– DOCUMENTATION / STEPS and APPROACHES

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
```

OUTPUT: "1000001 ./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;
```

OUTPUT:

```
NOTICE:  table "people_big" does not exist, skipping
DROP TABLE
CREATE TABLE
COPY 1000000
```

3.4 Enable timing

```
\timing on
```

OUTPUT: Timing is on.

4. Verification

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

OUTPUT:

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

Time: 20.863 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: 0.217 ms
```

5. Analytical queries

(a) Simple aggregation

```
SELECT department, AVG(salary)
FROM people_big
```

```
GROUP BY department
LIMIT 10;
```

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: 81.128 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;
```

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: 135.756 ms '''

(c) Top-N sort

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

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: 31.839 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**.

TODO:

```
cd .\ecommerce\
python3 .\dataset_generator.py
```

```
$ wc -l ./orders_1M.csv
```

OUTPUT: 1000001 ./orders_1M.csv

- create db:

```
DROP TABLE IF EXISTS orders_big;

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

\COPY orders_big(customer_name,product_category,quantity,price_per_unit,order_date,country)
FROM '/ecommerce/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
  product_category,
```

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

ANSWER: The price is 2000 and the product-Category is Automotive

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

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

Answer:

```
 product_category | total_quantity_sold
------------------+---------------------
 Health & Beauty  |              300842
 Electronics      |              300804
 Toys             |              300598
```

They are Health & Beauty, Electronics and Toys.

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

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

ANSWER:

```
 product_category | total_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
(10 rows)
```

D. Which customers have the highest total spending?

```
SELECT
  customer_name,
  ROUND(SUM(price_per_unit * quantity), 2) AS total_spent
FROM orders_big
GROUP BY customer_name
ORDER BY total_spent DESC
LIMIT 10;
```

ANSWER: The top 10 customers which have the highest total spendigs are:

```
 customer_name  | total_spent
----------------+-------------
 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:

```
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**.

ANSWER: Why the query is slow

- The query performs a self-join on a low-cardinality column (country), causing a row explosion (N × N) per country.
- It requires full table scans and creates massive intermediate results.
- OLTP databases are optimized for small transactions, not large analytical joins!!!

HOW to fix it: - rewriting the sql by avoiding the join by aggregating first (reduces from $O(N^2)$ to just $O(N)$)

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

- or use columnar storage:
  Columnar databases scan only required columns, compress well, and execute aggregations much faster.
- or Separate OLTP and OLAP workloads
  - Run analytics on read replicas or data warehouses, not the OLTP system
  - Examples: BigQuery, Redshift, Snowflake, ClickHouse

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

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()
```

TODO - What I Did: - Just opened the link and created a new Notebook in the work folder which is the notebooks folder locally.
- I named the file: "Ex3_Spark_Schlosser.ipynb"
- then i just rewrite the code form the readme in to the cells - 1 point is one cell

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.).
  The code is seperated in 8 sections:
  - 0: just imports and set up spark session

- 1: sets up db conn details using JDBC
- 2: Data loading via JDBC: Reads people_big from PostgreSQL into a Spark DataFrame
  Forces materialization using .count() to measure load time
  Registers the DataFrame as a temporary Spark SQL view
- 3: Query a -> Simple: Average salary per department
- 4: Query b -> Nested agg: Average department salary per country
- 5: Query c -> Sorting + Top-N: Highest salaries
- 6: Query d -> Heavy self-join: Self-join on country to demonstrate a worst-case analytical join
  will do a count of a join on p1 and p2 with the contry
- 7: Query d (save) -> Join-equivalent rewrite: the same rewrition i mentioned above
- 8: stop the spark session

All of the qurries print the time needed and the last one writes the joins used as well

To compare the Query d with the same amount of joins:

- wrong one takes 5.8 sec
- rewritten one: 0.8 sec

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

  Spark

  - Distributed, shared-nothing architecture
  - Data is split into partitions and processed in parallel
  - Optimized for:
    * Large scans
    * Aggregations
    * Joins across big datasets
  - Uses in-memory execution and fault-tolerant DAGs

  PostgreSQL (OLTP)

  - Primarily single-node execution
  - Optimized for:
    * Transactions
    * Index lookups
    * Short queries
  - Limited parallelism for:
    * Large joins
    * Full table analytics

  Key difference:

  - Spark scales horizontally, PostgreSQL scales mainly vertically.

- **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
    * Massive parallelism for large datasets
    * Efficient aggregations and joins at scale
    * In-memory processing reduces disk I/O
    * Handles analytical workloads without impacting OLTP systems
  - Limitations
    * Higher setup and operational complexity
    * JDBC ingestion adds overhead
    * Less efficient for:
      · Small datasets
      · Low-latency point queries
    * Requires careful tuning (partitions, memory, shuffles)

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

  Exercise 2 highlighted how naive analytical queries (e.g., self-joins on large tables) can severely degrade OLTP systems.

  This Spark example shows:

  - Why such workloads should not run on OLTP databases
  - How distributed execution mitigates performance issues
  - The importance of query rewrites (aggregation vs join)
  - The benefit of separating OLTP and OLAP workloads

  So spark provides a scalable, efficient alternative for large-scale analytical queries that would otherwise overwhelm a traditional OLTP database like PostgreSQL.

  This directly reinforces the scalability, performance, and efficiency principles discussed in Exercise 2.

Exercise 4

Port the SQL queries from exercise 1 to spark.

I made a new Notebook - so I have to do the hole setup part again:

```
#SETUP
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")


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


print("\n=== Loading orders from PostgreSQL ===")
start_time = time.time()

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

row_count = orders_df.count()
print(f"Rows loaded: {row_count}")
print("Load time:", round(time.time() - start_time, 2), "seconds")

# Register temp view for Spark SQL
```

```
orders_df.createOrReplaceTempView("orders")
```

First Querry:

```
print("\n=== Query (A): Highest price_per_unit item ===")
start = time.time()

q_a = (
    orders_df
    .select("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")
```

Output:

```
=== Query (A): Highest price_per_unit item ===
+----------------+----------------------+
|product_category|price_per_unit        |
+----------------+----------------------+
|Automotive      |2000.0000000000000000|
+----------------+----------------------+

Query (A) time: 1.54 seconds
```

Second Querry:

```
print("\n=== Query (B): Top 3 product categories by total quantity sold ===")
start = time.time()

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

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

Output:

```
=== Query (B): Top 3 product categories by total quantity sold ===
+----------------+-------------------+
```

```
|product_category|total_quantity_sold|
+----------------+-------------------+
|Health & Beauty |300842             |
|Electronics     |300804             |
|Toys            |300598             |
+----------------+-------------------+

Query (B) time: 1.19 seconds
```

Third Querry:

```
print("\n=== Query (C): Total revenue per product category ===")
start = time.time()

q_c = (
    orders_df
    .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")
```

Output:

```
=== Query (C): Total revenue per product category ===
+----------------+-----------------+
|product_category|total_revenue    |
+----------------+-----------------+
|Automotive      |306589798.8600000|
|Electronics     |241525009.4500000|
|Home & Garden   |78023780.0900000 |
|Sports          |61848990.8300000 |
|Health & Beauty |46599817.8900000 |
|Office Supplies |38276061.6400000 |
|Fashion         |31566368.2200000 |
|Toys            |23271039.0200000 |
|Grocery         |15268355.6600000 |
|Books           |12731976.0400000 |
+----------------+-----------------+

Query (C) time: 2.76 seconds
```

First Querry:

```
print("\n=== Query (D): Top 10 customers by total spending ===")
start = time.time()
```

```
q_d = (
    orders_df
    .groupBy("customer_name")
    .agg(
        spark_round(_sum(col("price_per_unit") * col("quantity")), 2).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")
```

Output:

```
=== Query (D): Top 10 customers by total spending ===
+--------------+-----------+
|customer_name |total_spent|
+--------------+-----------+
|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: 2.63 seconds
```

Clean up

```
docker compose down
```