amazon SQL

Interview questions

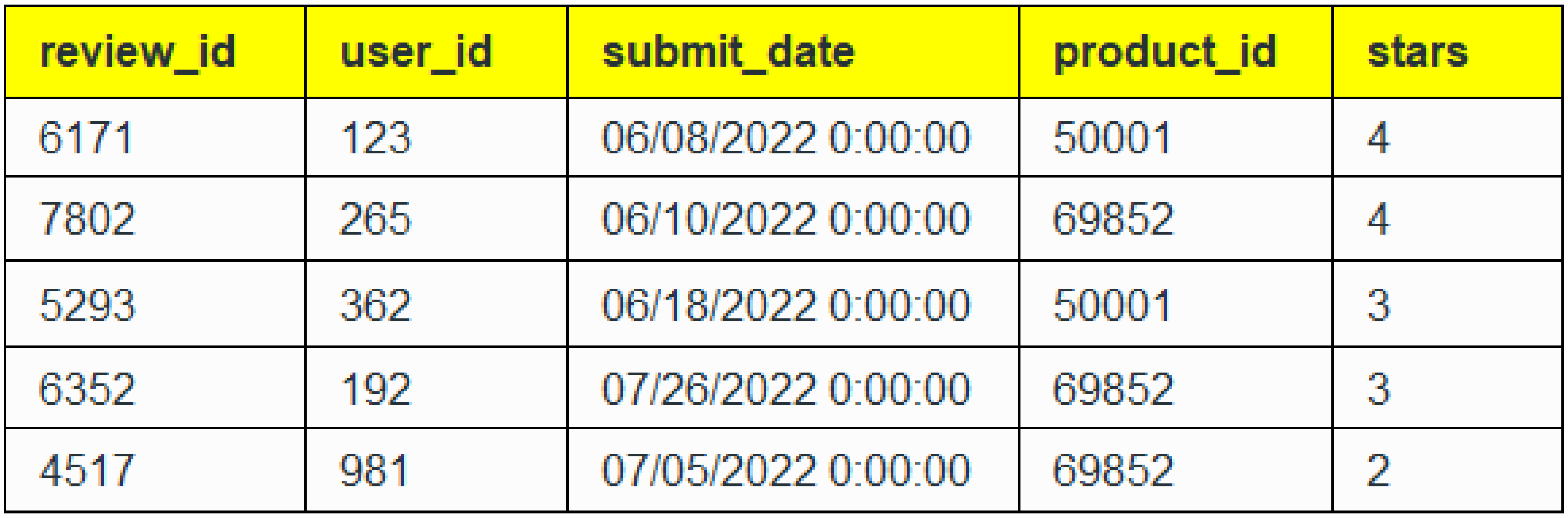
for Data Analysts

Part I

linkedin.com/in/ileonjose

1. Average Review Ratings per Product per Month

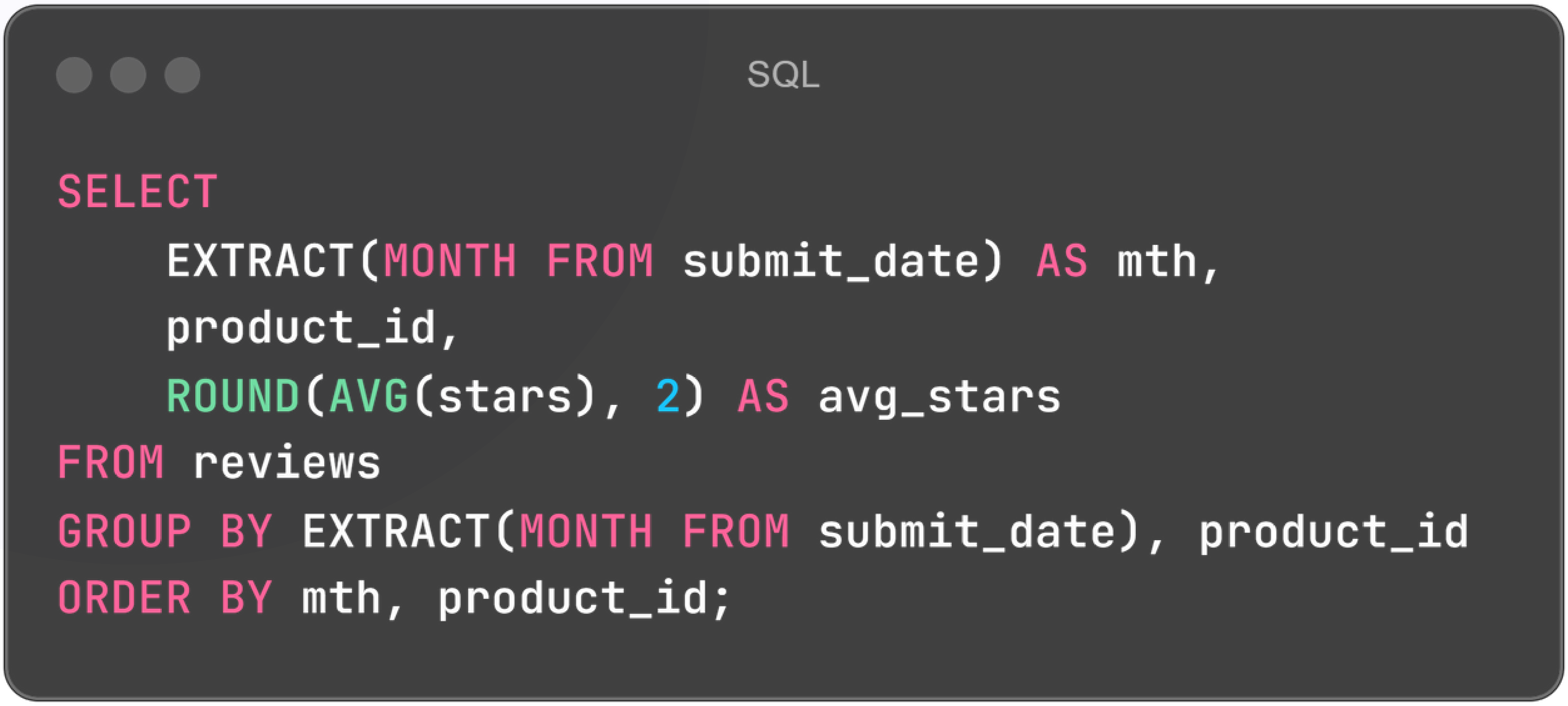
- **Problem Statement:** Given a table of product reviews, calculate the average review rating for each product for every month. The data is in a reviews table, which includes *review_id*, *user_id*, *submit_date*, *product_id*, and *stars*. The output should list the month (as a numerical value), *product_id*, and the average star rating rounded to two decimal places. Sort the result by month and then by *product_id*.



review_id	user_id	submit_date	product_id	stars
6171	123	06/08/2022 0:00:00	50001	4
7802	265	06/10/2022 0:00:00	69852	4
5293	362	06/18/2022 0:00:00	50001	3
6352	192	07/26/2022 0:00:00	69852	3
4517	981	07/05/2022 0:00:00	69852	2

How to Solve:

- Extract the month from the *submit_date* using the *EXTRACT* function.
- Group the results by the extracted month and *product_id*.
- Compute the average star rating for each group and round the result to two decimal places.
- Order the output by month and *product_id*.



```sql
SELECT
    EXTRACT(MONTH FROM submit_date) AS mth,
    product_id,
    ROUND(AVG(stars), 2) AS avg_stars
FROM reviews
GROUP BY EXTRACT(MONTH FROM submit_date), product_id
ORDER BY mth, product_id;
```

2. Optimizing a Slow SQL Query

- **Problem Statement:** Amazon handles massive datasets, and optimizing SQL queries is crucial for performance. Discuss various methods to optimize a slow SQL query.

How to Solve:

- Select Specific Fields: Use *SELECT field1, field2* instead of *SELECT ** to retrieve only necessary columns.

- Avoid SELECT DISTINCT: Use *DISTINCT* only when absolutely needed, as it can be expensive.

- Use INNER JOIN: Prefer *INNER JOIN* over using multiple *WHERE* clauses to join tables.

- Minimize Joins: Where possible, denormalize the data to reduce the need for complex joins.

- Add Indexes: Create indexes on columns that are frequently used in *WHERE* clauses and joins to speed up queries.

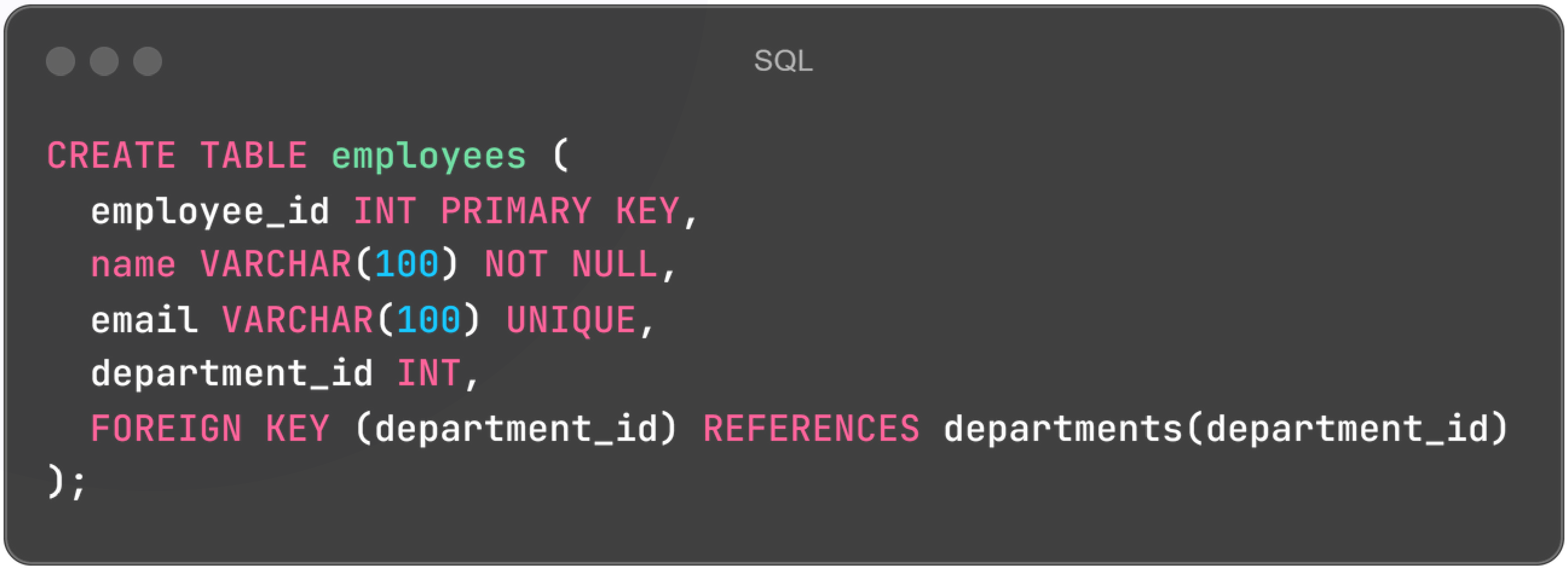
- Examine Execution Plans: Use the SQL query execution plan to identify bottlenecks and optimize accordingly.

3. SQL Constraints

- **Problem Statement:** Explain SQL constraints and provide examples of different types of constraints used to enforce data integrity in databases.

How to Solve:

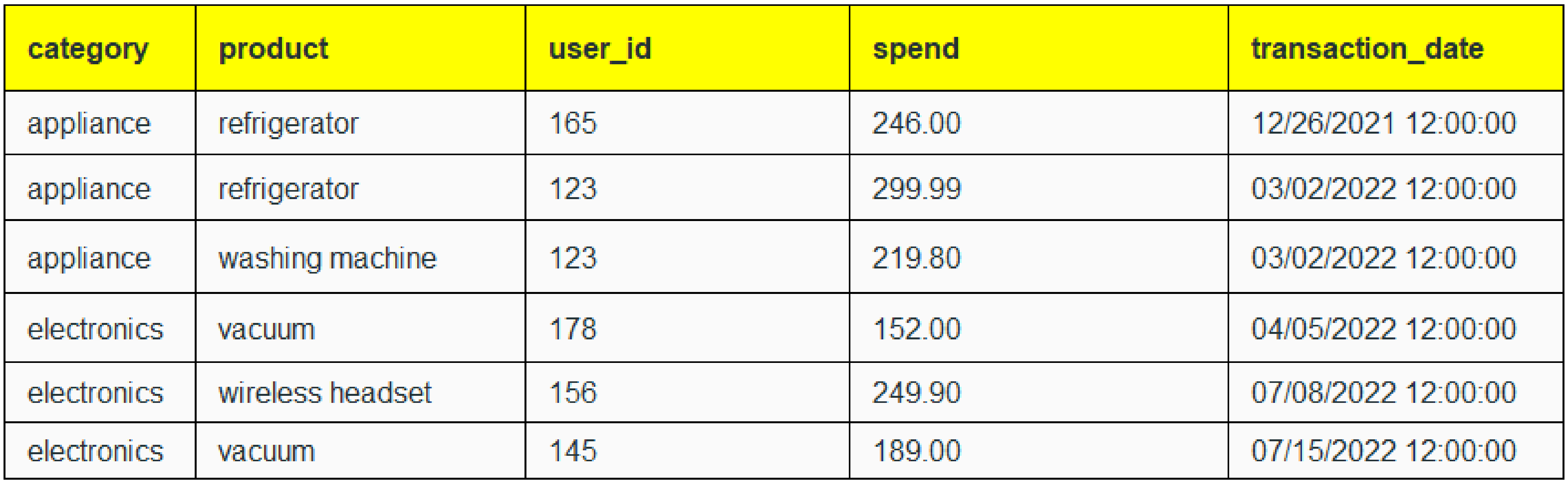
- NOT NULL: Ensures that a column cannot have NULL values.
- UNIQUE: Ensures all values in a column are unique.
- INDEX: Improves query performance by indexing frequently queried columns.
- PRIMARY KEY: Uniquely identifies each record in a table.
- FOREIGN KEY: Ensures referential integrity between tables.



```sql
CREATE TABLE employees (
  employee_id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE,
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

4. Highest-Grossing Items

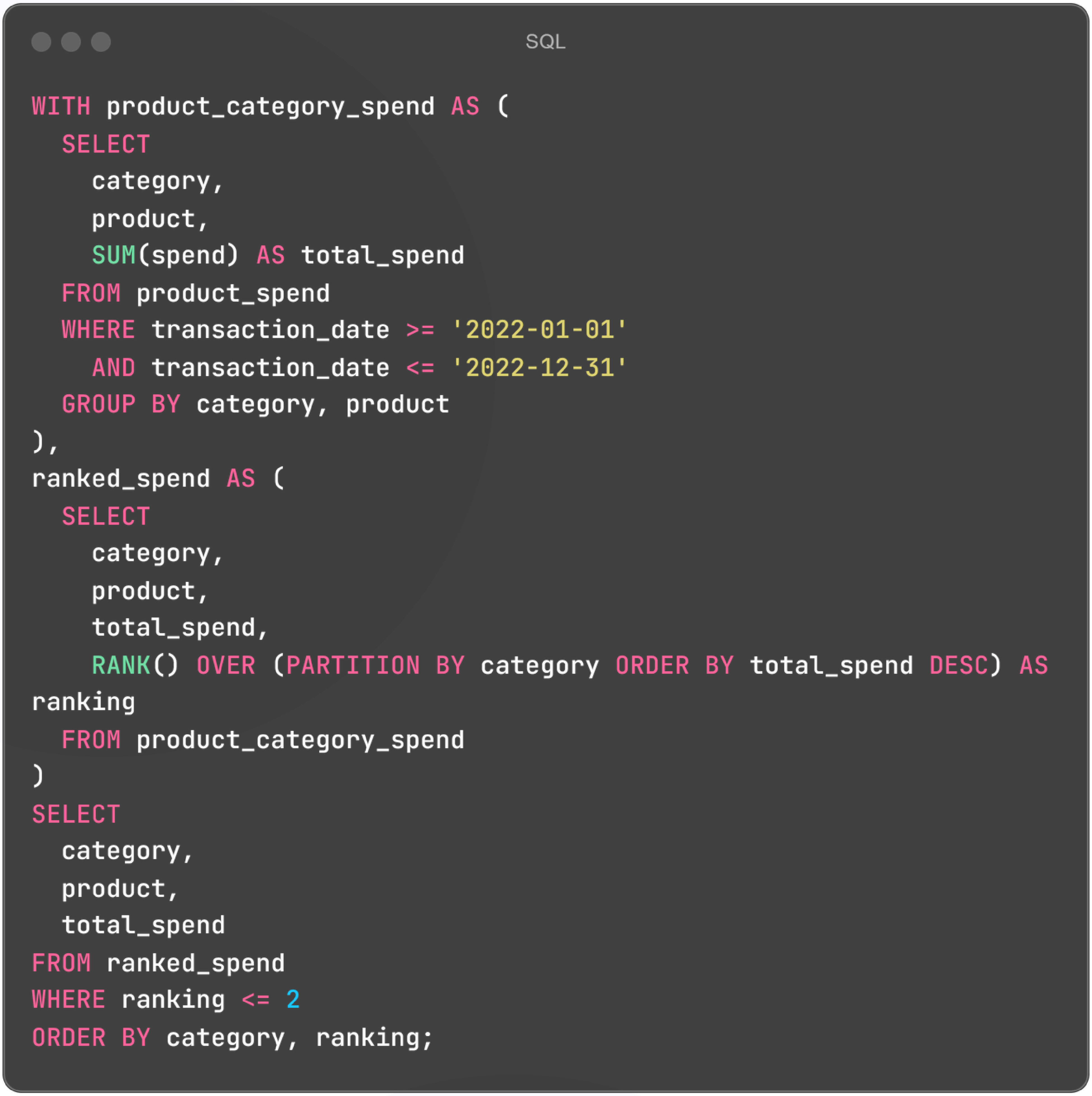
- **Problem Statement:** Find the top two highest-grossing products in each category for the year 2022 from a table *product_spend*. The table contains *category*, *product*, *user_id*, *spend*, and *transaction_date*. The output should include the *category*, *product*, and total spend.



category	product	user_id	spend	transaction_date
appliance	refrigerator	165	246.00	12/26/2021 12:00:00
appliance	refrigerator	123	299.99	03/02/2022 12:00:00
appliance	washing machine	123	219.80	03/02/2022 12:00:00
electronics	vacuum	178	152.00	04/05/2022 12:00:00
electronics	wireless headset	156	249.90	07/08/2022 12:00:00
electronics	vacuum	145	189.00	07/15/2022 12:00:00

How to Solve:

- Step 1: Aggregate the total spend by category and product for 2022.
- Step 2: Use a Common Table Expression (CTE) to rank the products within each category based on total spend.
- Step 3: Filter the results to include only the top two products per category.



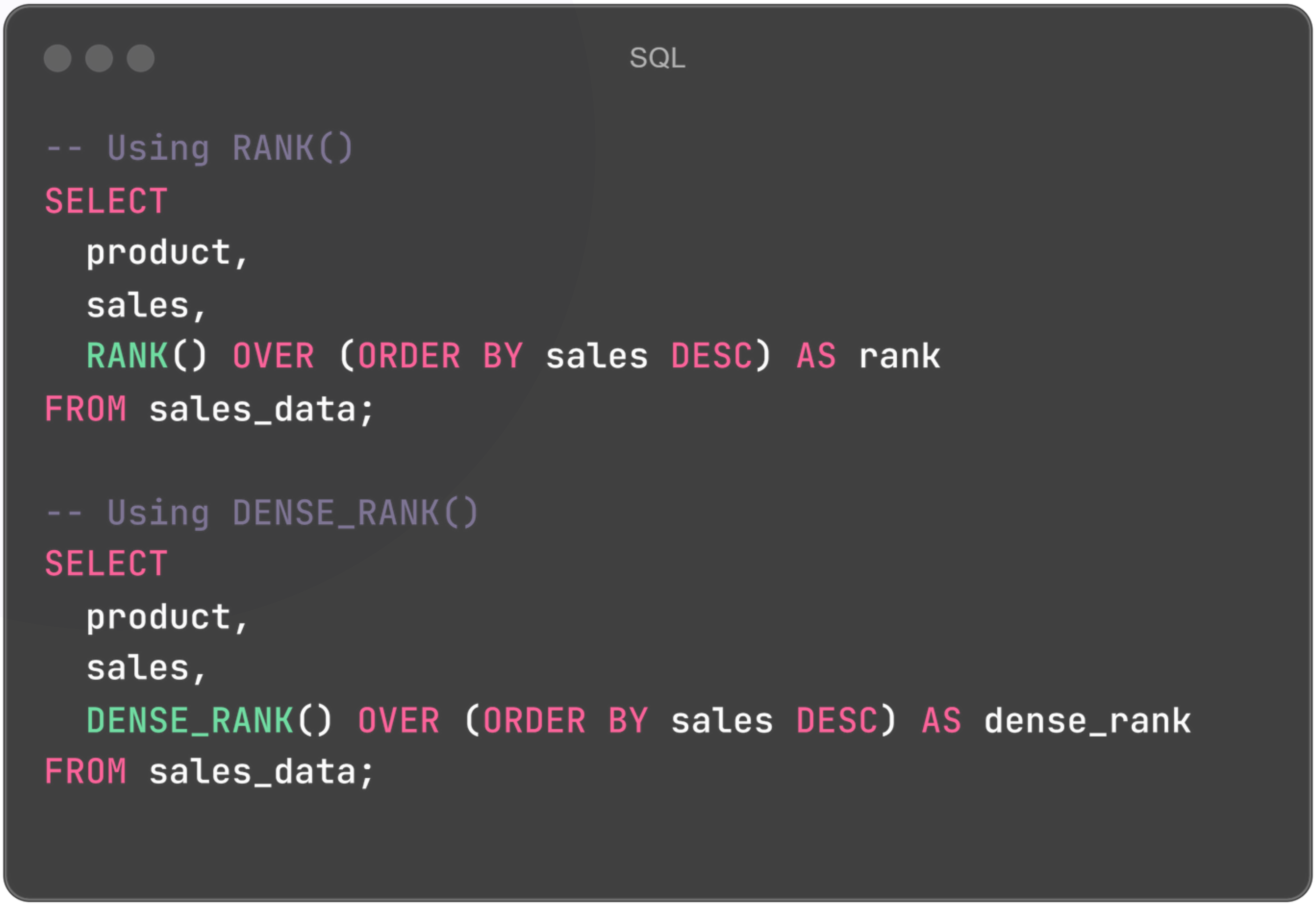
```sql
WITH product_category_spend AS (
  SELECT
    category,
    product,
    SUM(spend) AS total_spend
  FROM product_spend
  WHERE transaction_date >= '2022-01-01'
    AND transaction_date <= '2022-12-31'
  GROUP BY category, product
),
ranked_spend AS (
  SELECT
    category,
    product,
    total_spend,
    RANK() OVER (PARTITION BY category ORDER BY total_spend DESC) AS
ranking
  FROM product_category_spend
)
SELECT
  category,
  product,
  total_spend
FROM ranked_spend
WHERE ranking <= 2
ORDER BY category, ranking;
```

5. Difference Between RANK() and DENSE_RANK()

Problem Statement: Explain the difference between the RANK() and DENSE_RANK() functions in SQL.

- RANK(): Assigns a unique rank to each row within a partition of a result set. If there are ties, the rank values will have gaps (e.g., if two items are ranked 2, the next rank will be 4).

- DENSE_RANK(): Similar to *RANK()*, but does not leave gaps between ranks. If two items are ranked 2, the next rank will be 3.



```sql
-- Using RANK()
SELECT
  product,
  sales,
  RANK() OVER (ORDER BY sales DESC) AS rank
FROM sales_data;

-- Using DENSE_RANK()
SELECT
  product,
  sales,
  DENSE_RANK() OVER (ORDER BY sales DESC) AS dense_rank
FROM sales_data;
```

Found this helpful? Repost!

linkedin.com/in/ileonjose