Sql Query Based Interview Questions And Answers

MASTERING THE ART OF THE SQL INTERVIEW: QUERY-BASED QUESTIONS AND ANSWERS

In today's data-driven world, Structured Query Language (SQL) remains the cornerstone of database management and analysis. Whether you are a junior developer, a data analyst, or a seasoned database administrator, your ability to write efficient and correct SQL queries is often the deciding factor in landing your next role. Interviewers frequently move beyond theoretical knowledge, choosing instead to test candidates with practical, query-based problems. This article delves into the most common and revealing **SQL query-based interview questions and answers**, equipping you with the logic, syntax, and problem-solving techniques needed to impress. We'll explore real-world scenarios, from basic joins to advanced window functions, ensuring you understand not just the answer, but the reasoning behind it.

WHY QUERY-BASED QUESTIONS MATTER IN SQL INTERVIEWS

Corporate databases are rarely simple. They contain multiple tables with complex relationships, millions of rows, and performance considerations. A question like "What is a JOIN?" can be memorized, but "Write a query to find customers who placed orders in the last 30 days but have not made any returns" demands a practical command of SQL. Interviewers use these problems to assess your logical thinking, familiarity with SQL syntax, and ability to handle edge cases. The following sections break down classic query types, providing clear explanations and optimized answer structures.

Basic Retrieval and Filtering

Every SQL interview begins with the fundamentals. You can expect questions involving **SELECT**, **WHERE**, **DISTINCT**, and ordering. While simple, these questions test your attention to detail and understanding of NULL handling.

Question: Write a query to fetch all employees from the "employees" table who joined in the year 2022 and have a salary greater than $50,000. Order the results by hire date descending.

Answer:

```
SELECT *
FROM employees
WHERE YEAR(hire_date) = 2022
  AND salary > 50000
ORDER BY hire_date DESC;
```

Explanation: The `YEAR()` function extracts the year from a date column. An alternative in some databases is `hire_date BETWEEN '2022-01-01' AND '2022-12-31'` which can also

utilize indexes more efficiently. Always consider database-specific functions. Also note the use of `salary > 50000` (rather than >=) as specified.

Joining Tables Effectively

Relational databases rely on joins. Expect questions that require you to combine data from two or more tables. Many interviewers also ask you to differentiate between **INNER JOIN**, **LEFT JOIN**, and **RIGHT JOIN** by providing a scenario.

Question: Given two tables: *students (id, name, department_id)* and *departments (id, department_name)*. Write a query to list all students along with their department name. If a student is not assigned to any department, show "Not Assigned".

Answer:

```
SELECT s.id, s.name,
COALESCE(d.department_name, 'Not
Assigned') AS department_name
FROM students s
LEFT JOIN departments d ON
s.department_id = d.id;
```

Explanation: Using `LEFT JOIN` ensures that all students from the left table appear, even if `department_id` is `NULL`. The `COALESCE` function (or `ISNULL` in SQL Server, `IFNULL` in MySQL) handles the NULL replacement. An alternative is `CASE WHEN` but `COALESCE` is more concise.

Grouping and Aggregation

Questions involving **GROUP BY** and aggregate functions (**COUNT**, **SUM**, **AVG**, **MAX**, **MIN**) test your ability to summarize data. A common twist is using **HAVING** to filter groups.

Question: From the "orders" table (columns: order_id, customer_id, order_amount, order_date), write a query to find customers who have placed more than 5 orders, and display the total amount they have spent. Show only customers with total spending exceeding $1,000.

Answer:

```
SELECT customer_id, COUNT(order_id)
AS order_count, SUM(order_amount) AS
total_spent
FROM orders
GROUP BY customer_id
HAVING COUNT(order_id) > 5
   AND SUM(order_amount) > 1000;
```

Explanation: The `WHERE` clause filters rows before grouping, while `HAVING` filters groups after aggregation. Note the use of `COUNT(order_id)` rather than `COUNT(*)` to count only non-NULL order IDs, which is safer if there could be NULLs. Also, `SUM(order_amount)` is calculated within the `HAVING` clause; you can also use an alias, but not all databases allow that in `HAVING` (standard SQL does not permit it).

Subqueries and Common Table Expressions

(CTEs)

Complex queries often require breaking the problem into smaller parts. Subqueries (nested SELECTs) and CTEs (WITH clause) are essential tools.

Question: Retrieve the names of employees whose salary is greater than the average salary of their respective department. Assume table: *employees (id, name, salary, department_id)*.

Answer using a correlated subquery:

```
SELECT e1.name, e1.salary,
e1.department_id
FROM employees e1
WHERE e1.salary > (SELECT
AVG(e2.salary)
                   FROM employees e2
                   WHERE
e2.department_id = e1.department_id);
```

Answer using a CTE:

```
WITH dept_avg AS (
    SELECT department_id, AVG(salary)
AS avg_salary
    FROM employees
    GROUP BY department_id
)
SELECT e.name, e.salary,
e.department_id
FROM employees e
JOIN dept_avg d ON e.department_id =
d.department_id
WHERE e.salary > d.avg_salary;
```

Explanation: The correlated subquery is simpler but may be less efficient on large datasets because it executes the inner query for each row. The CTE approach calculates the average once per department then joins, often performing better. Both are valid and interviewers appreciate candidates who discuss performance trade-offs.

Handling Duplicates and Ranking

Duplicate detection and ranking are frequent real-world tasks. Window functions like **ROW_NUMBER()**, **RANK()**, **DENSE_RANK()** are increasingly common in interview questions.

Question: From the "sales" table (columns: sale_id, product_id, sale_date, amount), find the top 3 products by total sales amount. However, if there is a tie for the 3rd position, include all tied products.

Answer:

```
WITH product_totals AS (
    SELECT product_id, SUM(amount) AS
total_sales
    FROM sales
    GROUP BY product_id
),
ranked_products AS (
    SELECT product_id, total_sales,
           DENSE_RANK() OVER (ORDER
BY total_sales DESC) AS rnk
    FROM product_totals
)
SELECT product_id, total_sales
FROM ranked_products
WHERE rnk <= 3;
```

Explanation: DENSE_RANK() gives consecutive ranks even when there are ties (e.g., ranks 1,2,2,3). If we used RANK(), ties would skip a rank (1,2,2,4). ROW_NUMBER() would assign unique numbers arbitrarily, which is not desired. The CTE structure keeps the logic clean.

Date and Time Calculations

Many systems rely heavily on date ranges. Interviewers often ask for queries that involve date arithmetic, format conversions, or extracting parts of dates.

Question: Write a query to show the number of orders placed on each day of the week (Monday, Tuesday, etc.) from the "orders" table (order_date). Sort the result by day order (Monday first).

Answer:

```
SELECT
    DATENAME(dw, order_date) AS
day_name,
    COUNT(order_id) AS order_count
FROM orders
GROUP BY DATENAME(dw, order_date),
DATEPART(dw, order_date)
ORDER BY DATEPART(dw, order_date);
```

Note: The above uses SQL Server functions. In PostgreSQL, you would use `TO_CHAR(order_date, 'Day')` and `EXTRACT(DOW FROM order_date)`. In MySQL, `DAYNAME(order_date)` and `DAYOFWEEK(order_date)`. The key is to group by both the name and the numeric representation to avoid merging different week start conventions, and then order by the numeric day of week.

Self-Joins and Hierarchical Data

When tables reference themselves (e.g., an employee with a manager_id pointing to the same table), self-joins are necessary.

Question: Given an "employees" table with columns (id, name, manager_id), write a query to display each employee along with their manager's name. If the employee is the top manager (manager_id is NULL), show "CEO".

Answer:

```
SELECT e.name AS employee_name,
       COALESCE(m.name, 'CEO') AS
manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id
= m.id;
```

Explanation: The self-join uses aliases `e` and `m` (employee and manager). The `LEFT JOIN` ensures employees with no manager (NULL manager_id) are still included. `COALESCE` replaces NULL with 'CEO'.

Performance Considerations in Answers

Beyond getting the correct result, interviewers appreciate candidates who consider performance. When presenting an answer, you can mention potential issues like:

- Using `SELECT *` vs specifying columns (bad for production).
- Using `DISTINCT` unnecessarily (often a sign of poorly joined tables).
- Ensuring indexes are used (e.g., avoid wrapping indexed columns in functions in WHERE clauses if possible).
- Using `EXISTS` instead of `IN` for subqueries when checking existence (usually faster).
- Avoiding correlated subqueries when a join or CTE can achieve the same result more efficiently.

For instance, for the previous question about employees earning above department average, the CTE version is generally more scalable. Mentioning such nuances demonstrates depth of knowledge.

ADVANCED QUERY SCENARIOS

Some interviews push further with questions on pivoting data, recursive CTEs, or handling gaps in sequences.

Pivoting Data Without PIVOT

Question: The "sales" table has columns (year, quarter, amount). Write a query to display years as rows and quarters as columns (Q1, Q2, Q3, Q4) showing total sales per quarter per year.

Answer:

```
SELECT year,
       SUM(CASE WHEN quarter = 1 THEN
amount ELSE 0 END) AS Q1,
       SUM(CASE WHEN quarter = 2 THEN
amount ELSE 0 END) AS Q2,
       SUM(CASE WHEN quarter = 3 THEN
amount ELSE 0 END) AS Q3,
       SUM(CASE WHEN quarter = 4 THEN
amount ELSE 0 END) AS Q4
FROM sales
GROUP BY year
ORDER BY year;
```

Explanation: This manual pivot using CASE works in any SQL database and is a fundamental technique. Some databases have a PIVOT operator, but knowing