Part 1 — Environment Setup and Basics

1. Start the environment

Download the repository and start the environment:

```
docker compose up -d
```

2. Access PostgreSQL

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

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.

```
wc -l people_1M.csv
```

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

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

5. Analytical queries

(a) Simple aggregation

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

(b) Nested aggregation

```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;
```

(c) Top-N sort

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

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**.

Generate the dataset and copy it to the mounted folder:

```
cd ecommerce
python3 dataset_generator.py
cp .\orders_1M.csv ..\data\
```

Go in postgres:

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

Load the data:

```
DROP TABLE IF EXISTS orders_big;

CREATE TABLE orders_big (
  id SERIAL PRIMARY KEY,
  customer_name TEXT NOT NULL,
  product_category TEXT NOT NULL,
  quantity BIGINT NOT NULL,
  price_per_unit DOUBLE PRECISION NOT NULL,
  order_date DATE NOT NULL,
  country TEXT NOT NULL
);

\COPY
orders_big(customer_name,product_category,quantity,price_per_unit,order_date,count
ry)
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`?

```
SELECT customer_name, product_category, price_per_unit
FROM orders_big
```

```sql
ORDER BY price_per_unit DESC
LIMIT 1;

--- Result: ---
--  customer_name | product_category | price_per_unit
-- ---------------+------------------+----------------
--  Emma Brown    | Automotive       |           2000
```

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

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

--- Result: ---
--  product_category | total_quantity
-- ------------------+----------------
--  Health & Beauty  |         300842
--  Electronics      |         300804
--  Toys             |         300598
```

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

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

--- Result: ---
--  product_category |   total_revenue
-- ------------------+--------------------
--  Automotive       |  306589798.8600007
--  Electronics      |  241525009.4500003
--  Home & Garden    |  78023780.09000018
--  Sports           |  61848990.82999988
--  Health & Beauty  | 46599817.890000105
--  Office Supplies  |  38276061.63999985
--  Fashion          | 31566368.220000084
--  Toys             | 23271039.020000055
--  Grocery          | 15268355.660000065
--  Books            | 12731976.040000042
```

D. Which customers have the highest total spending?

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

--- Result: ---
--    customer_name  |    total_spent
-- ------------------+------------------
--   Carol Taylor     | 991179.1799999999
--   Nina Lopez       |         975444.95
--   Daniel Jackson   |         959344.48
--   Carol Lewis      | 947708.5699999997
--   Daniel Young     | 946030.1400000004
--   Alice Martinez   | 935100.0200000001
--   Ethan Perez      |         934841.24
--   Leo Lee          | 934796.4799999995
--   Eve Young        | 933176.8600000003
--   Ivy Rodriguez    | 925742.6400000004
```

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:

Problem description:

- It is a self-join on a low-cardinality column:
  - If a country has n rows, the join produces n^2 rows
  - Total output ≈ $\Sigma(n_i^2)$ across all countries
- Materializing such a huge view requires a large amount of resources
- Example:
  - 1M rows

  - 200 countries
  - Average 5,000 rows per country
  - This leads to 5,000 × 5,000 × 200 = 5,000,000,000 join matches

How to handle:

- Scalability
  - Avoid full self-joins on low-cardinality columns
  - Offload heavy analytical queries to distributed systems (e.g. Spark)
  - Use pre-aggregated or materialized views to reduce repeated heavy computations
- Performance
  - Rewrite queries mathematically instead of materializing joins
  - Maintain indexes on join keys to speed up grouping or filtering
  - Cache intermediate results for frequently executed queries
- Overall Efficiency
  - Minimize resource usage by avoiding billion-row materializations
  - For very large datasets, consider approximate counting techniques
  - Keep OLTP workload isolated from heavy analytical queries to prevent system slowdowns

Please **elaborate with a technical discussion**.

> **Optional:** Demonstrate your proposed solution in practice (e.g., architecture diagrams, SQL examples, or code snippets).

Solution: Instead of performing an actual join, compute the count of pairs per country mathematically:

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

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

Spark Example Code

```
# ============================================
# 0. Imports & Spark session
# ============================================

import time
import builtins  # <-- IMPORTANT
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    avg,
    round as spark_round,   # Spark round ONLY for Columns
    count,
    col,
    sum as _sum
)

spark = (
    SparkSession.builder
    .appName("PostgresVsSparkBenchmark")
    .config("spark.jars.packages", "org.postgresql:postgresql:42.7.2")
    .config("spark.eventLog.enabled", "true")
    .config("spark.eventLog.dir", "/tmp/spark-events")
    .config("spark.history.fs.logDirectory", "/tmp/spark-events")
    .config("spark.sql.shuffle.partitions", "4")
    .config("spark.default.parallelism", "4")
    .getOrCreate()
)

spark.sparkContext.setLogLevel("WARN")

# ============================================
# 1. JDBC connection config
# ============================================

jdbc_url = "jdbc:postgresql://postgres:5432/postgres"
jdbc_props = {
    "user": "postgres",
    "password": "postgres",
    "driver": "org.postgresql.Driver"
}

# ============================================
# 2. Load data from PostgreSQL
# ============================================

print("\n=== Loading people_big from PostgreSQL ===")

start = time.time()

df_big = spark.read.jdbc(
    url=jdbc_url,
    table="people_big",
    properties=jdbc_props
)
```

```
# Force materialization
row_count = df_big.count()

print(f"Rows loaded: {row_count}")
print("Load time:", builtins.round(time.time() - start, 2), "seconds")

# Register temp view
df_big.createOrReplaceTempView("people_big")

# ============================================
# 3. Query (a): Simple aggregation
# ============================================

print("\n=== Query (a): AVG salary per department ===")

start = time.time()

q_a = (
    df_big
    .groupBy("department")
    .agg(spark_round(avg("salary"), 2).alias("avg_salary"))
    .orderBy("department", ascending=False)
    .limit(10)
)

q_a.collect()
q_a.show(truncate=False)
print("Query (a) time:", builtins.round(time.time() - start, 2), "seconds")

# ============================================
# 4. Query (b): Nested aggregation
# ============================================

print("\n=== Query (b): Nested aggregation ===")

start = time.time()

q_b = spark.sql("""
SELECT country, AVG(avg_salary) AS avg_salary
FROM (
    SELECT country, department, AVG(salary) AS avg_salary
    FROM people_big
    GROUP BY country, department
) sub
GROUP BY country
ORDER BY avg_salary DESC
LIMIT 10
""")

q_b.collect()
q_b.show(truncate=False)
print("Query (b) time:", builtins.round(time.time() - start, 2), "seconds")
```

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

print("\n=== Query (c): Top 10 salaries ===")

start = time.time()

q_c = (
    df_big
    .orderBy(col("salary").desc())
    .limit(10)
)

q_c.collect()
q_c.show(truncate=False)
print("Query (c) time:", builtins.round(time.time() - start, 2), "seconds")

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

print("\n=== Query (d): Heavy self-join COUNT (DANGEROUS) ===")

start = time.time()

q_d = (
    df_big.alias("p1")
    .join(df_big.alias("p2"), on="country")
    .count()
)

print("Join count:", q_d)
print("Query (d) time:", builtins.round(time.time() - start, 2), "seconds")

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

print("\n=== Query (d-safe): Join-equivalent rewrite ===")

start = time.time()

grouped = df_big.groupBy("country").agg(count("*").alias("cnt"))

q_d_safe = grouped.select(
    _sum(col("cnt") * col("cnt")).alias("total_pairs")
)

q_d_safe.collect()
q_d_safe.show()
print("Query (d-safe) time:", builtins.round(time.time() - start, 2), "seconds")

# =============================================
```

```
# 8. Cleanup
# ============================================

spark.stop()
```

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. Setup & import: SparkSession is created with PostgreSQL JDBC driver support
  1. JDBC config: Defines connection properties for PostgreSQL
  2. Load data:
     - Data is loaded via spark.read.jdbc(...) into a Spark DataFrame df_big
     - .count() is used to force materialization (force the loading of the data)
  3. Query (a): Computes average salary per department
  4. Query (b): Calculates average of department averages per country
  5. Query (c): Returns top 10 salaries across the dataset (using order by and limit to only load top 10)
  6. Query (d): Performs a self-join on country, counting all resulting rows
  7. Query (d-safe): Instead of a full self-join, it computes total pairs per country mathematically
  8. Cleanup: stop the spark session

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

  - Architecture
    - Spark: Distributed master-worker cluster
    - PostgreSQL: Single-node by default
  - Data processing model
    - Spark: In-memory, lazy evaluation
    - PostgreSQL: Disk-based, immediate execution
  - Parallelism
    - Spark: Automatic parallelism across partitions, multiple tasks run on different nodes concurrently
    - PostgreSQL: Limited parallelism, mostly single-threaded per query with some multi-threaded support
  - Scalability
    - Spark: Handles very large datasets by distributing workload across cluster
    - PostgreSQL: Efficient for small-to-medium datasets (large-scale data constrained by single-node resources)
  - Fault tolerance
    - Spark: Resilient Distributed Datasets and automatic recovery on node failure
    - PostgreSQL: Transaction logs (WAL) and replication

- **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).
    - Advantages
        - Distributed processing
        - In-memory computation (significantly faster for iterative jobs)
        - Flexible APIs (supports SQL, DataFrame and RDD APIs)
        - Fault tolerance
        - Scalable aggregations and joins
    - Limitations
        - Overhead for small datasets (spark startup and scheduling costs may exceed PostgreSQL execution time for small tables)
        - Setup complexity (requires cluster management or Spark standalone installation)
        - Resource intensive (needs proper tuning to achieve performance)
        - Potentially high shuffle costs (large joins or wide aggregations can trigger expensive network I/O)
- **Relation to Exercise 2:**
  Connect this approach to the concepts explored in Exercise 2, such as performance optimization and scalability considerations.
    - The code and Spark in general address many solutions discusses in exercise 2 like:
        - Offload heavy analytics (e.g., self-joins) from the OLTP DB to Spark's distributed engine
        - Parallelize scans/aggregations to avoid single-node bottlenecks seen in the naive join
        - Use math-based rewrites (e.g., SUM(cnt*cnt)) to avoid explosive low-cardinality self-joins
        - Measure each step with timing to surface hotspots and guide optimizations

Exercise 4

Port the SQL queries from exercise 1 to spark.

```
# ============================================
# 0. Imports & Spark session
# ============================================

import time
import builtins  # <-- IMPORTANT
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    avg,
    round as spark_round,   # Spark round ONLY for Columns
    count,
    col,
    sum as _sum
)

spark = (
    SparkSession.builder
    .appName("PostgresVsSparkBenchmark")
```

```
    .config("spark.jars.packages", "org.postgresql:postgresql:42.7.2")
    .config("spark.eventLog.enabled", "true")
    .config("spark.eventLog.dir", "/tmp/spark-events")
    .config("spark.history.fs.logDirectory", "/tmp/spark-events")
    .config("spark.sql.shuffle.partitions", "4")
    .config("spark.default.parallelism", "4")
    .getOrCreate()
)

spark.sparkContext.setLogLevel("WARN")

# ============================================
# 1. JDBC connection config
# ============================================

jdbc_url = "jdbc:postgresql://postgres:5432/postgres"
jdbc_props = {
    "user": "postgres",
    "password": "postgres",
    "driver": "org.postgresql.Driver"
}

# ============================================
# 2. Load data from PostgreSQL
# ============================================

print("\n=== Loading orders_big from PostgreSQL ===")

start = time.time()

df_big = spark.read.jdbc(
    url=jdbc_url,
    table="orders_big",
    properties=jdbc_props
)

# Force materialization
row_count = df_big.count()

print(f"Rows loaded: {row_count}")
print("Load time:", builtins.round(time.time() - start, 2), "seconds")

# Register temp view
df_big.createOrReplaceTempView("orders_big")

# ============================================
# 3. A: Single item with highest price_per_unit
# ============================================

print("\n=== Query (A): Max price_per_unit ===")

start = time.time()

q_a = (
```

```
    df_big.select("customer_name", "product_category", "price_per_unit")
          .orderBy(col("price_per_unit").desc())
          .limit(1)
)

q_a.collect()
q_a.show(truncate=False)
print("Query (A) time:", builtins.round(time.time() - start, 2), "seconds")

# ============================================
# 4. B: Top 3 product categories by total quantity
# ============================================

print("\n=== Query (B): Top 3 categories by quantity ===")

start = time.time()

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

q_b.collect()
q_b.show(truncate=False)
print("Query (B) time:", builtins.round(time.time() - start, 2), "seconds")

# ============================================
# 5. C: Total revenue per product category
# ============================================

print("\n=== Query (C): Total revenue per category ===")

start = time.time()

q_c = (
    df_big.groupBy("product_category")
          .agg(_sum(col("price_per_unit") *
col("quantity")).alias("total_revenue"))
          .orderBy(col("total_revenue").desc())
)

q_c.collect()
q_c.show(truncate=False)
print("Query (C) time:", builtins.round(time.time() - start, 2), "seconds")

# ============================================
# 6. D: Customers with highest total spending
# ============================================

print("\n=== Query (D): Top customers by total spending ===")

start = time.time()
```

```
q_d = (
    df_big.groupBy("customer_name")
          .agg(_sum(col("price_per_unit") * col("quantity")).alias("total_spent"))
          .orderBy(col("total_spent").desc())
          .limit(10)
)

q_d.collect()
q_d.show(truncate=False)
print("Query (D) time:", builtins.round(time.time() - start, 2), "seconds")

# ============================================
# 7. Cleanup
# ============================================

spark.stop()
```

Results:

```
=== Loading orders_big from PostgreSQL ===
Rows loaded: 1000000
Load time: 1.57 seconds

=== Query (A): Max price_per_unit ===
+-------------+----------------+--------------+
|customer_name|product_category|price_per_unit|
+-------------+----------------+--------------+
|Emma Brown   |Automotive      |2000.0        |
+-------------+----------------+--------------+

Query (A) time: 2.67 seconds

=== Query (B): Top 3 categories by quantity ===
+----------------+--------------+
|product_category|total_quantity|
+----------------+--------------+
|Health & Beauty |300842        |
|Electronics     |300804        |
|Toys            |300598        |
+----------------+--------------+

Query (B) time: 1.54 seconds

=== Query (C): Total revenue per category ===
+----------------+--------------------+
|product_category|total_revenue       |
+----------------+--------------------+
|Automotive      |3.065897988599943E8 |
|Electronics     |2.4152500945000267E8|
|Home & Garden   |7.80237800900001E7  |
|Sports          |6.1848990830000326E7|
```

```
|Health & Beauty |4.65998178900003E7  |
|Office Supplies |3.8276061640000574E7|
|Fashion         |3.1566368219999947E7|
|Toys            |2.3271039019999716E7|
|Grocery         |1.5268355660000028E7|
|Books           |1.273197603999989E7 |
+----------------+--------------------+

Query (C) time: 2.3 seconds

=== Query (D): Top customers by total spending ===
+--------------+----------------+
|customer_name |total_spent     |
+--------------+----------------+
|Carol Taylor  |991179.1800000003|
|Nina Lopez    |975444.9499999998|
|Daniel Jackson|959344.4800000001|
|Carol Lewis   |947708.5700000002|
|Daniel Young  |946030.1400000004|
|Alice Martinez|935100.0199999999|
|Ethan Perez   |934841.2399999991|
|Leo Lee       |934796.4799999993|
|Eve Young     |933176.8599999989|
|Ivy Rodriguez |925742.6400000005|
+--------------+----------------+

Query (D) time: 2.09 seconds
```

Clean up

```
docker compose down
```