Big Data Platform

Presented by:

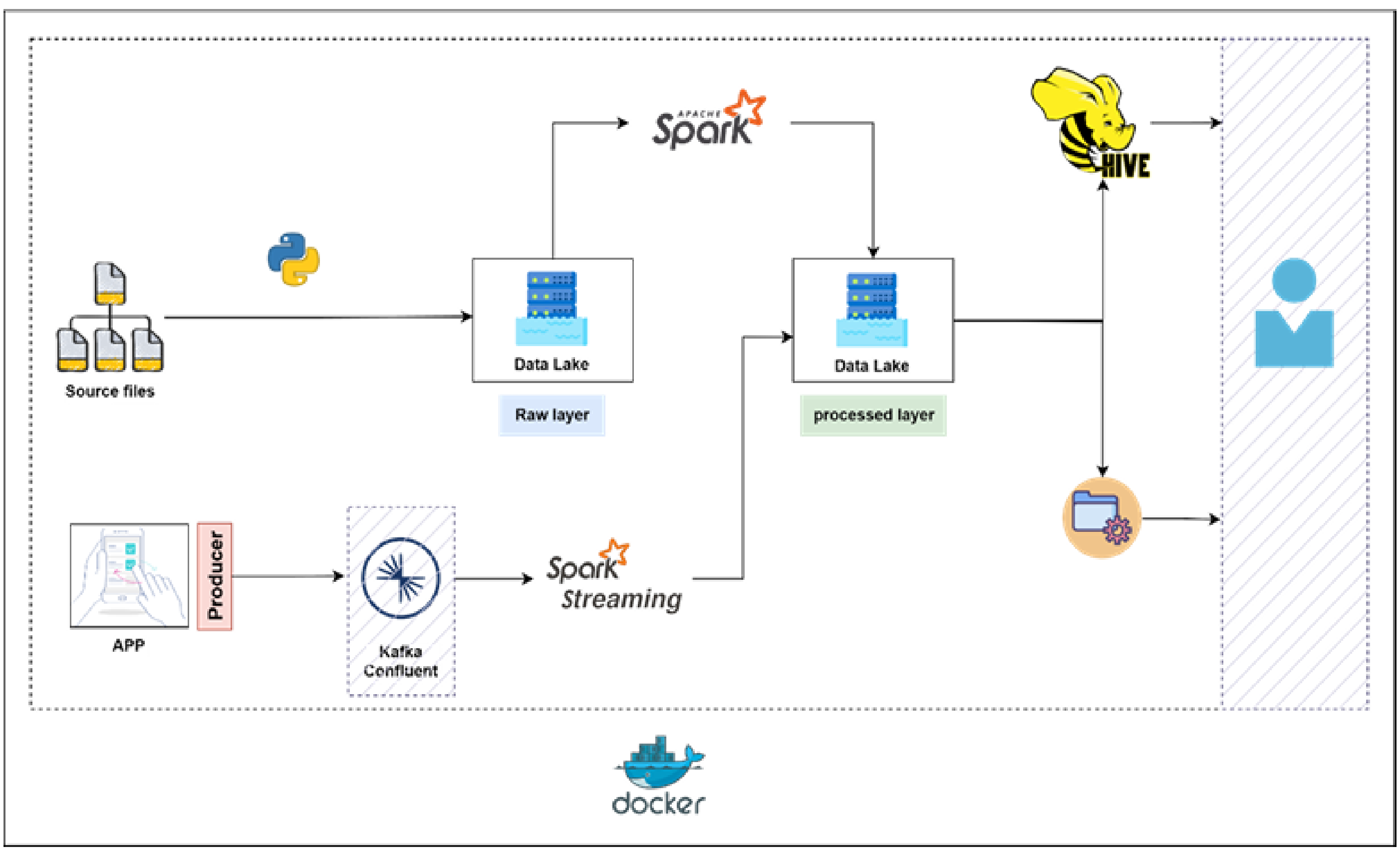
Radwa Esmaiel - Mark Girgis

Building a Scalable Data Platform for Q Company: Enhancing Retail Insights with Spark and Hive

Welcome to our documentation on developing a robust data platform for Q Company, a leading retailer operating through both physical branches and an E-commerce platform. In this project, we designed and implemented a scalable solution to manage and process data efficiently, leveraging the power of Apache Spark and Apache Hive.

Our data platform handles both batch and streaming data, ensuring timely and accurate insights for various business needs. Every hour, new files are ingested into our data lake, processed, and stored in a Hive-based data warehouse. This allows for seamless tracking of transactions, sales performance, and customer behavior.

Additionally, our platform processes real-time logs from the company app using Kafka and Spark Streaming, providing actionable insights from dynamic data. This presentation will walk you through the technical implementation and business insights derived from our data platform, demonstrating its impact on Q Company's operations and decision-making processes.



1- Ingestion.py:

Automated File Ingestion to HDFS

This Python script automates the ingestion of hourly data files from a local file system (LFS) to a Hadoop Distributed File System (HDFS) for Q Company's data platform. It handles the organization and transfer of data, ensuring each batch is processed and logged efficiently.

Browse Directory

/user/itversity/raw_layer Go!

Show 25 entries Search:

	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 02 02:00	0	0 B	20240701
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 03 01:00	0	0 B	20240702
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 04 02:00	0	0 B	20240703
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 05 00:00	0	0 B	20240704
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 02:59	0	0 B	20240705
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 09:40	0	0 B	20240706

Showing 1 to 6 of 6 entries Previous 1 Next

☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 12:03	0	0 B	09_group3
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 13:00	0	0 B	10_group4
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 14:00	0	0 B	11_group5
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 15:00	0	0 B	12_group6
☐	drwxr-xr-x	itversity	supergroup	0 B	Jul 06 16:00	0	0 B	13_group1

Browse Directory

/user/itversity/raw_layer/20240706/12_group6 Go!

Show 25 entries Search:

	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
☐	-rw-r--r--	itversity	supergroup	191 B	Jul 06 15:00	1	128 MB	branches_SS_raw_6_12_20240706.csv
☐	-rw-r--r--	itversity	supergroup	320 B	Jul 06 15:00	1	128 MB	sales_agents_SS_raw_6_12_20240706.csv
☐	-rw-r--r--	itversity	supergroup	68.02 MB	Jul 06 15:00	1	128 MB	sales_transactions_SS_raw_6_12_20240706.csv

Showing 1 to 3 of 3 entries Previous 1 Next

Hadoop, 2020.

Key Components:

1. **Directory and File Setup:**
   - **BASE_DIR:** Local directory containing data files organized in groups.
   - **HDFS_DIR:** HDFS target directory for raw data storage.
   - **LOG_FILE:** File to log the status of file transfers.
   - **INDEX_FILE:** File to track the last processed group index.
2. **Current Date and Time:**
   - Retrieves the current hour and day to create dynamic HDFS directories and log entries.
3. **Group Directory Management:**
   - A list of group directories (group1, group2, ..., group6) to simulate hourly file ingestion.
   - Reads the last processed group index from INDEX_FILE and determines the next group to process.
   - Updates the INDEX_FILE with the new index.
4. **HDFS Directory Creation:**
   - Constructs the target HDFS directory path using the current date, hour, and group directory name.
   - Executes a command to create the HDFS directory if it doesn't exist.

```
# Create the target HDFS directory if it doesn't exist

hdfs_target_dir = f"{HDFS_DIR}/{current_day}/{current_hour}_{group_dir}"

mkdir_command = ["/opt/hadoop/bin/hdfs", "dfs", "-mkdir", "-p", hdfs_target_dir]

print(f"Executing: {' '.join(mkdir_command)}")  # Print the command being executed

subprocess.run(mkdir_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
universal_newlines=True)
```

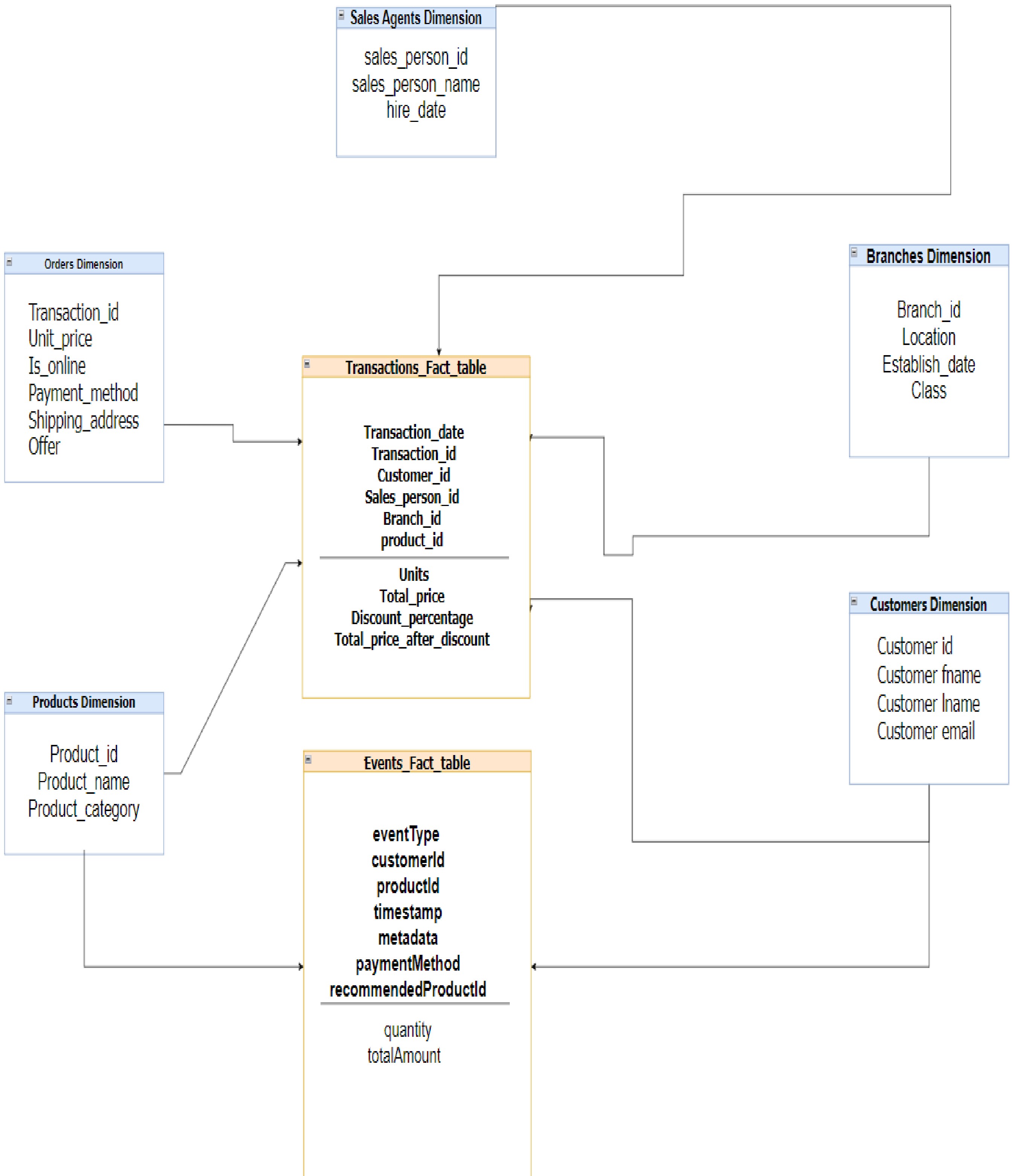
5. **File Transfer to HDFS:**
   - Iterates through files in the current group directory.
   - Constructs source file paths in LFS and destination file paths in HDFS.
   - Executes commands to move files from LFS to HDFS, renaming them to include the current date and hour.
   - Logs the success or failure of each file transfer in the LOG_FILE.
6. **Logging:**
   - Detailed logging of successful and failed file transfers to assist with monitoring and troubleshooting.

```
group_path = os.path.join(BASE_DIR, group_dir)

if os.path.exists(group_path):

    for file_name in os.listdir(group_path):

        src_file_path = os.path.join(group_path, file_name)

        dest_file_path =
f"{hdfs_target_dir}/{file_name.split('.')[0]}_{current_hour}_{current_day}.csv"

        put_command = ["/opt/hadoop/bin/hdfs", "dfs", "-put", src_file_path, dest_file_path]

        print(f"Executing: {' '.join(put_command)}")  # Print the command being executed

        result = subprocess.run(put_command, stdout=subprocess.PIPE,
stderr=subprocess.PIPE, universal_newlines=True)

        if result.returncode == 0:

            log_message = f"{now} Successfully moved {src_file_path} to {dest_file_path}\n"

        else:

            log_message = f"{now} Failed to move {src_file_path} to HDFS: {result.stderr}\n"

        with open(LOG_FILE, "a") as log_file:

            log_file.write(log_message)

else:

    log_message = f"{now} Directory {group_path} does not exist\n"

    with open(LOG_FILE, "a") as log_file:

        log_file.write(log_message)
```

This script ensures a systematic and traceable process for moving files from the local file system to HDFS, supporting the overall data ingestion pipeline for Q Company's data platform.

2- Hive.ipynb:

This Jupyter Notebook is designed to automate the creation of Hive tables required for the data platform at Q Company. The notebook includes functions to create and verify Hive tables, ensuring the data warehouse (DWH) structure is correctly established.



Key Components:

Table Creation Function:

A reusable function, create_and_verify_table, is defined to drop existing tables, create new tables with specified schemas, and verify the successful creation of these tables.

Database Creation:

Ensures the BigData_DWH database is created if it doesn't already exist.

```
def create_and_verify_table(create_table_query, table_name, schema_name):
  try:
    # Drop table if it exists
    spark.sql(f"DROP TABLE IF EXISTS {schema_name}.{table_name}")

    # Create new table with schema
    full_create_query = f"CREATE TABLE {schema_name}.{table_name} {create_table_query} STORED AS PARQUET"
    spark.sql(full_create_query)

    # Verify table creation
    table_exists = spark.sql(f"SHOW TABLES IN {schema_name} LIKE '{table_name}'").count() > 0
    if table_exists:
      print(f"Success: Table '{schema_name}.{table_name}' created successfully.")
    else:
      print(f"Failure: Table '{schema_name}.{table_name}' creation failed.")
  except Exception as e:
    print(f"Error creating table '{schema_name}.{table_name}': {e}")
```

Table Creation for Each Dimension:

Utilizes the create_and_verify_table function to create the necessary dimension and fact tables with specified schemas, including:

1. branches_dimension
2. sales_agents_dimension
3. Transactions_Fact_table
4. customers_dimension
5. products_dimension
6. orders_dimension

3- processing.ipynb:

This Spark script automates the processing of raw data files stored in HDFS and inserts the processed data into Hive tables for Q Company's data platform. It handles the extraction, transformation, and loading (ETL) of data, ensuring the creation of a well-structured Data Warehouse (DWH).

Key Components:

1. **Spark Session Initialization:**
    - Creates a Spark session with Hive support enabled.
2. **Date and Time Calculation:**
    - Determines the current date and previous hour, handling the special case for midnight processing.
3. **HDFS Input Directory Construction:**
    - Constructs the input path pattern based on the calculated date and hour.
    - Uses the Hadoop FileSystem API to list directories matching the input pattern.
4. **DataFrame Initialization:**
    - Initializes lists to store DataFrames for branches, agents, and transactions.
    - Loads existing data from Hive tables for comparison and deduplication.
5. **File Reading and DataFrame Creation:**
    - Iterates through the input directories and files.
    - Reads CSV files into DataFrames and categorizes them based on file names.
6. **Data Deduplication and Filtering:**
    - Unions and deduplicates new DataFrames.
    - Filters out existing records by performing anti-joins with existing Hive tables.
7. **Branches Data Processing:**
    - Merges new branch data and removes duplicates.
    - Shows the processed DataFrame for verification.
8. **Sales Agents Data Processing:**
    - Merges new sales agents data and removes duplicates.
    - Shows the processed DataFrame for verification.
9. **Transactions Data Processing:**
    - Merges new transactions data and removes duplicates.
    - Shows the processed DataFrame for verification.
    - Extracts and transforms specific columns for customer, product, and order dimensions.
10. **Customer and Product Dimensions:**

- Creates new DataFrames for customer and product dimensions.
- Inserts the data into respective Hive tables.

11. **Transactions Data Transformation:**

- Adds a derived "offer" column based on existing offer columns.
- Calculates the total price and price after discount.
- Shows the transformed DataFrame.

```
transactions_new_df = transactions_new_df.withColumn(
  "Total_price",
   col("units") * col("unit_price")
)
  # Add 'discount_percentage' column
  transactions_new_df = transactions_new_df.withColumn(
    "discount_percentage",
    when(col("offer") == "offer_1", 0.05)
    .when(col("offer") == "offer_2", 0.10)
    .when(col("offer") == "offer_3", 0.15)
    .when(col("offer") == "offer_4", 0.20)
    .when(col("offer") == "offer_5", 0.25)
    .otherwise(0.0)
  )
  # Add 'Total_price_paid_after_discount' column
  transactions_new_df = transactions_new_df.withColumn(
    "Total_price_after_discount",
    col("Total_price") * (1 - col("discount_percentage"))
  )
```

12. **Orders Dimension:**

- Creates a new DataFrame for the orders dimension.
- Inserts the data into the Hive table.

13. **Transactions Fact Table:**

- Selects and casts necessary columns for the transactions fact table.
- Inserts the data into the Hive table with partitioning based on the transaction date.

14. **Hive Table Insertions:**

- Inserts processed data into Hive tables for branches, sales agents, orders, customers, products, and transactions fact table.

15. **Spark Session Termination:**

- Stops the Spark session to release resources.

This script ensures the systematic processing of raw data files, transforming them into structured Hive tables that support the business's analytical needs, enabling insightful and actionable data analysis for Q Company.

transaction_date	transaction_id	customer_id	sales_agent_id	branch_id	product_id	units	Total_price	discount_percentage	Total_price_after_discount
2022-7-11	trx-187320333043	85476	10	5	8	7	559.93	0.0	559.93
2022-8-2	trx-211908445828	85510	6	5	24	4	199.96	0.0	199.96
2023-5-10	trx-504748286356	85474	null	null	2	2	1399.98	0.0	1399.98
2023-12-14	trx-537548137090	85519	9	4	25	9	4499.91	0.15	3824.9235
2023-9-6	trx-000112360946	85490	null	null	2	6	4199.9400000000005	0.0	4199.9400000000005
2023-2-17	trx-065786773740	85518	9	3	9	8	1039.92	0.0	1039.92
2022-3-26	trx-111359626084	85493	8	1	4	5	499.95	0.1	449.955
2023-7-9	trx-179541605067	85490	2	5	20	6	359.94	0.05	341.943
2022-4-1	trx-197477932940	85538	7	5	2	10	6999.9	0.2	5599.92
2022-10-18	trx-334836612567	85479	null	null	6	2	99.98	0.15	84.983
2023-3-24	trx-569727423317	85501	null	null	29	10	399.90000000000003	0.0	399.90000000000003
2023-7-24	trx-616637928705	85545	null	null	9	6	779.94	0.05	740.943
2023-7-9	trx-716638564987	85550	4	2	26	5	999.95	0.0	999.95
2022-6-18	trx-746803513484	85529	null	null	26	8	1599.92	0.0	1599.92
2023-2-13	trx-901595305127	85506	null	null	1	3	2999.9700000000003	0.0	2999.9700000000003
2023-5-21	trx-964650810363	85551	10	2	19	7	209.92999999999998	0.25	157.4475
2023-10-25	trx-137783890813	85480	8	3	24	3	149.97	0.0	149.97
2023-12-2	trx-155004665264	85497	1	3	17	2	59.98	0.2	47.984
2022-12-13	trx-293630405793	85546	5	5	26	4	799.96	0.1	719.964
2022-3-17	trx-311063724315	85471	null	null	8	3	239.96999999999997	0.2	191.976

only showing top 20 rows

3- daily_dump_from_source.py:

This script processes the previous day's data files from HDFS, consolidates them, performs transformations, and writes the result back to HDFS and the local file system. It involves reading CSV files, handling duplicates, performing SQL operations, and moving data between HDFS and local storage.

Key Components:

1. **Spark Session Initialization:**
   - Initializes a Spark session with Hive support to handle large-scale data processing.
2. **Reading Previous Day's Data:**
   - Identifies and reads data files from the previous day's directory in HDFS.
   - Consolidates data into DataFrames for branches, agents, and transactions.
3. **Data Cleaning and Union:**
   - Drops unnecessary columns (source and logs) if they exist.
   - Ensures all DataFrames have the same schema before union operations.
4. **Removing Duplicates:**
   - Drops duplicate records in each DataFrame.
5. **Creating Temporary Views:**
   - Creates temporary views for agents and transactions to perform SQL operations.
6. **Executing SQL Query:**
   - Executes a SQL query to calculate the total units sold by each sales agent for each product.

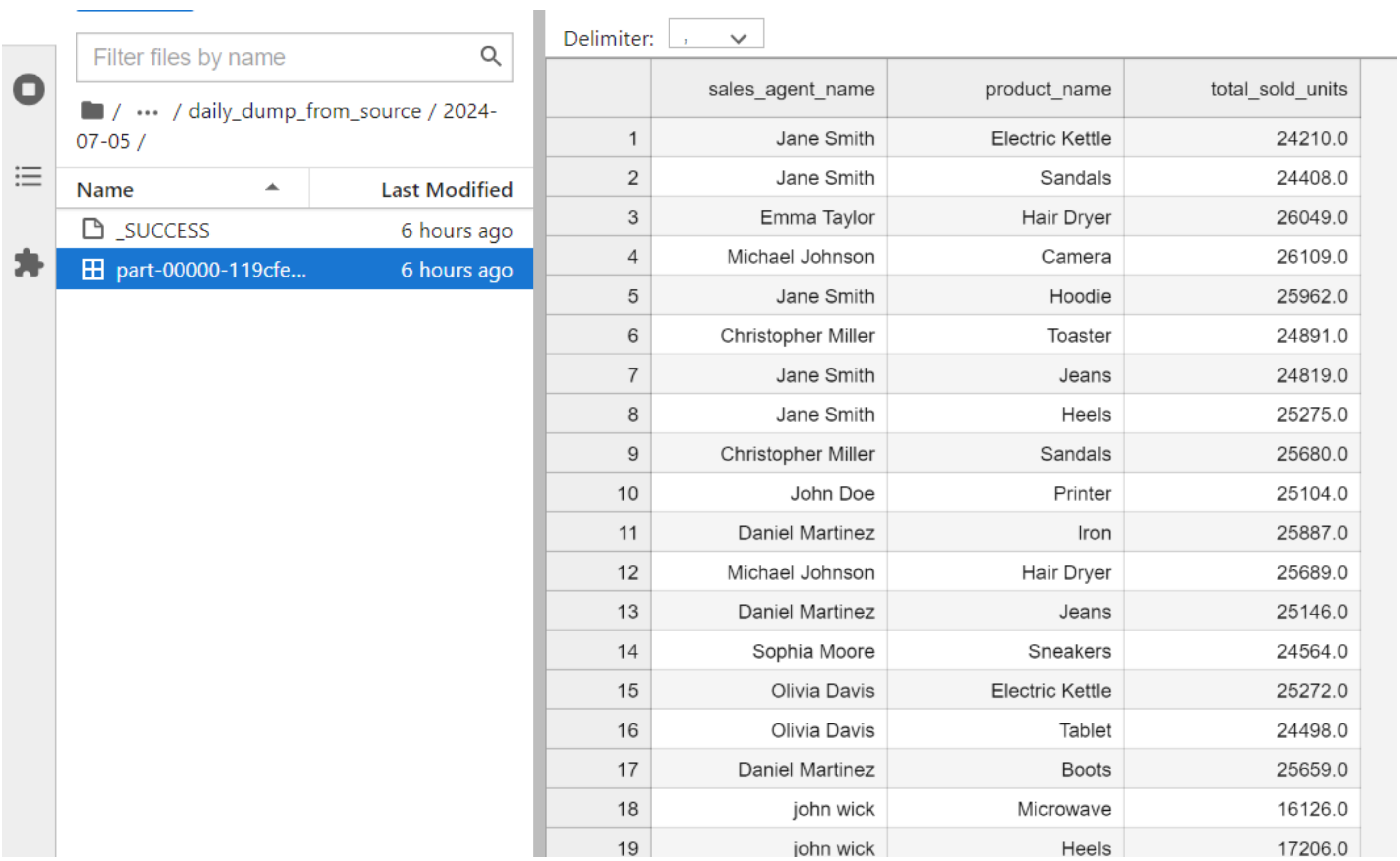
```
sql_query = """

   SELECT a.name AS sales_agent_name, t.product_name, SUM(t.units) AS total_sold_units

   FROM transactions_view t  JOIN agents_view a ON t.sales_agent_id = a.sales_person_id

   GROUP BY   a.name,  t.product_name, t.sales_agent_id,   t.product_id"""

# Execute the query and store the result in a DataFrame

result_df = spark.sql(sql_query)

distinct_df = result_df.distinct()
```

7. **Writing Results:**
   - Writes the resulting DataFrame to HDFS and then copies it to the local file system.

```
import subprocess

output_hdfs_path = "/user/itversity/daily_dump_from_source/2024-07-05"

local_output_path = "daily_dump_from_source"
```

```
try:

  # Copy the directory from HDFS to the local file system

  copy_command = ['hadoop', 'fs', '-get', '-f', output_hdfs_path, local_output_path]

  copy_result = subprocess.run(copy_command, stdout=subprocess.PIPE,
stderr=subprocess.PIPE, check=True)

  if copy_result.returncode == 0:

    print(f"Directory '{output_hdfs_path}' copied successfully to '{local_output_path}'.")

  else:

    print(f"Error: {copy_result.stderr.decode('utf-8')}")

except subprocess.CalledProcessError as e:

  print(f"Error: {e.stderr.decode('utf-8')}")
```

8. **Stopping Spark Session:**
    - Stops the Spark session to release resources.



	sales_agent_name	product_name	total_sold_units
1	Jane Smith	Electric Kettle	24210.0
2	Jane Smith	Sandals	24408.0
3	Emma Taylor	Hair Dryer	26049.0
4	Michael Johnson	Camera	26109.0
5	Jane Smith	Hoodie	25962.0
6	Christopher Miller	Toaster	24891.0
7	Jane Smith	Jeans	24819.0
8	Jane Smith	Heels	25275.0
9	Christopher Miller	Sandals	25680.0
10	John Doe	Printer	25104.0
11	Daniel Martinez	Iron	25887.0
12	Michael Johnson	Hair Dryer	25689.0
13	Daniel Martinez	Jeans	25146.0
14	Sophia Moore	Sneakers	24564.0
15	Olivia Davis	Electric Kettle	25272.0
16	Olivia Davis	Tablet	24498.0
17	Daniel Martinez	Boots	25659.0
18	john wick	Microwave	16126.0
19	john wick	Heels	17206.0

3- Business Requirements.ipynb:

Here's the notebook designed to execute the Spark jobs, connect to Hive, and answer business-related questions using Hive SQL queries. The steps include setting up the Spark session, executing queries, and displaying the results. This can be run as a standalone Jupyter notebook.

1. **Most Selling Products:**
   - **Query:** most_selling_products_query
   - **Description:** Retrieves the top-selling products based on total units sold.

```
Most Selling Products:
+----------+----------------+----------------+
|product_id|    product_name|total_units_sold|
+----------+----------------+----------------+
|        22|    Coffee Maker|          105984|
|        25| Washing Machine|           81380|
|         7|           Dress|           74812|
|        27|            Iron|           72098|
|        19|         Sandals|           71946|
|        17|          Blouse|           67716|
|         6|           Jeans|           66416|
|        28|      Hair Dryer|           66000|
|        29|Hair Straightener|           66000|
|        24|         Blender|           61152|
|         9|           Boots|           59508|
|        20|           Heels|           57200|
|         1|          Laptop|           55836|
|         5|         T-Shirt|           54880|
|         2|      Smartphone|           52440|
|        16|           Skirt|           52100|
|        14|          Camera|           50968|
|        26|  Vacuum Cleaner|           50904|
|        23|         Toaster|           50880|
|        11|              TV|           49980|
+----------+----------------+----------------+
```

2. **Most Redeemed Offers from Customers:**
   - **Query:** most_redeemed_offers_query
   - **Description:** Counts how many times each offer has been redeemed by customers.

```
Most Redeemed Offers from Customers:
+-------+-----------------+
|  offer|total_redemptions|
+-------+-----------------+
|offer_4|              336|
|offer_3|              324|
|offer_2|              302|
|offer_1|              276|
|offer_5|              268|
+-------+-----------------+
```

3. **Most Redeemed Offers per Product:**
   - **Query:** most_redeemed_offers_per_product_query
   - **Description:** Finds out which products have the highest number of offer redemptions, grouped by product and offer.

```
Most Redeemed Offers per Product:
+----------+-----------------+-------+-----------------+
|product_id|     product_name|  offer|total_redemptions|
+----------+-----------------+-------+-----------------+
|        22|     Coffee Maker|offer_3|             2880|
|        11|               TV|offer_4|             2856|
|        19|          Sandals|offer_4|             2772|
|        22|     Coffee Maker|offer_1|             2304|
|        29|Hair Straightener|offer_4|             2200|
|        28|       Hair Dryer|offer_1|             2160|
|         7|            Dress|offer_4|             2124|
|        27|             Iron|offer_4|             2124|
|         2|       Smartphone|offer_4|             2024|
|         5|          T-Shirt|offer_1|             2016|
|        22|     Coffee Maker|offer_2|             2016|
|         6|            Jeans|offer_1|             2016|
|        28|       Hair Dryer|offer_2|             1920|
|         7|            Dress|offer_5|             1888|
|        27|             Iron|offer_2|             1888|
|        25|  Washing Machine|offer_3|             1820|
|        25|  Washing Machine|offer_2|             1820|
|        25|  Washing Machine|offer_4|             1820|
|        20|            Heels|offer_3|             1800|
|         6|            Jeans|offer_3|             1792|
+----------+-----------------+-------+-----------------+
only showing top 20 rows
```

4. **Cities with the Lowest Online Sales:**

- **Query:** online_sales_summary_query
- **Description:** Lists cities with the lowest total online sales.

```
Cities with the lowest online sales:
+----------+------------------+
|      city|total_online_sales|
+----------+------------------+
|  Franklin|            59.985|
|  Falmouth| 75.96199999999999|
|  Redlands|            95.968|
| Edgewater|             99.98|
|   Fortuna|           101.966|
|    Dublin|           107.964|
|Youngstown|            119.94|
|  Freetown|            159.92|
|    Saugus|            159.96|
|   Greeley|227.92399999999998|
+----------+------------------+
```

Conclusion of Batch Processing

In this project, we successfully implemented a robust batch processing system for Q Company's retail data platform. Our primary goal was to ingest raw transactional data into a data lake, process it using Apache Spark, and store the transformed data in Hive for efficient querying and analysis.

Key Accomplishments:

1. **Data Ingestion:**
   - Successfully ingested raw transactional data into the data lake on a daily basis.
   - Leveraged HDFS to manage and organize the data efficiently.
2. **Data Processing with Spark:**
   - Utilized Spark for transforming and processing large volumes of transactional data.
   - Implemented various transformations including filtering, aggregating, and joining data from multiple sources.
3. **Hive Integration:**
   - Stored processed data in Hive tables, ensuring optimal organization and easy access.
   - Managed Hive partitions to improve query performance and maintainability.

4. **Scheduled Batch Jobs:**
    - Automated the batch processing tasks using cron jobs to ensure timely data processing and availability.
    - Implemented logging and monitoring to ensure reliability and troubleshoot issues promptly.
5. **Business Insights:**
    - Extracted valuable insights from the processed data, such as identifying the most selling products, analyzing offer redemptions, and pinpointing cities with the lowest online sales.
    - Enabled data-driven decision-making for various business operations.

Impact:

The batch processing pipeline significantly enhanced Q Company's ability to handle and analyze large volumes of transactional data. By automating the data ingestion and processing workflows, we ensured that up-to-date and accurate data is always available for business analysis and reporting. This improved operational efficiency and provided the business with actionable insights, helping to drive strategic decisions and optimize performance.

Streaming part

1-Producer:

Code: producer.py

How it works:

1. Configures Kafka producer with necessary security settings.
2. Randomly creates events mimicking customer actions (e.g., viewing a product, adding to cart).
3. Continuously sends these events to a Kafka topic (mark_topic).

Example of data:

Event produced: {'eventType': 'productView', 'customerId': '42943', 'productId': '1004', 'timestamp': '2024-07-06T06:55:04', 'metadata': {'category': 'Clothing', 'source': 'Advertisement'}}

2-Ingestion:

Code: Streaming.ipynb

Overview

This code is designed to read data from a Kafka topic, process it using Apache Spark, and write the processed data to HDFS in Parquet format.

First, the code specifies the Kafka connection details, including the bootstrap servers, topic name, and credentials (username and password). These details are used to establish a connection with the Kafka cluster.

Next, the Spark session is initialized with the application name "KafkaToParquet". This session serves as the entry point for using Spark functionalities.

The code then reads data from the specified Kafka topic as a streaming DataFrame. Various options are set to configure the Kafka source, such as the bootstrap servers, topic to subscribe to, starting offsets, and security settings for SASL_SSL authentication using the provided username and password.

Once the data is read from Kafka, it is parsed as JSON. The `selectExpr` method is used to cast the `value` column to a string, and the `from_json` function is used to parse this string into a DataFrame based on a predefined schema. The parsed data is then selected and flattened, with individual fields accessible as columns.

The code proceeds to further process the data by splitting the metadata column into separate `category` and `source` columns. It uses the `withColumn` method to create these new columns and drops the original metadata column.

Additionally, a new column `date` is added by extracting the date from the timestamp column. This is done using the `to_date` function, which converts the timestamp to a date format.

Finally, the processed DataFrame is written to HDFS in Parquet format, partitioned by the `date` column. The `writeStream` method is used to specify the output format, path, and checkpoint location, ensuring the streaming job's progress is tracked. The stream is started with the `start` method and awaits termination with the `awaitTermination` method, allowing the streaming job to run continuously.

1-schema:

```
# Define schema for the incoming JSON data
schema = StructType() \
  .add("eventType", StringType()) \
  .add("customerId", StringType()) \
  .add("productId", StringType()) \
  .add("timestamp", StringType()) \
  .add("metadata", MapType(StringType(), StringType())) \
  .add("quantity", IntegerType()) \
  .add("totalAmount", FloatType()) \
  .add("paymentMethod", StringType()) \
  .add("recommendedProductId", StringType())
```

2-**Parsing Schema Into DataFrame:**

```
# Parse the JSON data
json_df = df.selectExpr("CAST(value AS STRING)") \
      .select(from_json("value", schema).alias("data")) \
      .select("data.*")


# Split the metadata column into category and source
json_df2 = json_df.withColumn("category", col("metadata")["category"]) \
          .withColumn("source", col("metadata")["source"]) \
```

```
        .drop("metadata")

# Add a new column with the date extracted from timestamp
partitioned_df = json_df2.withColumn("date", to_date(col("timestamp")))
```

3- **Writing Data Into HDFS**

```
# Write the stream to HDFS partitioned by 'date'
query = partitioned_df \
    .writeStream \
    .partitionBy("date") \
    .format("parquet") \
    .option("path", outputDir) \
    .option("checkpointLocation", checkpointDir) \
    .start()
```

3-Reading Data With HIVE:

Code: Streaming.ipynb

Steps:

1- **Creating Hive Table**
2- **Reading Data With Hive**
3- **Querying Data**

First, a SQL-like string `create_table_query` is defined. This string specifies the schema for a new table, including columns for event type, customer ID, product ID, timestamp, quantity, total amount, payment method, recommended product ID, category, and source. The table is partitioned by the `date` column, which is of type DATE.

Next, the function `create_and_verify_table` is called with the table schema, table name (`"stream_output"`), schema name (`"BigData_DWH"`), and the output directory. This function is responsible for creating the table in the specified schema and verifying its existence.

The code then sets the schema name (`"BigData_DWH"`) and table name (`"stream_output"`) to be used in subsequent operations.

<u>1-Creating Hive Table</u>

```
create_table_query = """
(
    eventType STRING,
    customerId STRING,
    productId STRING,
    timestamp STRING,
    quantity INT,
    totalAmount FLOAT,
    paymentMethod STRING,
    recommendedProductId STRING,
    category STRING,
    source STRING
)
PARTITIONED BY (date DATE)
"""

# Call the function to create and verify the table
create_and_verify_table(create_table_query, "stream_output", "BigData_DWH", outputDir)
```

```
Success: Table 'BigData_DWH.stream_output' created successfully at hdfs://localhost:9000/user/itversity/stream_output.
```

<u>2- Reading Data With Hive</u>

To check if the Hive table contains any data, a query string `row_count_query` is defined, which counts the number of rows in the table. This query is executed using Spark's `sql` method, and the result is printed to show the row count.

```
schema_name="BigData_DWH"
table_name="stream_output"

# 1. Check if the Hive table has data
row_count_query = f"SELECT COUNT(*) AS row_count FROM {schema_name}.{table_name}"
row_count_result = spark.sql(row_count_query)
print("Row count in the table:")
row_count_result.show()
```

```
Row count in the table:
+---------+
|row_count|
+---------+
|      476|
+---------+
```

```
sample_data_query = f"SELECT * FROM {schema_name}.{table_name} LIMIT 5"
sample_data_result = spark.sql(sample_data_query)
print("Sample data from the table:")
sample_data_result.show()
```

```
Sample data from the table:
+-----------+----------+---------+-------------------+--------+-----------+-------------+--------------------+-------------+-----------+----------+
|  eventType|customerId|productId|          timestamp|quantity|totalAmount|paymentMethod|recommendedProductId|     category|     source|      date|
+-----------+----------+---------+-------------------+--------+-----------+-------------+--------------------+-------------+-----------+----------+
|  addToCart|     94660|     1479|2024-07-06T06:55:32|       1|       null|         null|                null|         null|       null|2024-07-06|
|productView|     48947|     1372|2024-07-06T06:55:33|    null|       null|         null|                null|Home & Kitchen|     Search|2024-07-06|
```

3-Querying Data

Query_1:

The first analytical query, `query1`, calculates the total sales amount and total quantity sold for each product. It groups the data by `productId` and orders the results by total sales amount in descending order. This query is executed, and the results are displayed.

```
query1 = """
SELECT
    productId,
    SUM(totalAmount) AS total_sales_amount,
    SUM(quantity) AS total_quantity_sold
FROM
    BigData_DWH.stream_output
GROUP BY
    productId
ORDER BY
    total_sales_amount DESC
LIMIT 10
"""

# Execute Query 1
result1 = spark.sql(query1)
result1.show()
```

```
+---------+------------------+-------------------+
|productId|total_sales_amount|total_quantity_sold|
+---------+------------------+-------------------+
|     1013| 943.1399841308594|                  4|
|     1322| 830.2200012207031|                  7|
|     1309| 635.3699951171875|                  4|
|     8059| 499.8999938964844|                  5|
|     8084| 495.6099853515625|                  5|
|     1370| 493.2699890136719|                  3|
|     3950| 488.6400146484375|                  4|
|     9783| 481.5799865722656|                  2|
|     8367| 477.7300109863281|                  2|
|     5907| 474.2200012207031|                  2|
+---------+------------------+-------------------+
```

Query_2:

The second analytical query, `query2`, calculates the daily sales amount and daily quantity sold, grouped by `date` and `paymentMethod`. The results are ordered by date and daily sales amount in descending order. This query is also executed, and the results are displayed.

```
# Query 2: Daily Sales and Quantity by Payment Method
query2 = """
SELECT
    date,
    paymentMethod,
    SUM(totalAmount) AS daily_sales_amount,
    SUM(quantity) AS daily_quantity_sold
FROM
    BigData_DWH.stream_output
WHERE
    paymentMethod IS NOT NULL
GROUP BY
    date, paymentMethod
ORDER BY
    date, daily_sales_amount DESC
"""

# Execute Query 2
result2 = spark.sql(query2)
result2.show()
```

```
+----------+-------------+------------------+-------------------+
|      date|paymentMethod|daily_sales_amount|daily_quantity_sold|
+----------+-------------+------------------+-------------------+
|2024-07-04|  Credit Card|3293.6600189208984|                 37|
|2024-07-04|   Debit Card|3099.7600326538086|                 26|
|2024-07-04|       PayPal| 2838.580047607422|                 42|
|2024-07-05|  Credit Card| 6837.699935913086|                 69|
|2024-07-05|       PayPal| 6566.679931640625|                 85|
|2024-07-05|   Debit Card| 4298.559982299805|                 64|
|2024-07-06|  Credit Card|1380.9299926757812|                 10|
|2024-07-06|   Debit Card| 879.9199752807617|                  9|
|2024-07-06|       PayPal| 534.0700073242188|                  9|
+----------+-------------+------------------+-------------------+
```