Exercise2_Group1

January 28, 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]:
```
from pyspark.sql import SparkSession

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

spark
```

[1]:
```
<pyspark.sql.session.SparkSession at 0x7f5d70185f50>
```

---

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]:
```
# Creating a database to store tables
spark.sql("CREATE DATABASE IF NOT EXISTS ecommerce")
spark.sql("USE ecommerce")
```

[2]:
```
DataFrame[]
```

[3]:
```
print("Databases in Spark:")
spark.sql("SHOW DATABASES").show()
```

```
Databases in Spark:
+---------+
|namespace|
+---------+
```

```
|   default|
|ecommerce|
+----------+
```

[4]:
```
# Customers table - Partitioned by country because it is useful for queries␣
↪with country as a filter
spark.sql("""
CREATE TABLE IF NOT EXISTS ecommerce.customers (
    customer_id INT,
    name STRING,
    age INT,
    country STRING,
    preferred_category STRING,
    loyalty_score FLOAT
)
USING PARQUET
PARTITIONED BY (country)
""")
```

[4]: DataFrame[]

[5]:
```
# Products table - Partitioned by region because it is useful for queries with␣
↪regions as a filter (customers from Europe will have Europe as a filter)
spark.sql("""
CREATE TABLE IF NOT EXISTS ecommerce.products (
    product_id INT,
    product_name STRING,
    category STRING,
    price FLOAT,
    popularity INT,
    region STRING
)
USING PARQUET
PARTITIONED BY (region)
""")
```

[5]: DataFrame[]

[6]:
```
# Transactions table - improves query performance for time-based analysis,␣
↪which is the most common analysis for transactions
spark.sql("""
CREATE TABLE IF NOT EXISTS ecommerce.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
PARTITIONED BY (transaction_time)
""")
```

[6]: DataFrame[]

[7]:
```
# Reviews table - improves speed for analysis of reviews by rating
spark.sql("""
CREATE TABLE IF NOT EXISTS ecommerce.reviews (
    customer_id INT,
    product_id INT,
    product_name STRING,
    review_text STRING,
    rating INT
)
USING PARQUET
PARTITIONED BY (rating)
""")
```

[7]: 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.

[8]:
```
# Load raw data from shared folder
customers_df = spark.read.csv("customers.csv", header=True, inferSchema=True)
products_df = spark.read.csv("products.csv", header=True, inferSchema=True)
transactions_df = spark.read.json("transactions.json")
reviews_rdd = spark.sparkContext.textFile("reviews.txt")
```

[9]:
```python
from pyspark.sql.functions import col, when
```

[10]:
```python
# Clean customers data
# All columns kept
cleaned_customers_df = (
    customers_df
    .filter(col("customer_id").isNotNull())  # Filter out rows with null␣
↪customer_id
    .select(
        col("customer_id").cast("int"),
        col("name").cast("string"),
        col("age").cast("int"),
        col("country").cast("string"),
        col("preferred_category").cast("string"),
        col("loyalty_score").cast("float")
    )
)
```

[11]:
```python
# Clean products data
# All columns kept
cleaned_products_df = (
    products_df
    .filter(col("product_id").isNotNull())  # Filter out rows with null␣
↪product_id
    .select(
        col("product_id").cast("int"),
        col("product_name").cast("string"),
        col("category").cast("string"),
        col("price").cast("float"),
        col("popularity").cast("int"),
        col("region").cast("string")
    )
)
```

[12]:
```python
# Clean transactions data
# All columns kept
cleaned_transactions_df = (
    transactions_df
    .filter(col("transaction_id").isNotNull())  # Filter out rows with null␣
↪transaction_id
    .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")
    )
)
```

[13]:
```
# Parse reviews data from RDD
reviews_df = reviews_rdd.map(lambda line: line.split("|")).toDF([
    "customer_id", "product_id", "product_name", "review_text", "rating"
])

# Clean reviews data
# All columns kept
cleaned_reviews_df = (
    reviews_df
    .filter(col("customer_id").isNotNull() & col("product_id").isNotNull())  #␣
 ↪Filter out rows with null IDs
    .select(
        col("customer_id").cast("int"),
        col("product_id").cast("int"),
        col("product_name").cast("string"),
        col("review_text").cast("string"),
        col("rating").cast("int")
    )
)
```

[14]:
```
# Write data to warehouse tables
cleaned_customers_df.write.mode("overwrite").insertInto("ecommerce.customers")
cleaned_products_df.write.mode("overwrite").insertInto("ecommerce.products")
cleaned_transactions_df.write.mode("overwrite").insertInto("ecommerce.
 ↪transactions")
cleaned_reviews_df.write.mode("overwrite").insertInto("ecommerce.reviews")
```

[15]:
```
# Check table contents
spark.sql("SELECT * FROM ecommerce.customers LIMIT 10").show()
spark.sql("SELECT * FROM ecommerce.products LIMIT 10").show()
spark.sql("SELECT * FROM ecommerce.transactions LIMIT 10").show()
spark.sql("SELECT * FROM ecommerce.reviews LIMIT 10").show()
```

```
+-----------+-----------------+---+--------------+-----------------+-----------
--+
|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|
|          6|Priscilla Stewart| 72|        France|             Books|
0.06|
|          7|    Katie Allison| 24|United Kingdom|       Electronics|
0.04|
|          8|     Jeremy Weiss| 73| United States|             Books|
0.11|
|          9| Shelly Castaneda| 59|        Canada|            Sports|
0.13|
|         10|      Karen Jones| 45|        France|            Sports|
0.44|
+-----------+-----------------+---+--------------+------------------+-----------
--+

+----------+-----------------+-----------+------+----------+-------------+
|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|
|         6|     Dumbbell Set|     Sports|229.99|         8|       Europe|
|         7|  Mattress Topper|       Home|467.99|         1|North America|
|         8|         Curtains|       Home|128.99|         2|       Europe|
|         9| Resistance Bands|     Sports| 83.99|         4|North America|
|        10|Programming Guide|      Books| 46.99|         3|       Europe|
+----------+-----------------+-----------+------+----------+-------------+

+--------------+-----------+----------+--------+------+-------------+-----+-----
-------+--------------------+
|transaction_id|customer_id|product_id|quantity| price|shipping_cost|
tax|total_amount|    transaction_time|
+--------------+-----------+----------+--------+------+-------------+-----+-----
-------+--------------------+
|             1|         32|        22|       3| 77.99|        11.03| 23.4|
268.4|2024-11-09 09:48:…|
|             2|         40|        40|       1|384.99|        13.44| 38.5|
436.93|2024-08-06 08:26:…|
|             3|         10|        18|       1|174.99|        19.44| 17.5|
211.93|2024-02-26 12:33:…|
```

```
|             4|         91|        20|       5| 19.99|         11.71| 9.99|
121.65|2024-08-15 16:23:…|
|             5|         86|        42|       2|161.99|          15.7| 32.4|
372.08|2024-03-03 21:44:…|
|             6|         90|        21|       5| 74.99|         19.17|37.49|
431.61|2024-09-14 00:39:…|
|             7|         96|        49|       3|263.99|          8.02| 79.2|
879.19|2024-08-10 07:33:…|
|             8|         47|        11|       2| 66.99|          5.19| 13.4|
152.57|2024-02-10 04:48:…|
|             9|         78|         5|       1|824.99|          6.78| 82.5|
914.27|2024-03-22 16:02:…|
|            10|         71|        16|       1|185.99|          8.86| 18.6|
213.45|2024-10-26 06:17:…|
+--------------+-----------+----------+--------+------+--------------+-----+-----
-------+--------------------+
```

```
+-----------+----------+----------------+------------------+------+
|customer_id|product_id|    product_name|       review_text|rating|
+-----------+----------+----------------+------------------+------+
|         92|         6|    Dumbbell Set|Fantastic build q…|     5|
|         17|        50|Wireless Earbuds|A premium product…|     4|
|         34|         8|        Curtains|Exceeded my expec…|     5|
|          7|        46|   Exercise Bike|High-quality and …|     4|
|         35|        31|    Water Bottle|A premium product…|     4|
|          4|        18|   Tennis Racket|Does the job, but…|     3|
|         57|        46|   Exercise Bike|Exceeded my expec…|     4|
|         76|        32|  Vacuum Cleaner|Decent quality fo…|     3|
|         75|        42|        Yoga Mat|Good for the pric…|     3|
|         22|        17|     Formal Suit|Absolutely worth …|     5|
+-----------+----------+----------------+------------------+------+
```

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.

[18]:
```
# Total Revenue and Transactions per Product Category
spark.sql("""
    SELECT p.category, SUM(t.price) as total_revenue, COUNT(transaction_id) as␣
 ↪num_of_transactions
    FROM transactions t
    JOIN products p
    ON t.product_id = p.product_id
    GROUP BY p.category
    ORDER BY num_of_transactions DESC
""").show()
```

```
+-----------+------------------+-------------------+
|   category|     total_revenue|num_of_transactions|
+-----------+------------------+-------------------+
|   Clothing|21233.000051498413|                300|
|      Books| 6439.710102081299|                229|
|Electronics| 83316.10888671875|                189|
|     Sports|31397.290817260742|                171|
|       Home|27461.890014648438|                111|
+-----------+------------------+-------------------+
```

[20]:
```
#Identify the 5 Least Sold Products
spark.sql("""
    SELECT p.product_name, SUM(t.quantity) as total_quantity_sold
    FROM transactions t
    JOIN products p
    ON t.product_id = p.product_id
    GROUP BY p.product_name
    ORDER BY total_quantity_sold ASC
    LIMIT 5
""").show()
```

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

[21]:
```
#Identify the Top 5 Spending Customers
spark.sql("""
    SELECT c.name, SUM(t.total_amount) as total_money_spent
```

```
    FROM transactions t
    JOIN customers c
    ON t.customer_id = c.customer_id
    GROUP BY c.name
    ORDER BY total_money_spent DESC
    LIMIT 5
""").show()
```

```
+----------------+------------------+
|            name| total_money_spent|
+----------------+------------------+
|     Nancy Jones|   15662.419921875|
|     Cesar Davis| 14997.11003112793|
|Valerie Mitchell|14160.320007324219|
|  Anthony Pruitt|12767.850021362305|
|  Nicholas Davis|11635.649841308594|
+----------------+------------------+
```

[43]:
```
# Product categories ranked by ratings
spark.sql("""
    SELECT p.category, AVG(r.rating) as avg_rating
    FROM reviews r
    JOIN products p
    ON r.product_id = p.product_id
    GROUP BY p.category
    ORDER BY avg_rating DESC
""").show()
```

```
+-----------+------------------+
|   category|        avg_rating|
+-----------+------------------+
|Electronics| 4.194444444444445|
|     Sports|3.6315789473684212|
|       Home|3.5813953488372094|
|   Clothing|3.1463414634146343|
|      Books| 3.119047619047619|
+-----------+------------------+
```

---

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

[22]:
```
mkdir -p data_lake
```

[23]:
```
import shutil
import os

# Paths
source_dir = "./"
destination_dir = "data_lake/"

# Ensure destination directory exists
os.makedirs(destination_dir, exist_ok=True)

# List of files to copy
files = ["customers.csv", "products.csv", "transactions.json", "reviews.txt"]

# Copy files
for file in files:
    shutil.copy(os.path.join(source_dir, file), destination_dir)

print("Files copied to data_lake/ successfully!")
```

```
Files copied to data_lake/ successfully!
```

[24]:
```
ls -l data_lake/
```

```
total 216
-rw-r--r--. 1 jovyan root   4430 Jan 28 17:19 customers.csv
-rw-r--r--. 1 jovyan root   2235 Jan 28 17:19 products.csv
-rw-r--r--. 1 jovyan root  20147 Jan 28 17:19 reviews.txt
-rw-r--r--. 1 jovyan root 185370 Jan 28 17:19 transactions.json
```

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.

[27]:
```
# Load raw data from data_lake folder
customers_df = spark.read.csv("data_lake/customers.csv", header=True,␣
↪inferSchema=True)
products_df = spark.read.csv("data_lake/products.csv", header=True,␣
↪inferSchema=True)
transactions_df = spark.read.json("data_lake/transactions.json")
reviews_rdd = spark.sparkContext.textFile("data_lake/reviews.txt")
```

[28]:
```
# Clean customers data
cleaned_customers_df = (
    customers_df
    .filter(col("customer_id").isNotNull())  # Filter out rows with null␣
↪customer_id
    .select(
        col("customer_id").cast("int"),
        col("name").cast("string"),
        col("age").cast("int"),
        col("country").cast("string"),
        col("preferred_category").cast("string"),
        col("loyalty_score").cast("float")
    )
)
```

[29]:
```
# Clean products data
cleaned_products_df = (
    products_df
    .filter(col("product_id").isNotNull())  # Filter out rows with null␣
↪product_id
    .select(
        col("product_id").cast("int"),
        col("product_name").cast("string"),
        col("category").cast("string"),
        col("price").cast("float"),
        col("popularity").cast("int"),
        col("region").cast("string")
    )
)
```

[30]:

```
# Clean transactions data
cleaned_transactions_df = (
    transactions_df
    .filter(col("transaction_id").isNotNull())  # Filter out rows with null␣
 ↪transaction_id
    .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")
    )
)
```

[31]:

```
# Parse reviews data from RDD
reviews_df = reviews_rdd.map(lambda line: line.split("|")).toDF([
    "customer_id", "product_id", "product_name", "review_text", "rating"
])

# Clean reviews data
cleaned_reviews_df = (
    reviews_df
    .filter(col("customer_id").isNotNull() & col("product_id").isNotNull())  #␣
 ↪Filter out rows with null IDs
    .select(
        col("customer_id").cast("int"),
        col("product_id").cast("int"),
        col("product_name").cast("string"),
        col("review_text").cast("string"),
        col("rating").cast("int")
    )
)
```

[32]:

```
# Temporary views
cleaned_customers_df.createOrReplaceTempView("customers_view")
cleaned_products_df.createOrReplaceTempView("products_view")
cleaned_transactions_df.createOrReplaceTempView("transactions_view")
cleaned_reviews_df.createOrReplaceTempView("reviews_view")
```

[33]:

```
#Total Revenue and Transactions per Product Category
spark.sql("""
    SELECT p.category, SUM(t.price) as total_revenue, COUNT(transaction_id) as␣
 ↪num_of_transactions
```

```
    FROM transactions_view t
    JOIN products_view p
    ON t.product_id = p.product_id
    GROUP BY p.category
    ORDER BY num_of_transactions DESC
""").show()
```

```
+-----------+-----------------+-------------------+
|   category|    total_revenue|num_of_transactions|
+-----------+-----------------+-------------------+
|   Clothing|21233.000051498413|                300|
|      Books| 6439.710102081299|                229|
|Electronics| 83316.10888671875|                189|
|     Sports|31397.290817260742|                171|
|       Home|27461.890014648438|                111|
+-----------+-----------------+-------------------+
```

[34]:

```
#Identify the 5 Least Sold Products
spark.sql("""
    SELECT p.product_name, SUM(t.quantity) as total_quantity_sold
    FROM transactions_view t
    JOIN products_view p
    ON t.product_id = p.product_id
    GROUP BY p.product_name
    ORDER BY total_quantity_sold ASC
    LIMIT 5
""").show()
```

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

[35]:

```
#Identify the Top 5 Customers by Spending
spark.sql("""
    SELECT c.name, SUM(t.total_amount) as total_money_spent
    FROM transactions_view t
    JOIN customers_view c
    ON t.customer_id = c.customer_id
    GROUP BY c.name
```

```
    ORDER BY total_money_spent DESC
    LIMIT 5
""").show()
```

```
+----------------+------------------+
|            name| total_money_spent|
+----------------+------------------+
|     Nancy Jones|   15662.419921875|
|     Cesar Davis| 14997.11003112793|
|Valerie Mitchell|14160.320007324219|
|  Anthony Pruitt|12767.850021362305|
|  Nicholas Davis|11635.649841308594|
+----------------+------------------+
```

[36]:
```
# Product categories ranked by ratings
spark.sql("""
    SELECT p.category, AVG(r.rating) as avg_rating
    FROM reviews_view r
    JOIN products_view p
    ON r.product_id = p.product_id
    GROUP BY p.category
    ORDER BY avg_rating DESC
""").show()
```

```
+-----------+------------------+
|   category|        avg_rating|
+-----------+------------------+
|Electronics| 4.194444444444445|
|     Sports|3.6315789473684212|
|       Home|3.5813953488372094|
|   Clothing|3.1463414634146343|
|      Books| 3.119047619047619|
+-----------+------------------+
```

5.0.2 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?

When working with the warehouse (ETL) approach, the data was cleaned, transformed, and structured before loading it into predefined tables. This ensured the data was consistent and ready for analysis, with a fixed schema. Queries ran smoothly because the data was already cleaned and optimized for performance.

On the other hand, with the lake (ELT) approach, the raw files were directly loaded into the data lake without modifying them. Transformations and schema were applied dynamically at query time. This was more flexible because the raw data remained as raw, allowing for adjustment of queries or transformations as needed.

Given the explanations above, the ELT approach appears to be more adaptable. Since the raw data was stored without enforcing a schema, the changes in data structure or format could be handled easily. For example, if new fields were added or the file format was changed, the transformations or queries could be simply updated without reloading the data. However, the schema needs to be defined upfront in ETL approach, which makes that approach less flexible.

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?

In ETL, the data needed to be cleaned and filtered (removing rows with missing values, matching the data types defined in the tables) to match the predefined schema, otherwise it wouldn't be inserted into tables. In ELT, the approach is more flexible but transforming the data dynamically during querying requires extra effort to handle raw files, and extra time -> performance might be affected. This may not me noticable for small datasets, but for larger ones constantly applying transformations when querying can impact the performance. The trade-offs are very general, if the goal is to care less about the data structure and load it as fast as possible, then querying will be less efficient and take more time. On the other hand, if we want to spend more time working on processing the data before loading it in tables, the query performance will be optimized afterwards.

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

A data warehouse is better for structured data, such as transactional records, or data where schema is constant and will not change in the future, where schema consistency and high-performance querying are essential. On the other hand, a data lake is better for handling unstructured and semi-structured data, such as images, videos etc. If we are dealing with a combination of both structured and unstructured data, a hybrid approach can be used.

DW is better for BI, reporting and tasks which require repeatable queries. Data lake is better for analysis, machine learning and big data operations since flexibility is needed for such cases. When both types of analysis are needed, it would be better to use the hybrid approach.

DW requires higher costs but it performs fast. On the other hand, lakes work slower but with lower costs. So depending on the trade off between cost and performance (what is the most important for the task at hand), one can choose one of the approaches.