SBD3 – Exercise 1

- Mohamed HAROUN - S2410929012

Repository setup

- Forked the repository provided by the professor
- Cloned it locally
- Created a `solution/` folder
- Copied the provided exercise skeleton into `solution/` to keep my work separated from the original files

Part 1 – Environment setup and PostgreSQL basics

Start environment

Started the Docker environment:

```
docker compose up -d
```

Verified that the containers were running (`pg-bigdata`, `spark-lab`).

Generate large dataset

Generated a synthetic dataset with 1 million rows:

```
cd data
python3 expand.py
wc -l people_1M.csv
```

- File size: ~45 MB
- Rows: 1,000,000 (+ header)

Load data into PostgreSQL

Connected to PostgreSQL:

```
docker exec -it pg-bigdata psql -U postgres
```

Created and loaded the table:

```
DROP TABLE IF EXISTS people_big;
```

```sql
CREATE TABLE people_big (
  id SERIAL PRIMARY KEY,
  first_name TEXT,
  last_name TEXT,
  gender TEXT,
  department TEXT,
  salary INTEGER,
  country TEXT
);

\COPY people_big(first_name,last_name,gender,department,salary,country)
FROM '/data/people_1M.csv' DELIMITER ',' CSV HEADER;
```

Enabled timing and verified data:

```sql
\timing on
SELECT COUNT(*) FROM people_big;
SELECT * FROM people_big LIMIT 5;
```

Result:

- Rows loaded: 1,000,000

id	first_name	last_name	gender	department	salary	country
1	Andreas	Scott	Male	Audit	69144	Bosnia
2	Tim	Lopez	Male	Energy Management	62082	Taiwan
3	David	Ramirez	Male	Quality Assurance	99453	South Africa
4	Victor	Sanchez	Male	Level Design	95713	Cuba
5	Lea	Edwards	Female	Energy Management	60425	Iceland

Analytical queries

(a) Average salary per department

```sql
SELECT department, AVG(salary)
FROM people_big
GROUP BY department
LIMIT 10;
```

department	avg
Accounting	85150.560834888851
Alliances	84864.832756437315

department	avg
Analytics	122363.321232406454
API	84799.041690986409
Audit	84982.559610499577
Backend	84982.349086542585
Billing	84928.436430727944
Bioinformatics	85138.080510264425
Brand	85086.881434454358
Business Intelligence	85127.097446808511

> Execution time: ~250 ms

(b) Nested aggregation by country

```sql
SELECT country, AVG(avg_salary)
FROM (
  SELECT country, department, AVG(salary) AS avg_salary
  FROM people_big
  GROUP BY country, department
) sub
GROUP BY country
LIMIT 10;
```

country	avg
Algeria	87230.382040504578
Argentina	86969.866763623360
Armenia	87245.059590528218
Australia	87056.715662987876
Austria	87127.824046597584
Bangladesh	87063.832793583033
Belgium	86940.103641985310
Bolivia	86960.615658334041
Bosnia	87102.274664951815
Brazil	86977.731228862018

> Execution time: ~330–380 ms

(c) Top 10 salaries

```sql
SELECT *
FROM people_big
ORDER BY salary DESC
LIMIT 10;
```

id	first_name	last_name	gender	department	salary	country
764650	Tim	Jensen	Male	Analytics	160000	Bulgaria
10016	Anastasia	Edwards	Female	Analytics	159998	Kuwait
754528	Adrian	Young	Male	Game Analytics	159997	UK
240511	Diego	Lopez	Male	Game Analytics	159995	Malaysia
893472	Mariana	Cook	Female	People Analytics	159995	South Africa
359891	Mariana	Novak	Female	Game Analytics	159992	Mexico
53102	Felix	Taylor	Male	Data Science	159989	Bosnia
768143	Teresa	Campbell	Female	Game Analytics	159988	Spain
729165	Antonio	Weber	Male	Analytics	159987	Moldova
952549	Adrian	Harris	Male	Analytics	159986	Georgia

> Execution time: ~150–200 ms

Exercise 1 – E-commerce analytical queries

Dataset generation

Generated the e-commerce dataset:

```bash
cd ecommerce
python3 dataset_generator.py
```

File created:

- `orders_1M.csv`

Because only `data/` is mounted into the PostgreSQL container, the file was copied:

```bash
cp ecommerce/orders_1M.csv data/
```

Load e-commerce data

Created table:

```
DROP TABLE IF EXISTS ecommerce_orders;

CREATE TABLE ecommerce_orders (
  id SERIAL PRIMARY KEY,
  customer_name TEXT,
  product_category TEXT,
  quantity INTEGER,
  price_per_unit NUMERIC,
  order_date DATE,
  country TEXT
);
```

Loaded data:

```
\COPY
ecommerce_orders(customer_name,product_category,quantity,price_per_unit,or
der_date,country)
FROM '/data/orders_1M.csv' DELIMITER ',' CSV HEADER;
```

Verified:

```
SELECT COUNT(*) FROM ecommerce_orders;
```

Result:

- Rows loaded: 1,000,000

Question A – Highest price per unit

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

Result:

- Category: Automotive
- Price per unit: 2000.00
- Customer: Emma Brown
- Country: Italy

Question B – Top 3 products by total quantity sold

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

Result:

1. Health & Beauty – 300,842
2. Electronics – 300,804
3. Toys – 300,598

Question C – Total revenue per product category

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

Result:

- Automotive – 306,589,798.86
- Electronics – 241,525,009.45
- Books – 12,731,976.04

Question D – Customers with highest total spending

```
SELECT
  customer_name,
  SUM(price_per_unit * quantity) AS total_spent
FROM ecommerce_orders
GROUP BY customer_name
ORDER BY total_spent DESC
LIMIT 5;
```

Result:

- Carol Taylor – 991,179.18
- Nina Lopez – 975,444.95
- Daniel Jackson – 959,344.48
- Carol Lewis – 947,708.57
- Daniel Young – 946,030.14

Exercise 2 – Performance and scalability

Problematic query

```sql
SELECT COUNT(*)
FROM people_big p1
JOIN people_big p2
  ON p1.country = p2.country;
```

Problem explanation

- The table is joined with itself using `country`
- For each country with `n` rows, the join produces `n × n` row pairs
- This causes quadratic growth of the intermediate result
- PostgreSQL still has to process all pairs even though only `COUNT(*)` is needed
- The join adds no new information and introduces unnecessary computation

Optimized solution

The same result can be computed without a join by aggregating first:

```sql
SELECT SUM(cnt * cnt) AS total_pairs
FROM (
  SELECT country, COUNT(*) AS cnt
  FROM people_big
  GROUP BY country
) sub;
```

Why this works better

- Aggregation reduces the data size early
- No self-join is executed
- Much lower CPU and memory usage
- Better scalability in large and cloud-based systems

> An unnecessary self-join causes the performance issue. Using aggregation instead of a join solves the problem efficiently and scales better.

Exercise 3 – Analysis of Spark results

Data loading

- Spark loads the `people_big` table from PostgreSQL using JDBC
- Loading and materializing 1,000,000 rows takes ~1.8 seconds
- This overhead is expected due to data transfer from PostgreSQL to Spark

Query (a): Simple aggregation

- Computes average salary per department
- Execution time: ~1.9 seconds
- Spark performs a distributed group-by and aggregation
- Slightly slower than PostgreSQL due to Spark startup and scheduling overhead

Query (b): Nested aggregation

- Computes average salary per country using a nested aggregation
- Execution time: ~1.8 seconds
- Spark handles the two-level aggregation efficiently
- Performance remains stable despite increased query complexity

Query (c): Sorting + Top-N

- Sorts the full dataset by salary and returns the top 10 rows
- Execution time: ~2.5 seconds
- Sorting requires data shuffling across partitions
- Still performs well due to parallel execution

Query (d): Heavy self-join (dangerous)

- Performs a self-join on `country` and counts the result
- Execution time: ~7.0 seconds
- This is the slowest query by far
- The join produces a very large intermediate result
- Confirms the scalability issue discussed in Exercise 2

Query (d-safe): Join-equivalent rewrite

- Rewrites the self-join using aggregation and arithmetic
- Execution time: ~0.9 seconds
- Avoids the join completely
- Produces the same result with much lower cost

Key observations

- Spark handles large-scale aggregations and sorting efficiently
- Query design has a bigger impact on performance than the execution engine
- Expensive joins should be avoided when aggregation is sufficient
- The join-free rewrite clearly outperforms the self-join

> Spark is well suited for large analytical workloads and scales better than PostgreSQL for complex queries.
> However, poor query design (such as unnecessary self-joins) can still cause significant performance issues.
> Efficient query formulation is essential regardless of the processing engine.

Exercise 4 – Porting SQL queries to Spark

The e-commerce data was loaded from PostgreSQL into Spark using JDBC and registered as a temporary view.

All queries from Exercise 1 (A–D) were reimplemented using Spark SQL:

- highest price per unit
- top products by quantity sold
- total revenue per category
- top customers by total spending

> The results produced by Spark match the PostgreSQL results exactly. This confirms the correctness of the Spark implementations.