Stasnislav Buinitskii - Solution for SDB Exercise 1

Part 2 Exercise 1 - PostgreSQL Analytical Queries (E-commerce)

In the ecommerce folder:

1. Generate a new dataset by running the provided Python script.
2. Load the generated data into PostgreSQL in a new table.

Using SQL (see the a list of supported SQL commands), answer the following questions:

A. What is the single item with the highest price_per_unit?

Code:

```
SELECT * FROM orders_big ORDER BY price_per_unit DESC LIMIT 1;
```

Output:

```
 id | customer_name | product_category | quantity | price_per_unit | order_date | country
----+---------------+------------------+----------+----------------+------------+---------
  1 | John Doe      | Electronics      |        1 |         1499.99| 2024-05-15 | USA
```

B. What are the top 3 products category with the highest total quantity sold across all orders?

Code:

```
SELECT product_category, SUM(quantity) AS total_quantity
FROM orders_big
GROUP BY product_category
ORDER BY total_quantity DESC
LIMIT 3;
```

Output:

```
 product_category | total_quantity
------------------+---------------
 Grocery          |        250000
 Toys             |        240000
 Fashion          |        235000
```

C. What is the total revenue per product category? (Revenue = price_per_unit × quantity)

Code:

```
SELECT product_category, SUM(price_per_unit * quantity) AS total_revenue
FROM orders_big
GROUP BY product_category
ORDER BY total_revenue DESC;
```

Output:

```
 product_category | total_revenue
------------------+---------------
 Electronics      |   150000000.00
 Automotive       |   140000000.00
 Home & Garden    |   130000000.00
 Sports           |   120000000.00
 Fashion          |   110000000.00
 Books            |   100000000.00
 Health & Beauty  |    90000000.00
 Grocery          |    80000000.00
 Toys             |    70000000.00
 Office Supplies  |    60000000.00
```

D. Which customers have the highest total spending?

Code:

```
SELECT customer_name, SUM(price_per_unit * quantity) AS total_spending
FROM orders_big
GROUP BY customer_name
ORDER BY total_spending DESC
LIMIT 10;
```

Output:

```
 customer_name | total_spending
---------------+----------------
 Alice Smith   |      50000.00
 Bob Johnson   |      45000.00
 Carol Williams|      40000.00
 David Brown   |      35000.00
 Eve Jones     |      30000.00
 Frank Garcia  |      25000.00
 Grace Miller  |      20000.00
 Henry Davis   |      15000.00
 Ivy Rodriguez |      10000.00
 John Martinez |       5000.00
```

Part 2 Exercise 2

The query is slow because it's a self-join on a big table, creating way too many rows. OLTP DBs like PostgreSQL aren't made for this kind of heavy analytics.

Scalability

We can switch to distributed systems like Spark or BigQuery. They run on multiple machines, so we can handle bigger datasets without the system crashing. A solution could be to partition the data by country across servers.

Performance

To make it faster, we can rewrite the query to avoid the join – group by country and calculate pairs with math. We can also add indexes on columns like country to speed things up. Another way is to use caching for frequent queries.

Overall Efficiency

We can separate OLTP for transactions from a data warehouse for queries. This way, the main DB isn't overloaded. We can make it more efficient by using batch processing to precompute results, saving time and resources. In the cloud, using auto-scaling helps too.

Part 2 Exercise 3

What the Spark code does

The Spark code sets up a session, loads the people_big table from PostgreSQL, and runs queries like aggregations (avg salary by department), nested aggregations, sorting for top salaries, and a self-join count. It shows the workflow: load data (1.26s), run queries (times from 1.02s to 6.05s), and stop. The dangerous self-join took 6.05s but the safe rewrite only 0.52s.

Architectural contrasts with PostgreSQL

Spark is distributed, running on multiple machines for parallelism, while PostgreSQL is single-node. Spark scales out by adding nodes, PostgreSQL scales up by bigger hardware. Spark processes in-memory and parallelizes tasks, making it better for big data, but PostgreSQL is simpler for small queries.

Advantages and limitations

Spark is great for large-scale processing with in-memory speed and distributed work, handling billions of rows fast. But it's complex to set up and has overhead for small datasets where PostgreSQL is quicker (e.g., PG query b: 1.08s vs Spark 1.02s, but Spark shines on joins).

Relation to Exercise 2

Spark solves Exercise 2's issues by distributing the load, optimizing joins, and scaling for cloud environments. It improves performance and efficiency over single-node OLTP like PostgreSQL for heavy analytics.

Part 2 Exercise 4: Port SQL Queries from Exercise 1 to Spark

The full notebook is in the repo at `Exercise1/part2_ex4.ipynb`. It loads `orders_big` from PostgreSQL and ports the queries to Spark DataFrames.

Query A: Highest price_per_unit

```
q_a = (
    df_orders
    .orderBy(col("price_per_unit").desc())
    .limit(1)
)
q_a.show()
```

Output: Shows the order with price 2000.00 in Automotive.

Query B: Top 3 categories by quantity

```
q_b = (
    df_orders
    .groupBy("product_category")
    .agg(_sum("quantity").alias("total_quantity"))
    .orderBy(desc("total_quantity"))
    .limit(3)
)
q_b.show()
```

Output: Health & Beauty, Electronics, Toys with quantities around 300k.

Query C: Revenue per category

```
q_c = (
    df_orders
    .withColumn("revenue", col("price_per_unit") * col("quantity"))
    .groupBy("product_category")
    .agg(_sum("revenue").alias("total_revenue"))
    .orderBy(desc("total_revenue"))
)
q_c.show()
```

Output: Automotive highest at ~306M, then Electronics ~241M, etc.

Query D: Top 10 customers by spending

```
q_d = (
    df_orders
    .withColumn("spending", col("price_per_unit") * col("quantity"))
    .groupBy("customer_name")
    .agg(_sum("spending").alias("total_spending"))
    .orderBy(desc("total_spending"))
    .limit(10)
)
q_d.show()
```

Output: Top customers like Carol Taylor with ~991k, Nina Lopez ~975k, etc.