUNT(*) will return 5, even if some columns have NULL values.

2. **COUNT(column_name)**
   - Purpose: COUNT(column_name) counts only the rows where the specified column has a non-NULL value. Rows with NULL in the specified column are ignored.
   - Usage: Use COUNT(column_name) when you want to count only the rows where the specified column has data (i.e., non-NULL values).
   - Example: If a table has 5 rows and the specified column has 2 NULL values, COUNT(column_name) will return 3 (only counting non-NULL values).

3. What is the difference between LEFT, RIGHT, FULL outer join and INNER join?

Answer:

1. **INNER JOIN**

What it does: An inner join returns only the rows that have matching values in both tables.

When to use it: Use an inner join when you only want the data that exists in both tables.

Example:

Suppose we have two tables:

Employees table:

EmployeeID	Name
1	Alice
2	Bob
3	Carol

Departments table:

EmployeeID	Department

1	HR
2	IT
4	Marketing

Query using inner join:

```
SELECT Employees.Name, Departments.Department
FROM Employees
INNER JOIN Departments ON Employees.EmployeeID =
Departments.EmployeeID;
```

Result:

Name	Department
Alice	HR
Bob	IT

Only Alice and Bob are returned because they have matching EmployeeID values in both tables.

2. **LEFT JOIN (or LEFT OUTER JOIN)**

What it does: A left join returns all rows from the left table (Employees), and the matched rows from the right table (Departments). If there is no match, the result is `NULL` on the right side.

When to use it: Use a left join when you want to see all the records from the left table, even if there are no matching records in the right table.

Query using left join:

```
SELECT Employees.Name, Departments.Department
FROM Employees
LEFT JOIN Departments ON Employees.EmployeeID =
Departments.EmployeeID;
```

Result:

Name	Department
Alice	HR
Bob	IT
Carol	NULL

Carol is returned even though there's no matching department, and her department is shown as `NULL`.

3. **RIGHT JOIN (or RIGHT OUTER JOIN)**

What it does: A right join returns all rows from the right table (Departments), and the matched rows from the left table (Employees). If there is no match, the result is `NULL` on the left side.

When to use it: Use a right join when you want to see all the records from the right table, even if there are no matching records in the left table.

Query using right join:

```
SELECT Employees.Name, Departments.Department
FROM Employees
RIGHT JOIN Departments ON Employees.EmployeeID =
Departments.EmployeeID;
```

Result:

Name	Department
Alice	HR
Bob	IT
NULL	Marketing

The department "Marketing" is returned even though there's no matching employee, and the name is shown as `NULL`.

4. **FULL OUTER JOIN**

What it does: A full outer join returns all rows when there is a match in either left table or right table records. Rows that do not have a match in one of the tables are returned with `NULL` for the columns of the other table.

When to use it: Use a full outer join when you want to see all records from both tables, regardless of whether they have a match.

Query using full outer join:

```
SELECT Employees.Name, Departments.Department
FROM Employees
```

FULL OUTER JOIN Departments ON Employees.EmployeeID = Departments.EmployeeID;

Result:

Name	Department
Alice	HR
Bob	IT
Carol	NULL
NULL	Marketing

All records from both tables are returned. Carol doesn't have a matching department, and "Marketing" doesn't have a matching employee, so their missing parts are filled with `NULL`.

4. What is the difference between DISTINCT and GROUP BY?

Answer:
DISTINCT

- Purpose: DISTINCT is used to remove duplicate rows and return only unique values from a column or combination of columns in the result set.
- When to use: Use DISTINCT when you want to eliminate duplicate values and see each value only once in the result.

GROUP BY

- Purpose: GROUP BY is used to group rows that have the same values in specified columns. It is often used with aggregate functions (like COUNT(), SUM(), AVG()) to perform calculations on each group.
- When to use: Use GROUP BY when you want to organize data into groups and perform summary operations on these groups.

5. What is the difference between DELETE and TRUNCATE statement?

Answer:

DELETE Statement

- Purpose: DELETE is used to remove specific rows from a table based on a condition. You can delete all rows or just a subset of rows using a WHERE clause.
- Usage: You can specify which rows to delete.

TRUNCATE Statement

- Purpose: TRUNCATE is used to quickly remove all rows from a table, effectively emptying it.
- Usage: You cannot specify individual rows to delete; it removes all rows.

6. What is the difference between Where and Having?

WHERE Clause

- Purpose: WHERE is used to filter rows before any grouping or aggregation happens in the query.
- Usage: It is used with individual rows and is applied directly to the columns of the table.
- Functionality: You cannot use aggregate functions (like SUM(), COUNT()) directly in the WHERE clause.
- Example Use Case: Filtering rows based on a condition (e.g., WHERE age > 18).

HAVING Clause

- Purpose: HAVING is used to filter groups of rows after the grouping operation (like GROUP BY) has been performed.
- Usage: It is used with aggregated data to filter the result set after the aggregation.
- Functionality: You can use aggregate functions (like SUM(), COUNT()) in the HAVING clause.

7. **What is the order of execution of a SQL query?**

Answer:

Here’s the simplified order of execution for a SQL query:

1. **FROM/JOIN**: Identifies the table(s) to retrieve data from. Joins between tables are also processed here.
2. **WHERE**: Filters rows based on specified conditions. Only rows that match these conditions proceed.
3. **GROUP BY**: Groups rows that have the same values in specified columns. This is useful for aggregation.
4. **HAVING**: Filters groups based on conditions, similar to `WHERE` but works on grouped data.
5. **SELECT**: Selects the columns or expressions to return in the result. Performs any calculations needed.
6. **ORDER BY**: Sorts the result set based on one or more columns or expressions.
7. **LIMIT**: Limits the number of rows returned by the query.

8. **What is the difference between a Primary Key, Unique Key, and Foreign Key?**

Answer:

1. **Primary Key**
   - **Purpose**: A primary key uniquely identifies each row in a table. It ensures that no duplicate values exist in the specified column(s) and that each row can be uniquely identified.
   - **Characteristics**:
     - Must be unique for each row.
     - Cannot contain NULL values.
     - A table can have only one primary key.
2. **Unique Key**
   - **Purpose**: A unique key also ensures that all values in a column (or combination of columns) are unique, similar to a primary key.
   - **Characteristics**:
     - Must be unique, but unlike a primary key, it can have one NULL value (depending on the database).
     - A table can have multiple unique keys.
3. **Foreign Key**
   - **Purpose**: A foreign key establishes a relationship between two tables by referencing the primary key or unique key of another table. It is used to enforce referential integrity.

- **Characteristics**:
  - It can have duplicate values.
  - It can contain NULL values.
  - Ensures that the value in the foreign key column matches a value in the referenced primary key or unique key.

9. **What are DDL and DML languages? Give examples.**

Answer:

1. **DDL (Data Definition Language)**
   - **Purpose**: DDL is used to define and modify the structure of database objects such as tables, indexes, and schemas. It deals with the schema of the database.
   - **Examples of DDL Commands**:
     - CREATE: Used to create new database objects like tables and indexes.
     - ALTER: Used to modify existing database objects (e.g., adding a column to a table).
     - DROP: Used to delete existing database objects (e.g., dropping a table or index).
     - TRUNCATE: Used to remove all rows from a table but keep its structure intact.
2. **DML (Data Manipulation Language)**
   - **Purpose**: DML is used to manipulate data within the database. It deals with the data itself, allowing you to perform operations such as inserting, updating, deleting, and retrieving data.
   - **Examples of DML Commands**:
     - INSERT: Used to add new records (rows) into a table.
     - UPDATE: Used to modify existing records in a table.
     - DELETE: Used to remove records from a table.
     - SELECT: Used to retrieve data from one or more tables.

10. **Why do we use the CASE statement in SQL? Give an example.**

Answer:

Purpose of the CASE Statement:

- The CASE statement is used in SQL to implement conditional logic. It allows you to perform different actions or return different results based on specific conditions.
- It works like an IF-THEN-ELSE statement in programming. It evaluates conditions and returns a value when the first condition is met. If no conditions are met, it returns a default value (optional).

When to Use:

- To categorize data into different groups based on conditions.
- To replace multiple IF-THEN statements in queries.
- To create calculated fields or transform data without altering the original data.

11. **What are the prerequisites to use the UNION operator in SQL?**

Answer:

To use the UNION operator in SQL, which combines the result sets of two or more SELECT statements into a single result set, certain prerequisites must be met:

1. **Same Number of Columns**:
    - Each SELECT statement involved in the UNION must return the same number of columns. The columns should match in count across all queries.
2. **Same Data Types**:
    - The corresponding columns in each SELECT statement must have compatible data types. For example, if the first column in the first query is an integer, the first column in the other queries should also be of an integer type or a compatible type.
3. **Order of Columns**:
    - The order of columns in each SELECT statement must match. The first column in each query should correspond to the same kind of data, and so on.
4. **Column Names**:
    - The column names in the final result set are taken from the first SELECT statement. Subsequent SELECT statements do not need

to have the same column names, but the types and order must still match.

5. **Usage of UNION vs UNION ALL**:
    - UNION removes duplicate records from the combined result set, while UNION ALL includes all records, including duplicates. Make sure to choose the appropriate one based on whether you want duplicates removed.

12. **What is the difference between RANK(), DENSE_RANK(), and ROW_NUMBER() window functions in SQL?**

Answer:

RANK(), DENSE_RANK(), and ROW_NUMBER() are all window functions in SQL that assign a unique rank or number to rows within a partition of the result set. They are often used to rank or order data within groups. Here's how they differ:

1. **RANK()**
    - **Purpose**: Assigns a rank to each row within a partition of the result set, with gaps in ranking if there are ties.
    - **Behavior**: If two rows have the same rank, the next rank is skipped. For example, if two rows are ranked 1, the next row will be ranked 3 (not 2).
    - **Example**: If you rank students based on their scores and two students have the same score, both will get rank 1, and the next highest score will get rank 3.
2. **DENSE_RANK()**
    - **Purpose**: Similar to RANK(), but without gaps. Assigns consecutive ranks to rows within a partition of the result set.
    - **Behavior**: If two rows have the same rank, the next rank is not skipped. For example, if two rows are ranked 1, the next row will be ranked 2.
    - **Example**: In the same scenario of ranking students, if two students have the same score, both get rank 1, and the next highest score will get rank 2 (no rank is skipped).
3. **ROW_NUMBER()**
    - **Purpose**: Assigns a unique sequential integer to each row within a partition of the result set, without considering ties.

    - **Behavior**: Each row gets a unique number, even if there are ties. It does not consider the value of the data for ranking.
    - **Example**: If you use ROW_NUMBER() to number students, each student will get a different number regardless of whether they have the same score or not (1, 2, 3, etc.).

13. What are subqueries, and where can we use them?

Answer:

Subqueries:

- **Definition**: A subquery is a query nested inside another query (the main query). It is also known as an inner query or a nested query. Subqueries are used to perform operations that would otherwise require multiple steps or complex logic.
- **Purpose**: They are often used to filter data, perform calculations, or return data that is then used by the main query.

14. **Which is better to use, CTE or subquery?**

Answer:

1. **CTE (Common Table Expression)**
    - **Definition**: Temporary result set defined using the WITH clause.
    - **Pros**: Improves readability, easier to maintain, can be reused multiple times, supports recursion.
    - **Use When**: Query is complex, needs to be broken into steps, or results need to be reused.
    - **Example**: Easier to read and understand in complex queries.
2. **Subquery**
    - **Definition**: Query nested inside another query.
    - **Pros**: Simple and concise, good for single-use calculations or filters.
    - **Use When**: Query is straightforward, result is used only once.
    - **Example**: Suitable for one-time filtering and calculations.

15. **What is the difference between a function and a procedure?**

Answer:

1. **Function**
    - **Purpose**: A function is designed to perform a calculation or operation and return a single value.
    - **Return Value**: Always returns a value (either scalar or table type).
    - **Usage**: Typically used within SQL statements, such as in the SELECT clause or as part of a WHERE clause.
    - **Syntax**: Invoked in SQL statements. Example: SELECT function_name(parameters);
    - **Side Effects**: Cannot perform actions that modify the database state (like INSERT, UPDATE, DELETE).
    - **Example Use Case**: Calculate a sum, average, or format a string.
2. **Procedure (Stored Procedure)**
    - **Purpose**: A procedure is designed to perform a sequence of operations or tasks, which can include modifying the database state.
    - **Return Value**: May or may not return a value. Can return multiple results and output parameters.
    - **Usage**: Executed using a call statement. It can perform actions like inserting, updating, or deleting records in the database.
    - **Syntax**: Invoked using EXEC or CALL. Example: EXEC procedure_name(parameters);
    - **Side Effects**: Can change the database state, handle transactions, and perform complex operations.
    - **Example Use Case**: Update records in a table, log information, handle complex business logic.

16. **Can we use an alias in the WHERE clause?**

Answer:

No, you cannot use an alias directly in the WHERE clause. Aliases are defined in the SELECT clause and are not recognized in the WHERE clause because the WHERE clause is processed before the SELECT clause in the order of execution.

Workaround: If you need to use an alias in a condition, you can use a Common Table Expression (CTE) or a subquery.

17. Find the 3rd Highest Salary

Sample Table: `Employees`

| EmployeeID | Name | Salary |

|---|---|---|

| 1 | Alice | 5000 |

| 2 | Bob | 7000 |

| 3 | Carol | 6000 |

| 4 | Dave | 8000 |

| 5 | Eve | 5500 |

Query Solution:

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

18. Find Employees Whose Salary is Greater Than Their Manager's Salary

Sample Table: `Employees`

| EmployeeID | Name | Salary | ManagerID |

|---|---|---|---|

| 1 | Alice | 5000 | 3 |

| 2 | Bob | 7000 | 1 |

| 3 | Carol | 6000 | NULL |

| 4 | Dave | 8000 | 3 |

| 5 | Eve | 5500 | 2 |

Query Solution:

```
SELECT e1.Name, e1.Salary
FROM Employees e1
JOIN Employees e2 ON e1.ManagerID = e2.EmployeeID
WHERE e1.Salary > e2.Salary;
```

19. Calculate Cumulative Sum in a New Column

Sample Table: `Sales`

| SaleID | Amount |

|---|---|

| 1 | 100 |

| 2 | 200 |

| 3 | 300 |

| 4 | 150 |

Query Solution:

```
SELECT SaleID, Amount,
    SUM(Amount) OVER (ORDER BY SaleID) AS CumulativeSum
```

```
FROM Sales;
```

20. Write a Query to Delete Duplicate Rows

Sample Table: `Employees`

EmployeeID	Name	Salary
1	Alice	5000
2	Bob	7000
3	Bob	7000
4	Dave	8000
5	Eve	5500

Query Solution:

```
WITH CTE AS (
    SELECT Name, Salary,
        ROW_NUMBER() OVER (PARTITION BY Name, Salary ORDER BY EmployeeID) AS RowNum
    FROM Employees
)
DELETE FROM CTE
WHERE RowNum > 1;
```

21. Query to Calculate Mean, Median, and Mode

Sample Table: `Employees`

| EmployeeID | Name | Salary |

|---|---|---|

| 1 | Alice | 5000 |

| 2 | Bob | 7000 |

| 3 | Carol | 6000 |

| 4 | Dave | 8000 |

| 5 | Eve | 5000 |

Query Solution (Mean):

```
SELECT AVG(Salary) AS MeanSalary
FROM Employees;
```

Query Solution (Median):

```
SELECT AVG(Salary) AS MedianSalary
FROM (
    SELECT Salary
    FROM Employees
    ORDER BY Salary
    LIMIT 2 - (SELECT COUNT(*) FROM Employees) % 2    -- Handles even/odd row count
    OFFSET (SELECT (COUNT(*) - 1) / 2 FROM Employees)
) AS MedianSubquery;
```

Query Solution (Mode):

```
SELECT Salary AS ModeSalary

FROM Employees

GROUP BY Salary

ORDER BY COUNT(*) DESC

LIMIT 1;
```

22. Query to Fetch Employees Earning More Than Their Department's Average Salary

Sample Table: `Employees`

EmployeeID	Name	Salary	DepartmentID
1	Alice	5000	1
2	Bob	7000	1
3	Carol	6000	2
4	Dave	8000	2
5	Eve	5500	1

Query Solution:

```
SELECT e.Name, e.Salary, e.DepartmentID

FROM Employees e

JOIN (

    SELECT DepartmentID, AVG(Salary) AS AvgSalary

    FROM Employees
```

```
    GROUP BY DepartmentID
) AS dept_avg ON e.DepartmentID = dept_avg.DepartmentID
WHERE e.Salary > dept_avg.AvgSalary;
```

23. Given the following tables:

Table_1:

id
1
1
1
2
2
NULL
NULL
3
NULL

Table_2:

id
1
1

| 2 |

| 2 |

| NULL |

| NULL |

| 2 |

| 3 |

| 4 |

| 4 |

Find the number of rows in the output of the following joins between Table_1 and Table_2 on the `id` column:

1. INNER JOIN

2. LEFT JOIN

3. RIGHT JOIN

4. FULL OUTER JOIN

Answers:

1**. INNER JOIN**

An INNER JOIN returns only the rows with matching values in both tables.

- Matching values: 1, 2, 3, and NULL.

  - For id = 1: 3 rows from Table_1 × 2 rows from Table_2 = 6 rows

  - For id = 2: 2 rows from Table_1 × 3 rows from Table_2 = 6 rows

  - For id = 3: 1 row from Table_1 × 1 row from Table_2 = 1 row

Total Rows for INNER JOIN: 13 ROWS

2. LEFT JOIN

A LEFT JOIN returns all rows from Table_1, and the matching rows from Table_2. If no match, it returns NULL for Table_2.

- Total not matching rows from Table_1: 3 (NULL ROWS will not match)

- Matches as in INNER JOIN: 13 rows

Total Rows for LEFT JOIN: 16 ROWS

3. RIGHT JOIN

A RIGHT JOIN returns all rows from Table_2, and the matching rows from Table_1. If no match, it returns NULL for Table_1.

- Total not matching rows from Table_2: 4 (2 NULL rows & 2 Rows containing 4 will not match)

- Matches as in INNER JOIN: 13 rows

Total Rows for RIGHT JOIN: 17 rows

4. **FULL OUTER JOIN**

A FULL OUTER JOIN returns rows when there is a match in either Table_1 or Table_2. If there is no match, it returns NULL in the unmatched side.

- Matching rows: 13 (from INNER JOIN)

- Non-matching rows from Table_1: 2 Rows (Null Rows)

- Non-matching rows from Table_2: 4 (2 NULL rows & 2 Rows containing 4 will not match)

Total Rows for FULL OUTER JOIN: 19 Rows

24. Calculate Month-over-Month Sales Growth

Sample Table: Sales

SaleID	SaleDate	SaleAmount
1	2024-01-15	500
2	2024-01-20	700
3	2024-02-10	600
4	2024-02-15	800
5	2024-03-05	900
6	2024-03-20	1000

Goal: Find the total sales amount for each month and calculate the month-over-month sales growth.

Query:

```
SELECT
    DATE_FORMAT(SaleDate, '%Y-%m') AS SaleMonth,
    SUM(SaleAmount) AS TotalSales,
    LAG(SUM(SaleAmount)) OVER (ORDER BY DATE_FORMAT(SaleDate, '%Y-%m')) AS PreviousMonthSales,
```

```
    (SUM(SaleAmount) - LAG(SUM(SaleAmount)) OVER (ORDER BY
DATE_FORMAT(SaleDate, '%Y-%m'))) / LAG(SUM(SaleAmount)) OVER
(ORDER BY DATE_FORMAT(SaleDate, '%Y-%m')) * 100 AS
MonthOverMonthGrowth

FROM

    Sales

GROUP BY

    DATE_FORMAT(SaleDate, '%Y-%m')

ORDER BY

    SaleMonth;
```