Exercise2

February 3, 2025

1 Exercise 2: Data Warehousing and Data Lakes with Spark + Hive

1.1 Introduction

In modern data engineering, we often encounter two primary paradigms: 1. **Data Warehouse** (schema-on-write): where data is cleansed, transformed, and loaded into structured tables before analysis. 2. **Data Lake** (schema-on-read): where data is stored in raw format and the schema is applied when querying.

This exercise showcases both approaches using **Spark** and **Hive**. You will load and query **e-commerce data** in a structured (warehouse) format, then contrast this with a more flexible (lake) approach. By the end, you should understand key **ETL/ELT** concepts, the rationale behind each paradigm, and be able to discuss the differences.

Useful links and notebooks: - https://spark.apache.org/docs/latest/api/python/index.html - https://spark.apache.org/docs/3.5.1/sql-data-sources-hive-tables.html - /shared/ETL_ELT

1.2 Objectives

This exercise is worth 18 points. To earn full points, make sure to include comments in your code explaining your approach and the reasoning behind your choices.

1. **Data Warehouse Fundamentals (6p)**:
    - Define and create schemas using Apache Hive.
    - Perform schema-on-write transformations and run analytical queries.

3. **Questions (6p)**:
    - Answer three questions about the ETL and ELT.

2 E-Commerce Data Schema

We will be working with a couple of datasets from an e-commerce site located in the /shared folder.

2.1 1. `customers.csv`

- **Description**: Contains information about customers.
- **Fields**:
    - `customer_id` (int): Unique identifier for each customer.

- name (string): Customer's full name.
- age (int): Age of the customer.
- country (string): Country of residence.
- preferred_category (string): Preferred product category (e.g., Electronics, Books).
- loyalty_score (float): Loyalty score between 0.00 and 1.00.

2.2 2. products.csv

- **Description**: Contains information about products.
- **Fields**:
  - product_id (int): Unique identifier for each product.
  - product_name (string): Name of the product.
  - category (string): Product category (e.g., Electronics, Clothing).
  - price (float): Unit price of the product.
  - popularity (int): Popularity score (1–10).
  - region (string): Shipping region for the product (e.g., North America, Europe).

2.3 3. transactions.json

- **Description**: Contains information about transactions.
- **Fields**:
  - transaction_id (int): Unique identifier for each transaction.
  - customer_id (int): ID referencing a row in customers.csv.
  - product_id (int): ID referencing a row in products.csv.
  - quantity (int): Number of items purchased in the transaction.
  - price (float): Unit price of the product.
  - shipping_cost (float): Shipping cost for the transaction.
  - tax (float): Tax amount applied to the transaction.
  - total_amount (float): Computed total cost (quantity * price + shipping_cost + tax).
  - transaction_time (string, ISO format): Timestamp of the transaction (e.g., YYYY-MM-DDTHH:MM:SS).

2.4 4. reviews.txt

- **Description**: Semi-structured text file containing product reviews.
- **Format**: Each line follows the format: customer_id|product_id|product_name|review_text|rating.
- **Fields**:
  - customer_id (int): ID referencing a row in customers.csv.
  - product_id (int): ID referencing a row in products.csv.
  - product_name (string): Name of the reviewed product.
  - review_text (string): Freeform text describing the customer's opinion.
  - rating (int): Numeric score (1–5).

2.5 Notes

- **Relationships**:
  - `customer_id` links `transactions.json` and `reviews.txt` to `customers.csv`.
  - `product_id` links `transactions.json` and `reviews.txt` to `products.csv`.

Start by setting up a Spark session, enable Hive support so we can create databases and tables.

[1]:
```python
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Exercise2") \
    .config("spark.sql.warehouse.dir", "data_warehouse") \
    .enableHiveSupport() \
    .getOrCreate()

csv_customers_path = "../shared/customers.csv"
csv_products_path = "../shared/products.csv"
json_transactions_path = "../shared/transactions.json"
txt_reviews_path = "../shared/reviews.txt"

spark
```

[1]:
```
<pyspark.sql.session.SparkSession at 0x7fe22820ad50>
```

---

3 1. ETL: Load data into a Data Warehouse (6p)

3.1 Instructions

1. Define the following tables:
   - **`customers`**
   - **`products`**
   - **`transactions`**
   - **`reviews`**
2. Use **Parquet** format for optimized storage and query performance.
3. Write `CREATE TABLE` statements in Hive to define the schema.
4. **Optional**: Consider partitioning tables if you think it's reasonable, and explain the reasoning behind your decision.

[2]:
```python
spark.sql("CREATE DATABASE IF NOT EXISTS exercise_2_db")
spark.sql("USE exercise_2_db")

print("Databases in Spark:")
spark.sql("SHOW DATABASES").show()
```

```
Databases in Spark:
+-------------+
```

```
|    namespace|
+-------------+
|      default|
|exercise_2_db|
+-------------+
```

Tables

[3]:
```
# Create the tables
spark.sql("""
CREATE TABLE IF NOT EXISTS exercise_2_db.customers (
    customer_id INT,
    name STRING,
    age INT,
    country STRING,
    preferred_category STRING,
    loyalty_score FLOAT
)
USING PARQUET;
""")


spark.sql("""
CREATE TABLE IF NOT EXISTS exercise_2_db.products (
    product_id INT,
    product_name STRING,
    category STRING,
    price FLOAT,
    popularity INT,
    region STRING
)
USING PARQUET;
""")

spark.sql("""
CREATE TABLE IF NOT EXISTS exercise_2_db.transactions (
    transaction_id INT,
    customer_id INT,
    product_id INT,
    quantity INT,
    price FLOAT,
    shipping_cost FLOAT,
    tax FLOAT,
    total_amount FLOAT,
    transaction_time TIMESTAMP
)
USING PARQUET;
```

```
""")

spark.sql("""
CREATE TABLE IF NOT EXISTS exercise_2_db.reviews (
    customer_id INT,
    product_id INT,
    review_text STRING,
    rating INT
)
USING PARQUET;
""")
```

[3]: DataFrame[]

3.1.1 ETL Process

Now that we have defined the tables we can extract raw data, clean it, and load it into the predefined tables.

3.1.2 Instructions

1. Read raw data from the provided files located in the shared folder (`customers.csv`, `products.csv`, `transactions.json`, `reviews.txt`).
2. Apply transformations:
   - Cast columns to the correct data types.
   - Handle missing or invalid data (e.g., filter out rows with null IDs, if such rows exist)
   - Only insert the columns you find necessary.
3. Use spark.sql or DataFrame APIs to insert the cleaned data into the warehouse tables.

Customers ETL

[4]:
```
from pyspark.sql.functions import split, col

df_source = spark.read \
    .option("header", True) \
    .option("inferSchema", True) \
    .csv(csv_customers_path)

print("=== Source Data (Customers) ===")
df_source.show(5, truncate=False)

# Transform: Remove null customer_id and cast data types
df_cleaned = df_source\
    .filter(col("customer_id").isNotNull()) \
    .withColumn("age", col("age").cast("int")) \
    .withColumn("loyalty_score", col("loyalty_score").cast("float"))

# Load into table
```

```
df_cleaned.write.mode("overwrite").format("parquet").saveAsTable("exercise_2_db.
↪customers")
```

```
=== Source Data (Customers) ===
+-----------+-------------+---+--------------+------------------+-------------+
|customer_id|name         |age|country       |preferred_category|loyalty_score|
+-----------+-------------+---+--------------+------------------+-------------+
|1          |Cindy Simpson|60 |United Kingdom|Clothing          |0.15         |
|2          |Eric White   |41 |United Kingdom|Clothing          |0.22         |
|3          |Linda Todd   |54 |United Kingdom|Home              |0.5          |
|4          |Shannon Woods|52 |Canada        |Sports            |0.71         |
|5          |Michael Brown|48 |France        |Clothing          |0.36         |
+-----------+-------------+---+--------------+------------------+-------------+
only showing top 5 rows
```

Products ETL

[5]:
```
df_source = spark.read \
    .option("header", True) \
    .option("inferSchema", True) \
    .csv(csv_products_path)

print("=== Source Data (Products) ===")
df_source.show(5, truncate=False)

# Transform: Remove null product_id and cast data types
df_cleaned = df_source \
    .filter(col("product_id").isNotNull()) \
    .withColumn("price", col("price").cast("float")) \
    .withColumn("popularity", col("popularity").cast("int"))
# Load into table
df_cleaned.write.mode("overwrite").format("parquet").saveAsTable("exercise_2_db.
↪products")
```

```
=== Source Data (Products) ===
+----------+--------------+-----------+------+----------+-------------+
|product_id|product_name  |category   |price |popularity|region       |
+----------+--------------+-----------+------+----------+-------------+
|1         |Raincoat      |Clothing   |43.99 |10        |North America|
|2         |Sneakers      |Clothing   |25.99 |6         |Europe       |
|3         |Self-Help Book|Books      |48.99 |6         |Europe       |
|4         |Action Camera |Electronics|680.99|7         |North America|
|5         |4K Monitor    |Electronics|824.99|5         |North America|
+----------+--------------+-----------+------+----------+-------------+
only showing top 5 rows
```

Transactions ETL

[6]:
```python
df_source = spark.read \
    .option("multiline", True) \
    .json(json_transactions_path)

print("=== Source Data (Transactions) ===")
df_source.show(5, truncate=False)

# Transform: Convert transaction_time to TIMESTAMP, remove invalid rows and␣
↪cast data types
df_cleaned = df_source \
    .filter(
        col("transaction_id").isNotNull() &
        col("customer_id").isNotNull() &
        col("product_id").isNotNull()
    ) \
    .withColumn("transaction_id", col("transaction_id").cast("int")) \
    .withColumn("quantity", col("quantity").cast("int")) \
    .withColumn("price", col("price").cast("float")) \
    .withColumn("shipping_cost", col("shipping_cost").cast("float")) \
    .withColumn("tax", col("tax").cast("float")) \
    .withColumn("total_amount", col("total_amount").cast("float")) \
    .withColumn("transaction_time", col("transaction_time").cast("timestamp"))

# Load into table
df_cleaned.write.mode("overwrite").format("parquet").saveAsTable("exercise_2_db.
↪transactions")
```

```
=== Source Data (Transactions) ===
+-----------+------+----------+--------+-------------+----+------------+--------
------+--------------------------+
|customer_id|price |product_id|quantity|shipping_cost|tax
|total_amount|transaction_id|transaction_time          |
+-----------+------+----------+--------+-------------+----+------------+--------
------+--------------------------+
|32         |77.99 |22        |3       |11.03        |23.4|268.4       |1
|2024-11-09T09:48:12.057267|
|40         |384.99|40        |1       |13.44        |38.5|436.93      |2
|2024-08-06T08:26:45.609302|
|10         |174.99|18        |1       |19.44        |17.5|211.93      |3
|2024-02-26T12:33:55.105117|
|91         |19.99 |20        |5       |11.71        |9.99|121.65      |4
|2024-08-15T16:23:58.540147|
|86         |161.99|42        |2       |15.7         |32.4|372.08      |5
|2024-03-03T21:44:01.045684|
+-----------+------+----------+--------+-------------+----+------------+--------
------+--------------------------+
only showing top 5 rows
```

Reviews ETL

[7]:
```python
df_source = spark.read.text(txt_reviews_path)

print("=== Source Data (Reviews) ===")

# Split the column into separate fields using "|"
df_split = df_source.withColumn("customer_id", split(col("value"), "\\|")[0].
 ↪cast("int")) \
    .withColumn("product_id", split(col("value"), "\\|")[1].cast("int")) \
    .withColumn("product_name", split(col("value"), "\\|")[2].cast("string")) \
    .withColumn("review_text", split(col("value"), "\\|")[3].cast("string")) \
    .withColumn("rating", split(col("value"), "\\|")[4].cast("int")) \
    .drop("value")

df_split.show(5, truncate=False)

# Transform: Drop product_name for normalization purposes, remove invalid rows
df_cleaned = df_split.drop("product_name") \
    .filter(df_split.customer_id.isNotNull() & df_split.product_id.isNotNull())

# Load into table
df_cleaned.write.mode("overwrite").format("parquet").saveAsTable("exercise_2_db.
 ↪reviews")
```

```
=== Source Data (Reviews) ===
+-----------+----------+-----------------+-----------------------------------
-------------------------------------------------------------+------+
|customer_id|product_id|product_name     |review_text
|rating|
+-----------+----------+-----------------+-----------------------------------
-------------------------------------------------------------+------+
|5          |16        |Running Shoes    |Absolutely worth the price! The
quality is unmatched. Amazing product, highly recommend!              |5     |
|64         |46        |Exercise Bike    |Absolutely worth the price! The
quality is unmatched. Amazing product, highly recommend!              |5     |
|65         |14        |Gaming Console   |Fantastic build quality. You get what
you pay for! Audio quality is clear and immersive.             |4     |
|67         |38        |Historical Fiction|A premium product that delivers
premium results. A must-read for fans of the genre.                   |5     |
|64         |4         |Action Camera    |Absolutely worth the price! The
quality is unmatched. The battery life is phenomenal, lasts all day!|5     |
+-----------+----------+-----------------+-----------------------------------
-------------------------------------------------------------+------+
only showing top 5 rows
```

3.2 Analyze the Data

3.3 Objective

Run SQL queries to analyze the transformed data.

3.3.1 Example Queries to Run

1. **Total Revenue and Transactions per Product Category**
2. **Identify the 5 Least Sold Products**
3. **Identify the Top 5 Spending Customers**

You are encouraged to run these queries, but feel free to explore the data and create your own queries if you believe they provide better insights or are more relevant for analysis.

1. **Total Revenue and Transactions per Product Category**

[8]:
```
query = """
SELECT
    products.category,
    COUNT(transactions.transaction_id) AS total_transactions,
    ROUND(SUM(transactions.total_amount), 2) AS total_revenue
FROM exercise_2_db.transactions
JOIN exercise_2_db.products ON transactions.product_id = products.product_id
GROUP BY products.category
ORDER BY total_revenue DESC
"""

df_result = spark.sql(query)
df_result.show()
```

```
+-----------+------------------+-------------+
|   category|total_transactions|total_revenue|
+-----------+------------------+-------------+
|Electronics|               189|    154211.24|
|   Clothing|               300|     65483.09|
|     Sports|               171|     61551.06|
|       Home|               111|     50321.74|
|      Books|               229|     25490.83|
+-----------+------------------+-------------+
```

2. **5 Least Sold Products**

[9]:
```
query = """
SELECT
    products.product_name,
    SUM(transactions.quantity) AS total_quantity_sold
FROM exercise_2_db.transactions
```

```
JOIN exercise_2_db.products ON transactions.product_id = products.product_id
GROUP BY products.product_name
ORDER BY total_quantity_sold ASC
LIMIT 5
"""

df_result = spark.sql(query)
df_result.show()
```

```
+--------------------+-------------------+
|        product_name|total_quantity_sold|
+--------------------+-------------------+
|        Air Purifier|                  5|
|          Wall Clock|                  5|
|        Coffee Maker|                  5|
|       Action Camera|                  7|
|Noise-Canceling H...|                  9|
+--------------------+-------------------+
```

3. **Top 5 Spending Customers**

[10]:
```
query = """
SELECT
    customers.name AS customer_name,
    ROUND(SUM(transactions.total_amount), 2)  AS total_spent
FROM exercise_2_db.transactions
JOIN exercise_2_db.customers ON transactions.customer_id = customers.customer_id
GROUP BY customers.name
ORDER BY total_spent DESC
LIMIT 5
"""

df_result = spark.sql(query)
df_result.show()
```

```
+----------------+-----------+
|   customer_name|total_spent|
+----------------+-----------+
|     Nancy Jones|   15662.42|
|     Cesar Davis|   14997.11|
|Valerie Mitchell|   14160.32|
|  Anthony Pruitt|   12767.85|
|  Nicholas Davis|   11635.65|
+----------------+-----------+
```

4 2. ELT: Load Raw Data into a Data Lake (6p)

4.1 Objective

Copy the raw data files into a `data_lake/` directory and transform the data on read.

4.1.1 Instructions

1. Copy or use shell commands or scripts to move the files into a `data_lake/` directory in your `my-work` folder.
2. Do not modify the files; load them "as is" to retain their raw state.

Now the `data_lake/` folder contains all raw files, unmodified:

"'plaintext data_lake/ customers.csv products.csv transactions.json reviews.txt

5 Transform and Analyze

5.0.1 Instructions

1. Read the raw files from the `data_lake/` directory using Spark.
2. Clean and transform the data on read.
3. Register the transformed DataFrames as temporary views.
4. Run the same queries as in the warehouse approach:

- Total Revenue and Transactions per Product Category
- Identify the 5 Least Sold Products
- Identify the Top 5 Customers by Spending

You are encouraged to run these queries, but feel free to explore the data and create your own queries if you believe they provide better insights or are more relevant for analysis.

Copy the raw data files into a data_lake/ directory

[11]:
```
!mkdir -p data_lake
!cp ../shared/customers.csv data_lake/
!cp ../shared/products.csv data_lake/
!cp ../shared/transactions.json data_lake/
!cp ../shared/reviews.txt data_lake/
```

1. **Read the raw files**

[12]:
```python
df_customers = spark.read.csv("data_lake/customers.csv", header=True)
df_products = spark.read.csv("data_lake/products.csv", header=True)
df_transactions = spark.read.json("data_lake/transactions.json")
df_reviews = spark.read.text("data_lake/reviews.txt")
```

2. **Clean and transform the data on read.**

[13]:

```python
transformed_customers = df_customers.select(
    col("customer_id").cast("int"),
    col("name"),
    col("age").cast("int"),
    col("country"),
    col("preferred_category"),
    col("loyalty_score").cast("float")
)

cleaned_customers = transformed_customers.filter(col("customer_id").isNotNull())

transformed_products = df_products.select(
    col("product_id").cast("int"),
    col("product_name"),
    col("category"),
    col("price").cast("float"),
    col("popularity").cast("int"),
    col("region")
)

cleaned_products = transformed_products.filter(col("product_id").isNotNull())

transformed_transactions = df_transactions.select(
    col("transaction_id").cast("int"),
    col("customer_id").cast("int"),
    col("product_id").cast("int"),
    col("quantity").cast("int"),
    col("price").cast("float"),
    col("shipping_cost").cast("float"),
    col("tax").cast("float"),
    col("total_amount").cast("float"),
    col("transaction_time").cast("timestamp")
)

cleaned_transactions = transformed_transactions \
    .filter(
        col("transaction_id").isNotNull() &
        col("customer_id").isNotNull() &
        col("product_id").isNotNull()
    ) \
    .withColumn("transaction_time", col("transaction_time").cast("timestamp"))

transformed_reviews = df_reviews.select(
    split(col("value"), "\|").getItem(0).cast("int").alias("customer_id"),
    split(col("value"), "\|").getItem(1).cast("int").alias("product_id"),
    split(col("value"), "\|").getItem(2).alias("product_name"),
    split(col("value"), "\|").getItem(3).alias("review_text"),
```

```
    split(col("value"), "\|").getItem(4).cast("int").alias("rating")
).drop("value")

cleaned_reviews = transformed_reviews.drop("product_name") \
    .filter(transformed_reviews.customer_id.isNotNull() & transformed_reviews.
 ↪product_id.isNotNull())
```

3. **Register the transformed DataFrames as temporary views.**

[14]:
```
cleaned_customers.createOrReplaceTempView("customers_view")
cleaned_products.createOrReplaceTempView("products_view")
cleaned_transactions.createOrReplaceTempView("transactions_view")
cleaned_reviews.createOrReplaceTempView("reviews_view")
```

5.0.2 Analyze the Data

4. **Run the same queries as in the warehouse approach:**

4.1. **Total Revenue and Transactions per Product Category**

[15]:
```
spark.sql("""
    SELECT
        p.category,
        COUNT(t.transaction_id) AS total_transactions,
        ROUND(SUM(t.total_amount), 2) AS total_revenue
    FROM transactions_view AS t
    JOIN products_view AS p ON t.product_id = p.product_id
    GROUP BY p.category
    ORDER BY total_revenue DESC
""").show(truncate=False)
```

```
+-----------+------------------+-------------+
|category   |total_transactions|total_revenue|
+-----------+------------------+-------------+
|Electronics|189               |154211.24    |
|Clothing   |300               |65483.09     |
|Sports     |171               |61551.06     |
|Home       |111               |50321.74     |
|Books      |229               |25490.83     |
+-----------+------------------+-------------+
```

4.2. **5 Least Sold Products**

[16]:
```
spark.sql("""
    SELECT
        p.product_name,
        SUM(t.quantity) AS total_sold
    FROM transactions_view AS t
```

```
    JOIN products_view As p ON t.product_id = p.product_id
    GROUP BY p.product_name
    ORDER BY total_sold ASC
    LIMIT 5
""").show(truncate=False)
```

```
+--------------------------+----------+
|product_name              |total_sold|
+--------------------------+----------+
|Air Purifier              |5         |
|Wall Clock                |5         |
|Coffee Maker              |5         |
|Action Camera             |7         |
|Noise-Canceling Headphones|9         |
+--------------------------+----------+
```

4.3. **Top 5 Customers by Spending**

[17]:
```
spark.sql("""
    SELECT
        c.name AS customer_name,
        ROUND(SUM(t.total_amount), 2) AS total_spent
    FROM transactions_view AS t
    JOIN customers_view AS c ON t.customer_id = c.customer_id
    GROUP BY c.name
    ORDER BY total_spent DESC
    LIMIT 5
""").show(truncate=False)
```

```
+----------------+-----------+
|customer_name   |total_spent|
+----------------+-----------+
|Nancy Jones     |15662.42   |
|Cesar Davis     |14997.11   |
|Valerie Mitchell|14160.32   |
|Anthony Pruitt  |12767.85   |
|Nicholas Davis  |11635.65   |
+----------------+-----------+
```

5.0.3 Questions (6p)

Reflect on the following questions and provide thoughtful answers. Focus on your reasoning, insights, and key takeaways from the exercise.

1. What were the key differences in how data was handled and queried in the warehouse (ETL) versus the lake (ELT)? Which approach felt more adaptable to changes

in data structure or format, and why?

The key difference between ETL and ELT lies in when and where data transformation happens:

ETL (Warehouse): Data is transformed before loading into the warehouse, ensuring structured, clean, and optimized data for queries. ETL queries more efficient but can be rigid when handling evolving schemas.

ELT (Lake): Raw data is loaded first as-is, and the transformation happens later as needed. ELT is flexible for schema changes and accommodates semi-structured data, but queries were slower due to on-the-fly transformations.

For adaptability, **ELT** is more flexible, especially when handling different data formats (Example: CSV, JSON and TXT files in our case), since schema changes do not require reloading the entire dataset.

2. What challenges did you encounter when transforming and querying the data in each approach? How did these challenges help you better understand the trade-offs of schema-on-write vs. schema-on-read?

ETL challenges: Defining a strict schema and transforming raw data upfront before loading it into the storage was time-consuming. This can be seen as a drawback when fast data ingestion is a priority, especially with increasingly varied unstructured data.. However, queries were straightforward and efficient.

ELT challenges: We needed to transform the raw data and define the schema at runtime, which made the queries slower. Schema-on-read made querying more complex. However, this approach can be more suitable when the speed of data ingestion is a priority.

These challenges helped us to understand that schema-on-write ensures cleaner, optimized queries but lacks flexibility, while schema-on-read adapts better to diverse data but requires more processing at query time.

3. What factors would you consider when deciding between a warehouse, a lake, or a hybrid approach for a real-world data solution?

We would consider several factors:

1. **Data Type & Structure:** If dealing with structured data, a warehouse is better choice. For unstructured or semi-structured data, a lake is better.
2. **Query Performance Needs:** Warehouses are optimized for fast analytics, while lakes may require extra processing.
3. **Scalability & Cost:** Data lakes offer low-cost storage and can handle massive datasets.

So,
Data Warehouse → If real-time analytics and business intelligence are needed
Data Lakes → If flexibility and machine learning applications are the focus
Hybrid Approach → If both structured reporting and unstructured data analysis are required