Part 1 — Environment Setup and Basics

3. Load and query data in PostgreSQL

3.1 Create a large dataset

```
cd data
python3 expand.py
```

Creates `data/people_1M.csv` with ~1 million rows.

3.2 Enter PostgreSQL

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

3.3 Create and load the table

```
DROP TABLE IF EXISTS people_big;

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;
```

3.4 Enable timing

```
\timing on
```

4. Verification

```
SELECT COUNT(*) FROM people_big;
SELECT * FROM people_big LIMIT 10;
```

This gives the following output for me:

```
  count
---------
 1000000
(1 row)

Time: 46.107 ms
 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
  6 | Oliver     | Baker     | Male   | Payroll            |  74110 | Poland
  7 | Emily      | Lopez     | Female | SOC                |  83526 | Netherlands
  8 | Tania      | King      | Female | IT                 |  95142 | Thailand
  9 | Max        | Hernandez | Male   | Workforce Planning | 101198 | Latvia
 10 | Juliana    | Harris    | Female | Compliance         | 103336 | Chile
(10 rows)

Time: 0.278 ms
```

5. Analytical queries

(a) Simple aggregation

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

my output:

```
      department       |         avg
-----------------------+---------------------
 Accounting            |  85150.560834888851
 Alliances             |  84864.832756437315
 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
(10 rows)

Time: 299.784 ms
```

(b) Nested aggregation

```
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;
```

my output:

```
  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
(10 rows)

Time: 467.721 ms
```

(c) Top-N sort

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

my output:

```
   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
 893472 | Mariana      | Cook       | Female | People Analytics  | 159995 | South
Africa
 240511 | Diego        | Lopez      | Male   | Game Analytics    | 159995 | Malaysia
 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
(10 rows)

Time: 68.510 ms
```

Part 2 — Exercises

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**.

my Schema for the new table Orders:

```sql
DROP TABLE IF EXISTS orders;

CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  customer_name TEXT,
  product_category TEXT,
  quantity INTEGER,
  price_per_unit NUMERIC(10,2),
  order_date DATE,
  country TEXT
);

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

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`?

Query:

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

Output:

```
   id   | customer_name | product_category | quantity | price_per_unit |
order_date | country
--------+---------------+------------------+----------+----------------+----------
--+----------
 841292 | Emma Brown    | Automotive       |        3 |        2000.00 | 2024-10-
11 | Italy
(1 row)
```

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

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

Output:

```
 product_category | total_quantity
------------------+----------------
 Health & Beauty  |         300842
 Electronics      |         300804
 Toys             |         300598
(3 rows)
```

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

Query:

```
SELECT product_category, SUM(quantity * price_per_unit) AS revenue
FROM orders
GROUP BY product_category;
```

Output:

```
 product_category |   revenue
------------------+--------------
 Automotive       | 306589798.86
 Books            |  12731976.04
 Electronics      | 241525009.45
 Fashion          |  31566368.22
 Grocery          |  15268355.66
 Health & Beauty  |  46599817.89
 Home & Garden    |  78023780.09
 Office Supplies  |  38276061.64
 Sports           |  61848990.83
 Toys             |  23271039.02
(10 rows)
```

D. Which customers have the highest total spending?

Query:

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

Output:

```
 customer_name  | total_spending
----------------+----------------
 Carol Taylor   |      991179.18
 Nina Lopez     |      975444.95
 Daniel Jackson |      959344.48
 Carol Lewis    |      947708.57
 Daniel Young   |      946030.14
 Alice Martinez |      935100.02
 Ethan Perez    |      934841.24
 Leo Lee        |      934796.48
 Eve Young      |      933176.86
 Ivy Rodriguez  |      925742.64
(10 rows)
```

Exercise 2

Assuming there are naive joins executed by users, such as:

```sql
SELECT COUNT(*)
FROM people_big p1
```

```
JOIN people_big p2
  ON p1.country = p2.country;
```

Problem Statement

This query takes more than **10 minutes** to complete, significantly slowing down the entire system. Additionally, the **OLTP database** currently in use has inherent limitations in terms of **scalability and efficiency**, especially when operating in **large-scale cloud environments**.

Discussion Question

Considering the requirements for **scalability** and **efficiency**, what **approaches and/or optimizations** can be applied to improve the system's:

- Scalability
- Performance
- Overall efficiency

Please **elaborate with a technical discussion**.

Different approaches and optimizations:

- Create a (text) index on the column "country". This helps increasing the join build speed, but the query is still CPU- and memory-heavy. E.g.:

```
CREATE INDEX idx_people_big_country ON people_big(country);
```

- simple query-level optimizations, the following query leads to the same result without joining the table with itself:

```
-- multiply the count with itself to get the total number of pairs within a country
SELECT SUM(count * count)
FROM (
  -- count the number of people per country
  SELECT country, COUNT(*) AS count
  FROM people_big
  GROUP BY country
) t;
```

- clearly separating OLTP and OLAP, and moving these kind of analytical queries to OLAP, where it can be executed more efficiently

Exercise 3

Run with Spark (inside Jupyter)

Open your **Jupyter Notebook** environment:

- **URL:** http://localhost:8888/?token=lab
- **Action:** Create a new notebook

Then run the **updated Spark example**, which uses the same data stored in **PostgreSQL**.

Analysis and Discussion

Now, explain in your own words:

- **What the Spark code does:**
  Describe the workflow, data loading, and the types of queries executed (aggregations, sorting, self-joins, etc.).

0. Imports & Spark Session

All libraries are imported and a SparkSession is created. Some configurations like the log level or parallelism are set.

1. JDBC connection config

Here one can find the configuration settings to connect to the PostgreSQL database. Properties like the username and password are set.

2. Load data from PostgreSQL

With `spark.read.jdbc` all the contents from the table *people_big* are read into the variable *df_big*. Because Python prefers lazy execution, no data is actually read until you use *df_big* the first time, `df_big.count()` is called to force it to read and ensure the load time is correct.

3. Query(a): Simple aggregation

This first query returns the average salaries per department of the first 10 departments. This is done by aggregating the variable *df_big* that holds all the data. First, it is grouped by Department and then the rounded average salary with 2 decimals is computed per department. The result is ordered by Department descendingly and it is limited to the first 10 rows.

With `q_a.collect()` the data is retrieved and with `q_a.show` it is displayed. `truncate=False` prevents it from not printing the whole output.

4. Query(b): Nested aggregation

This query showcases that with spark we can also write SQL-Queries and execute them, it is not needed to always use the Spark DataFrame API.

This query is a nested aggregation, first we compute the average salary per country and department, then the average of this average salary is calculated, grouped by the country. Only the 10 highest average salaries are returned.

5. Query(c): Sorting + Top-N

This is a simple Sorting-Query, where only the top 10 rows with the highest salary are shown.

6. Query(d): Heavy self-join (COUNT only)

This query showcases the one in Exercise 2. It executes a self-join on the column "country".

7. Query(d-safe): Join-equivalent rewrite

The result of this query should be the same as the one previous. First, we group by country and count the rows. Then, we multiply the count with itself to get the total pairs.

- **Architectural contrasts with PostgreSQL:**
  Compare the Spark distributed architecture versus PostgreSQL’s single-node capabilities, including scalability, parallelism, and data processing models.

Spark:

Master-Worker architecture, where the Driver acts as master and the Executors as workers. The Driver is the brain of Spark, it tells the cluster what to do. The Executors execute the tasks, keep data in memory and report back to the driver. Spark decides where and how to run tasks in parallel on the Executors.

This driver-worker separation also improves fault isolation. If a worker node dies while executing a task, Spark can reassign that task to another worker without crashing your application.

When you write PySpark code, it is not run immediatley. You're building a plan, and Spark's Catalyst Optimizer takes that plan and transforms it into an efficient execution strategy.

That works in 4 phases: The 1. step contains resolving the column names, data types. Then, Spark applies rules to optimize filters and combine projections. In the third phase, Spark considers multiple execution startegies and picks the most efficient one, based on data size, partitioning, etc. Finally, it generates the code.

Once this plan is finished, the DAG scheduler takes over and breaks down the job into stages based on shuffle boundaries, where Spark decides what happens sequentially and what can be executed in parallel.

PostgreSQL:

Client/Server model, one PostgreSQL session consists of a server process and the user's client (frontend) application. The server process manages the database files, handles database connections, and performs database actions on behalf of the clients. The client wants to perform database operations. Client applications can be very diverse: from text-oriented tools over graphical applications to a specialized database maintenance tool.

Here the first difference to Spark is noticeable, as PostgreSQL relies on one server process for execution per client, whereas Spark has this driver-worker approach with multiple processes per client.

In terms of scaling, the next difference is apparent: Spark can be horizontally scaled by adding executors, whereas PostgreSQL only vertically by increasing the power of the machine. This leads to PostgreSQL being staticly scaleable, where a restart is often required. Spark can be scaled dynamically.

In a later PostgreSQL version, parallelism was introduced, but compared to Spark it still parallelizes within one machine, where Spark parallelizes across machines.

PostgreSQL executes the queries immediatly, while Spark is using lazy evaluation and execution.

- **Advantages and limitations:**
  Highlight the benefits of using Spark for large-scale data processing (e.g., in-memory computation, distributed processing) and its potential drawbacks (e.g., setup complexity, overhead for small datasets).

Spark is very suitable for large-scale data because of its distributed processing and in-memory computation. It is also Open-source, provides different APIs and supports many languages.

For small datasets there is probably too much overhead, other architectures with less setup complexity would probably achieve better results than Spark. Additionally, ACID support would require additional layers, it follows the concept of "eventually consistent".

- **Relation to Exercise 2:**
  Connect this approach to the concepts explored in Exercise 2, such as performance optimization and scalability considerations.

Just rewriting the query would lead to the best optimization, but Spark would help in Exercise 2 to improve the performance. It can partition the table by country and compute the aggregates per partition.

sources for exercise 3:

https://www.datacamp.com/blog/apache-spark-architecture

https://www.postgresql.org/docs/current/tutorial-arch.html

https://www.percona.com/blog/parallelism-in-postgresql/

Exercise 4

Port the SQL queries from exercise 1 to spark.

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

SQL-Query:

```sql
SELECT *
FROM orders
ORDER BY price_per_unit DESC
LIMIT 1;
```

Spark:

```python
q_a = (
    df_orders
    .orderBy("price_per_unit", ascending=False)
    .limit(1)
)
```

Output:

```
=== Query (a): What is the single item with the highest `price_per_unit`? ===
+------+-------------+----------------+--------+-------------+----------+-------+
|id    |customer_name|product_category|quantity|price_per_unit|order_date|country|
+------+-------------+----------------+--------+-------------+----------+-------+
|841292|Emma Brown   |Automotive      |3       |2000.00       |2024-10-11|Italy  |
+------+-------------+----------------+--------+-------------+----------+-------+

Query (a) time: 10.03 seconds
```

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

SQL-Query:

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

Spark:

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

Output:

```
=== Query (b): What are the top 3 products category with the highest total
quantity sold across all orders? ===
+----------------+--------------+
|product_category|total_quantity|
+----------------+--------------+
|Health & Beauty |300842        |
|Electronics     |300804        |
|Toys            |300598        |
+----------------+--------------+

Query (b) time: 2.78 seconds
```

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

SQL-Query:

```
SELECT product_category, SUM(quantity * price_per_unit) AS revenue
FROM orders
GROUP BY product_category;
```

Spark:

```
q_c = (
    df_orders
    .groupBy("product_category")
    .agg(_sum(col("quantity")*col("price_per_unit")).alias("revenue"))
    .orderBy("revenue", ascending=False)
)
```

Output:

```
=== Query (c): What is the total revenue per product category? ===
+----------------+------------+
|product_category|revenue     |
+----------------+------------+
|Automotive      |306589798.86|
|Electronics     |241525009.45|
|Home & Garden   |78023780.09 |
|Sports          |61848990.83 |
|Health & Beauty |46599817.89 |
|Office Supplies |38276061.64 |
|Fashion         |31566368.22 |
|Toys            |23271039.02 |
|Grocery         |15268355.66 |
|Books           |12731976.04 |
+----------------+------------+

Query (c) time: 3.41 seconds
```

D. Which customers have the highest total spending?

SQL-Query:

```
SELECT customer_name, SUM(quantity * price_per_unit) AS total_spending
FROM orders
GROUP BY customer_name
```

```
ORDER BY total_spending DESC
LIMIT 10;
```

Spark:

```
q_d = (
    df_orders
    .groupBy("customer_name")
    .agg(_sum(col("quantity")*col("price_per_unit")).alias("total_spending"))
    .orderBy("total_spending", ascending=False)
    .limit(10)
)
```

Output:

```
=== Query (d): Which customers have the highest total spending? ===
+--------------+--------------+
|customer_name |total_spending|
+--------------+--------------+
|Carol Taylor  |991179.18     |
|Nina Lopez    |975444.95     |
|Daniel Jackson|959344.48     |
|Carol Lewis   |947708.57     |
|Daniel Young  |946030.14     |
|Alice Martinez|935100.02     |
|Ethan Perez   |934841.24     |
|Leo Lee       |934796.48     |
|Eve Young     |933176.86     |
|Ivy Rodriguez |925742.64     |
+--------------+--------------+

Query (d) time: 3.9 seconds
```

Clean up

```
docker compose down
```