SQL for Data Analysis — Technical Task Submission

Elevate Labs- Task Day – 4

Objective

Leverage SQL to extract, manipulate, and analyze structured data from a relational database using industry-standard practices. The goal is to demonstrate end-to-end competency in querying, transforming, and summarizing data using SQL in a performance-conscious manner.

☑ ☑ Deliverables

- .sql file containing all SQL queries
- Screenshots showing output for each query
- Accompanying README.md file describing logic and query purpose GitHub repo
- containing all of the above

Tooling & Database

- ☑ SQL Engine: PostgreSQL / MySQL / SQLite (compatible with standard ANSI SQL syntax)
- ☑ Mock Dataset: E-commerce-based schema with tables like orders, customers, products, categories, etc.

Structured Responses — Interview Questions

1. Difference Between WHERE and HAVING Clauses

-
- HAVING filters data **after** a GROUP BY has aggregated rows. It applies to grouped/aggregated data.

```
-- Using WHERE
SELECT * FROM orders
WHERE order_status = 'Completed' ;

-- Using HAVING
SELECT customer_id, COUNT(*) AS total_orders
FROM orders
GROUP BY customer_id
HAVING COUNT(*) >= 5;
```

WHERE filters data **before** rows are grouped. It applies to individual rows.

2. Types of Joins in SQL

A relational system supports multiple types of joins to bring together data from multiple tables:

- **INNER JOIN**: Returns only records with matched keys.
- **LEFT JOIN**: All rows from the left table + matched rows from the right.
- **RIGHT JOIN**: All rows from the right table + matched rows from the left.
- **FULL OUTER JOIN**: Includes unmatched rows from both tables.
- **CROSS JOIN**: Produces a Cartesian product.

```
-- INNER JOIN Example
SELECT c.name, o.order_id
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

3. Average Revenue Per User

To calculate ARPU Average Revenue Per User), we first aggregate order amounts per customer, then compute the average across those customers.

```
SELECT
    AVG(customer_total) AS avg_revenue_per_user
FROM (
    SELECT customer_id, SUM(order_amount) AS customer_total
    FROM orders
    GROUP BY customer_id
) AS revenue_per_user;
```

4. What Are Subqueries?

A subquery is an embedded query used inside another SQL statement. It can return scalar values, rows, or even act as a temporary table.

Use Cases:

- **In WHERE**:

```
SELECT * FROM customers
WHERE customer_id IN (
    SELECT customer_id FROM orders WHERE order_amount > 5000
);
```

- **In FROM Derived Table)**:

```
SELECT region, AVG(total_spending )
FROM (
    SELECT region, SUM(order_amount) AS total_spending
    FROM orders
    GROUP BY region
) AS region_summary
GROUP BY region;
```

5. Query Optimization Techniques

Creating efficient SQL is critical when working with large datasets.

Best Practices:

- Use **indexes** on columns in WHERE, JOIN and ORDER BY clauses.
- Replace `SELECT *` with explicit columns.
- Use `WHERE` conditions to filter data early.
- Avoid non-SARGable expressions (functions on indexed columns).
- Analyze query plans with `EXPLAIN`.
- Consider **denormalized views** for aggregated reports.

```
-- Index Example :
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
```

6. SQL Views: Simplifying Reuse and Security

A **view** is a saved, named SQL query. It is logical (virtual) and helps encapsulate complex joins or aggregations, making it easier for downstream analysts or BI tools.

```
CREATE VIEW monthly_revenue AS
SELECT
    DATE_TRUNC( 'month' , order_date) AS order_month,
    SUM(order_amount) AS total_revenue
FROM orders
GROUP BY order_month;
```

Uses:

- Abstract business logic
- Control data access
- Build reusable insights

7. Handling NULL Values in SQL

- **IS NULL / IS NOT NULL** for filtering
- **COALESCE()**, **IFNULL()**, **NVL()** to substitute defaults
- Be cautious during aggregations (e.g., `AVG()`, `SUM()` ignore `NULLs`)

```
SELECT email FROM customers WHERE email IS NULL;

-- Replace null phone values
SELECT customer_id, COALESCE(phone, 'N/A') AS phone_contact
FROM customers;
```

Sample Analytical Queries on E-commerce Schema

a) Total Sales by Product

```
SELECT product_id, SUM(order_amount) AS total_sales
FROM orders
GROUP BY product_id
ORDER BY total_sales DESC;
```

b) Top 5 Customers by Lifetime Spend

```
SELECT customer_id, SUM(order_amount) AS customer_lifetime_value
FROM orders
GROUP BY customer_id
ORDER BY customer_lifetime_value DESC
LIMIT 5;
```

c) Sales Breakdown by Product Category

Assumes existence of `products` and `categories` tables.

```
SELECT
    cat.category_name,
    SUM(o.order_amount) AS total_category_sales
FROM orders o
JOIN products p ON o.product_id = p.product_id
JOIN categories cat ON p.category_id = cat.category_id
GROUP BY cat.category_name;
```

d) Improving Query Speed Using Indexes

```
-- Add index on product_id for faster GROUP BY or JOIN
CREATE INDEX idx_product_id ON orders(product_id);
```

e) Reusable Monthly Sales View

```
-- SQLite version using strftime
CREATE VIEW monthly_sales AS
SELECT
    strftime('%Y-%m', order_date) AS order_month,
    SUM(order_amount) AS monthly_total
FROM orders
GROUP BY order_month;
```

Submission Checklist

- ☑ All SQL queries saved in: `ecommerce_analysis.sql`
- ☑ Screenshots of output added to `/screenshots/` directory

☑ README with technical explanations and context

☑ Pushed to GitHub under a new public repo (name: `ecommerce-sql-analysis`)

Tips Before Submission

- If using a sample dataset: try **Chinook**, **Northwind**, or **open-source ecommerce schema**.
- Prefer running queries in a GUI like DBeaver, pgAdmin, or SQLite Browser for easier screenshotting.
- Use consistent formatting and SQL style guide (capitalized keywords, snake_case naming).

Let me know if you'd like help designing your own simple ecommerce schema or populating mock data using SQL `INSERT` statements.

Prepared by:

Sahil Virendra Tikait
Pune, Maharashtra — Data Analytics Professional
Submission Date: August 8th, 2025