ChatGPT

Structured Framework for SQL Interview Questions (Data Analyst Roles)

1. Common SQL Question Types

Data analyst SQL interviews typically cover a broad range of query-writing skills. Common categories of SQL questions include [1]:

- **Basic Queries** (SELECT, WHERE, ORDER BY, LIMIT) – Simple data retrieval and filtering operations, sorting results, and limiting output.
- **Joins** (INNER, LEFT, RIGHT, FULL OUTER) – Combining rows from multiple tables based on related keys. Includes understanding differences between join types (e.g. inner vs. outer joins).
- **Aggregations** (GROUP BY, HAVING, COUNT, SUM, AVG, etc.) – Summarizing data across rows, grouping results by categories, and filtering grouped results.
- **Window Functions** (RANK, DENSE_RANK, ROW_NUMBER, LAG, LEAD, etc.) – Performing calculations across a set of rows related to the current row without collapsing them [2] (useful for ranking, running totals, etc.).
- **Common Table Expressions (CTEs) and Subqueries** – Writing nested queries or using `WITH` clauses to break complex problems into smaller parts, or to reuse result sets.
- **CASE Statements and Conditional Logic** – Using `CASE` (or `IF` logic) to compute conditional values or categorize data within a query.
- **Date and Time Functions** – Handling dates/times (e.g., computing date differences, extracting parts of dates like year or month, formatting dates).
- **String Manipulation** – Working with text (concatenation, substring extraction, pattern matching with LIKE, converting case, trimming, etc.).
- **Data Cleaning and Transformation** – Tasks for data quality, such as removing duplicates, handling NULL values (using functions like `COALESCE`), and converting data formats.

These categories align with the core SQL skills interviewers expect in data analyst roles [1]. Mastery of these fundamentals ensures you can tackle most SQL interview questions.

2. Example Problems and Solutions

Below are example interview-style SQL questions for each category, along with solutions and brief explanations.

Basic Queries

Example 1: *Retrieve all columns from the* `Customers` *table for customers located in California, sorted by last name.*
Solution: This query uses a `WHERE` clause to filter rows to California customers, then `ORDER BY` to sort by last name. We select all columns with `*` for brevity.

```sql
SELECT *
FROM Customers
WHERE state = 'California'
ORDER BY last_name;
```

Example 2: *List the first 5 products by highest price, showing the product name and price.*
Solution: Here we use `ORDER BY ... DESC` to sort prices from highest to lowest, then `LIMIT 5` to return only the top five.

```sql
SELECT product_name, price
FROM Products
ORDER BY price DESC
LIMIT 5;
```

Joins

Example 1: *Retrieve a list of all orders with the customer's first and last name. Assume a* `Customers` *table and an* `Orders` *table, where* `Orders.customer_id` = `Customers.customer_id`.
Solution: An **INNER JOIN** is used to combine orders with their matching customer. We select fields from both tables to display order details alongside customer name.

```sql
SELECT o.order_id, o.order_date, c.first_name, c.last_name
FROM Orders AS o
INNER JOIN Customers AS c
  ON o.customer_id = c.customer_id;
```

Example 2: *Find all customers who have not placed any orders.*
Solution: This uses a **LEFT JOIN** from Customers to Orders. Customers with no orders will have `NULL` in fields from the Orders side; the `WHERE o.order_id IS NULL` filter keeps only those cases.

```sql
SELECT c.customer_id, c.first_name, c.last_name
FROM Customers AS c
LEFT JOIN Orders AS o
  ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

Aggregations

Example 1: *Calculate the total number of orders each customer has placed.*
Solution: We use `COUNT(*)` with `GROUP BY` to get one row per customer. Each group's count represents that customer's total orders.

```sql
SELECT customer_id, COUNT(*) AS order_count
FROM Orders
GROUP BY customer_id;
```

Example 2: *List customer IDs that have more than 5 orders.*
Solution: This extends the previous query by adding a `HAVING` clause to filter groups. Only customers with `COUNT(*) > 5` are returned.

```sql
SELECT customer_id
FROM Orders
GROUP BY customer_id
HAVING COUNT(*) > 5;
```

Window Functions

Example 1: *For each department, rank employees by salary (1 = highest salary in that department).*
Solution: We use `ROW_NUMBER()` window function partitioned by department. The window function computes a rank for each row within its department partition, ordered by salary descending. (Unlike a regular aggregate, this does **not** collapse the rows – each employee remains in the result with their rank [2].)

```sql
SELECT department_id,
       employee_id,
       salary,
       ROW_NUMBER() OVER (
           PARTITION BY department_id
           ORDER BY salary DESC
       ) AS salary_rank
FROM Employees;
```

Example 2: *Given a* `Sales` *table with columns (*`sale_date`*,* `amount`*) representing total sales for each date, show each date's sales, the previous day's sales, and the day-over-day difference.*
Solution: The `LAG` window function is used to access the previous row's sales (ordered by date). We then calculate the difference. This preserves the row-wise detail while peeking at a prior row's value.

```sql
SELECT
    sale_date,
    amount AS sales,
    LAG(amount, 1) OVER (ORDER BY sale_date) AS prev_day_sales,
    (amount
     - LAG(amount, 1) OVER (ORDER BY sale_date)) AS sales_diff
```

```
FROM Sales
ORDER BY sale_date;
```

Common Table Expressions (CTEs) and Subqueries

Example 1: *Find the employees who earn more than the average salary of all employees.*
Solution: We can use a **subquery** in the `WHERE` clause to get the overall average salary, and compare each employee's salary to it.

```
SELECT name, salary
FROM Employees
WHERE salary > (
    SELECT AVG(salary)
    FROM Employees
);
```

Example 2: *List products that have generated more than $10,000 in total sales. Assume an* `OrderItems` *table with* `product_id` *and* `amount` *for each sold item.*
Solution: We use a **CTE** to calculate total sales per product, then in the outer query filter those with `total_sales > 10000`. This approach breaks the problem into two steps, improving clarity.

```
WITH ProductSales AS (
    SELECT product_id, SUM(amount) AS total_sales
    FROM OrderItems
    GROUP BY product_id
)
SELECT product_id, total_sales
FROM ProductSales
WHERE total_sales > 10000;
```

CASE Statements and Conditional Logic

Example 1: *Classify each order as* `"High Value"` *if the amount > 100, or* `"Low Value"` *otherwise.*
Solution: A `CASE` expression in the SELECT list applies the condition and outputs a label per row. This adds a new derived column `value_category` based on the order amount.

```
SELECT order_id,
       amount,
       CASE
         WHEN amount > 100 THEN 'High Value'
         ELSE 'Low Value'
```

```
        END AS value_category
FROM Orders;
```

Example 2: *Count how many orders are Domestic vs. International. (Assume the* `Orders` *table has a* `country` *column and "USA" denotes domestic).*
Solution: We can use CASE inside aggregate functions to sum up conditional counts. Each `SUM(CASE ...)` adds 1 for each row meeting the condition.

```
SELECT
    SUM(CASE WHEN country = 'USA' THEN 1 ELSE 0 END) AS domestic_orders,
    SUM(CASE WHEN country <> 'USA' THEN 1 ELSE 0 END) AS international_orders
FROM Orders;
```

Date and Time Functions

Example 1: *Show each order's ID, order date, delivery date, and the number of days between order and delivery.*
Solution: Use a date difference function to compute the interval between two dates. For example, in many SQL dialects `DATEDIFF(end_date, start_date)` returns the day difference.

```
SELECT
    order_id,
    order_date,
    delivery_date,
    DATEDIFF(delivery_date, order_date) AS days_between
FROM Orders;
```

Explanation: Here `DATEDIFF` (syntax may vary by SQL flavor) calculates the days between `order_date` and `delivery_date` for each order.

Example 2: *Calculate the number of orders made each month in 2024.*
Solution: We extract the month and year from the date and aggregate. The `EXTRACT()` function (ANSI SQL) pulls out parts of a date (here month and year). We filter to year 2024, group by month, and count orders in each month.

```
SELECT
    EXTRACT(MONTH FROM order_date) AS order_month,
    COUNT(*) AS orders_count
FROM Orders
WHERE EXTRACT(YEAR FROM order_date) = 2024
GROUP BY EXTRACT(MONTH FROM order_date)
ORDER BY order_month;
```

String Manipulation

Example 1: *Extract the domain (portion after '@') from each user's email address in the* `Users` *table.*
Solution: We use string functions to find the position of '@' and then take the substring that follows. In MySQL, for instance, `SUBSTRING_INDEX(email, '@', -1)` returns the substring after the last `'@'`.

```
SELECT
    email,
    SUBSTRING_INDEX(email, '@', -1) AS domain
FROM Users;
```

Example 2: *Combine* `first_name` *and* `last_name` *into a single field called* `full_name` *for each customer.*
Solution: String concatenation can be done with functions or operators. Here we use the `CONCAT()` function to join first and last name with a space in between.

```
SELECT
    CONCAT(first_name, ' ', last_name) AS full_name
FROM Customers;
```

Data Cleaning and Transformation

Example 1: *Identify duplicate email addresses in the* `Customers` *table (i.e., emails that appear more than once).*
Solution: Group by the email and use `COUNT` to find occurrences. The `HAVING COUNT(*) > 1` clause filters to only those email groups that have more than one entry (duplicates).

```
SELECT email, COUNT(*) AS occurrences
FROM Customers
GROUP BY email
HAVING COUNT(*) > 1;
```

Example 2: *Show a list of products with their prices, but replace any NULL price with 0 as a default value.*
Solution: Use the `COALESCE` function to substitute a fallback value for NULLs. This query will return a price or 0 if the price is NULL, effectively "cleaning" missing values.

```
SELECT
    product_id,
    COALESCE(price, 0) AS price
FROM Products;
```

3. Problem-Solving Approach

When tackling SQL problems in an interview, it's important to approach them systematically and communicate your thought process. Here are some best practices:

- **Understand the Question and Data:** Carefully read the problem to determine exactly what is being asked. Identify the required output (columns, conditions, ordering) before writing any SQL. Often, interview questions include extra details; make sure to clarify the goal. If you're unsure about the schema or assumptions (e.g. which tables to use, or expected output format), ask clarifying questions [3]. Many candidates make the mistake of jumping into coding without fully understanding the context [4], so take a moment to confirm your interpretation of the question.

- **Break the Problem Down:** Decompose complex queries into smaller steps. Determine which SQL concepts are needed for each part of the problem. For example, ask yourself: Does the problem require filtering (WHERE)? Joining multiple tables? Aggregating results? Sorting or limiting output? If multiple operations are needed, you can build the query step-by-step. Start with a simple SELECT for the core table, then gradually add JOINs or subqueries, then GROUP BY or window functions as required. Using **CTEs** (`WITH` clauses) can help structure the solution in logical steps, which makes the query easier to write and read.

- **Consider Edge Cases:** Think about unusual or extreme scenarios in the data and how your query should handle them. For example, consider whether any columns might be NULL and if so, whether you need to handle that (e.g., using `COALESCE` or filtering out NULLs). If the question involves division or averages, ensure there's no risk of dividing by zero or an empty denominator. Consider duplicates if they might affect the result (do you need `DISTINCT` or a specific grouping to avoid double-counting?). By anticipating these cases, you can ensure your query is robust. *(For instance, if calculating an average, you might want to* `COALESCE` *NULL values to 0 before averaging so they don't skew or nullify the result* [5]*.)* Always clarify with the interviewer if it's unclear how edge cases should be handled.

- **Write Clear and Readable SQL:** Use a clean coding style for your SQL query – not just for the interviewer's benefit, but also to avoid errors. Format your query with line breaks and indentation (e.g., each clause `SELECT`, `FROM`, `WHERE`, `GROUP BY`, etc., on a new line) so that the structure is easy to follow. Use aliases for tables and columns that have long names, but keep them meaningful (`emp` for an `Employees` table is fine, `x` is less clear). If using multiple joins or subqueries, consider adding brief comments (if the interview format allows) or at least spacing out sections to delineate them. A well-structured query is easier to debug and explain. Interviewers generally prefer clarity over cleverness – it's better to write a straightforward query that's easy to understand than an obscure one-liner. In fact, one SQL interviewer noted that they don't look for the *"fanciest or most elegant solution"* but rather a clear demonstration of correct application of SQL principles [5].

- **Build and Verify Step by Step:** If possible, test parts of your query as you go (in an interactive environment or at least mentally with sample data). For example, first write a query to get the raw data you need (such as a join or a subquery result), and verify it conceptually, then incorporate aggregation or additional logic. This incremental approach helps catch mistakes early. Even if you

cannot run the query in an interview, you can explain how you would test it with a small example dataset to verify the correctness of each step.

- **Communicate Your Thought Process:** As you work through the problem, explain to the interviewer what you're thinking. State any assumptions you are making about the data. If the question is ambiguous, describing how you interpret it (and confirming that) shows good communication. Walk through how you plan to tackle the query (e.g., "First, I'll join these tables to get the necessary fields, then filter for last month, then aggregate by category..."). This not only helps you organize your approach but also lets the interviewer see your problem-solving method. Even if you get stuck, expressing your approach can earn you credit. Remember, in many cases *it's about problem-solving, not just perfection in syntax* [1] [6].

By following these practices, you systematically break down SQL questions and build correct, readable queries. Focus on understanding the problem, crafting a clear solution step-by-step, and handling any special cases. This approach will demonstrate your analytical thinking and SQL proficiency to the interviewer. Good luck!

---

[1] [6] Interview questions for SQL : r/SQL
https://www.reddit.com/r/SQL/comments/191lesk/interview_questions_for_sql/

[2] Ultimate SQL Interview Guide For Data Scientists & Data Analysts
https://datalemur.com/blog/sql-interview-guide

[3] [4] [5] SQL interview questions? : r/SQL
https://www.reddit.com/r/SQL/comments/o935sn/sql_interview_questions/