A UNIFIED DATA PLATFORM FOR Q COMPANY'S RETAIL OPERATIONS Documentation

By

Malak Sherif

Marina Fawzy

Mariam Maged

Ahmed Farid

ITI

06/07/24

INTRODUCTION

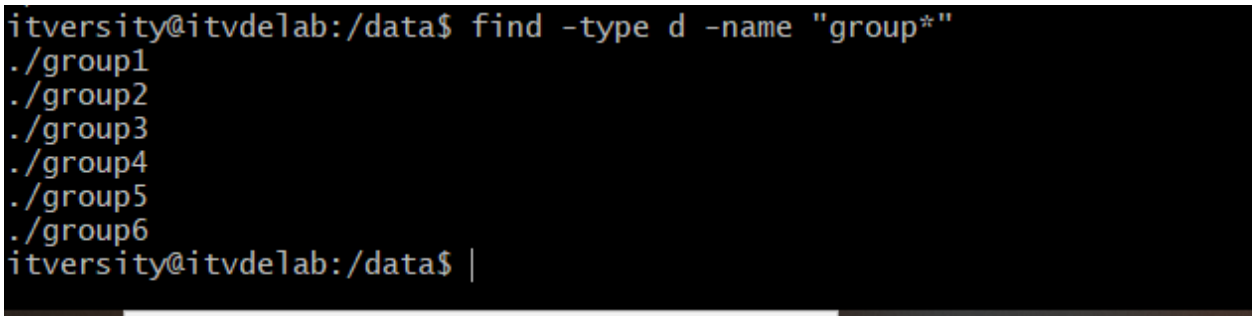
- **Company Overview:** Q company is a retail business with both physical branches and an e-commerce platform. They face challenges integrating data from various sources, including hourly updates on branches, sales agents, and sales transactions, as well as app logs streamed to a Kafka cluster. This disparate data landscape makes it difficult to derive timely insights for marketing and B2B initiatives.
- **Project Goals:**
    - Efficiently ingest raw sales data from multiple sources.
    - Transform and load data into a structured data warehouse (Hive).
    - Enable analysis to support marketing and B2B teams.
    - Process app logs from a Kafka cluster using a Spark streaming job and store the processed data on HDFS.

2. Architecture Diagram

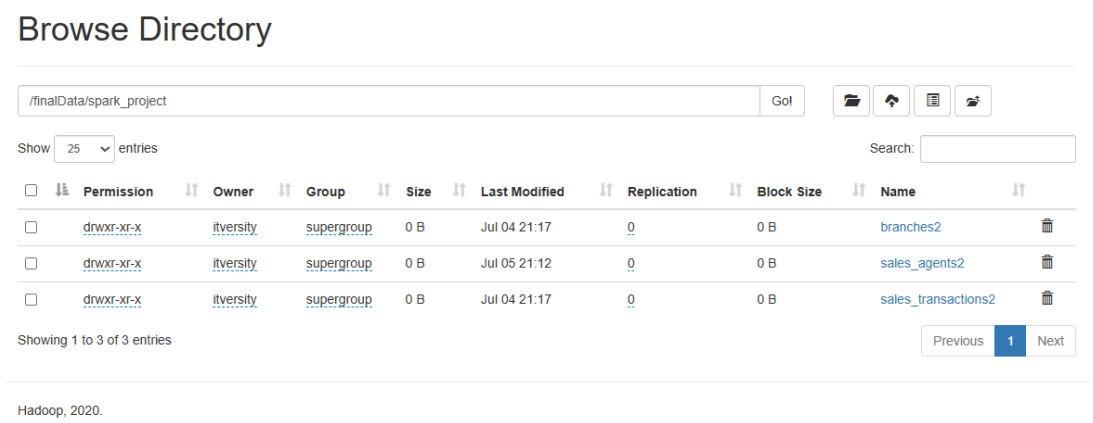
- **Local File System:** Where raw CSV files are initially placed.
- **HDFS (Hadoop Distributed File System):** The scalable storage layer for raw and processed data.
- **Spark:** The processing engine for data transformation and loading.
- **Hive:** The data warehouse for structured storage and querying.
- **Cron:** The scheduler for automating batch jobs.
- **Kafka :** The streaming platform used to handle real-time data feeds

The business send us 1 group every hour ,this group arrive in our local file system “on our linux machine “ , the groups will be stored at our “/data” directory

Inside every group we have 3 files: trancations , agents, branches



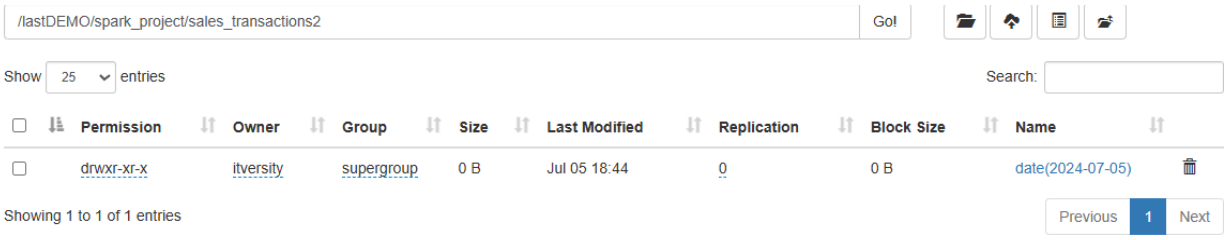
We will upload those groups to hdfs , using a python script , and we will create directory for each file meaning
1 directory for transaction
1 directory for branches
1 directory for agents



Our python script will not only upload the data to HDFS every hour through crontab job, But will also help ensure that your data is organized and accessible for further analysis or processing in a Hadoop environment.

How It Works:

1. **Define Local and HDFS Directories:**
    - It begins by setting up local directories likr (group1) where your files are stored.
    - It also defines where these files will be transferred to in the HDFS, creating directories based on the current date.

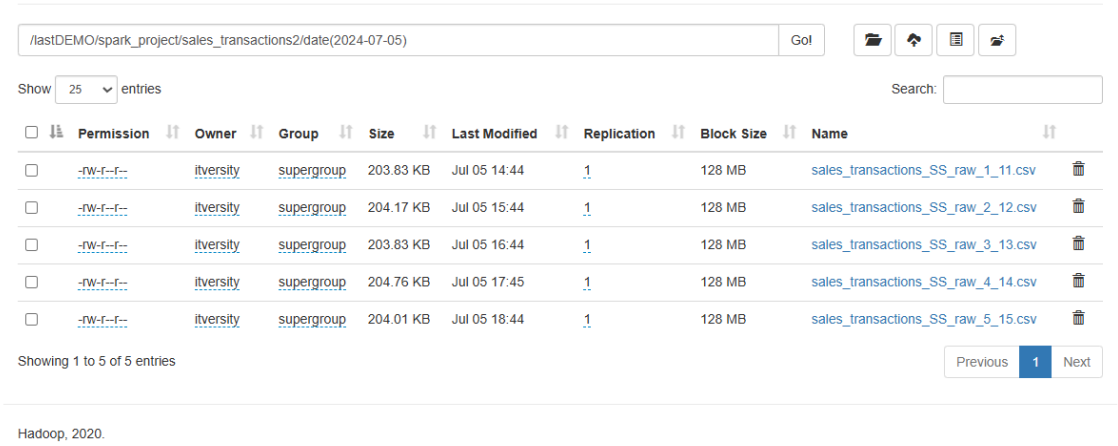


2. **Track Which Directory to Process:**
    - The script keeps track of which local directory to process next using a state_file. This file remembers the last processed directory, so the script knows which one to handle next in a cyclical manner.
3. **Identify and Create HDFS Destinations:**
    - For each file in the current local directory, it identifies the appropriate destination directory in HDFS based on predefined rules. For example, files starting with sales_transactions will go to a specific directory named with the current date.
4. **Move Files:**

    - It checks if the destination directory in HDFS exists, creating it if necessary.
    - The script then moves each file from the local directory to its corresponding location in HDFS, appending the current hour to the file name to prevent overwriting files and to keep track of when they were moved.
5. **Update State for Next Run:**
    - After moving the files, the script updates the state_file to the next directory index, preparing for the next scheduled run.
6. **Schedule Regular Runs:**
    - This script can be scheduled to run at regular intervals (e.g., every hour) using a scheduling tool like cron, ensuring continuous and automated file transfers.

Sposed that STATE FILE contain 5 , this means that we have uploaded file number 5 on HDFS

```
itversity@itvdelab:/data/movedata_test$ head state_file.txt
5
```



/lastDEMO/spark_project/sales_transactions2/date(2024-07-05) Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	itversity	supergroup	203.83 KB	Jul 05 14:44	1	128 MB	sales_transactions_SS_raw_1_11.csv
-rw-r--r--	itversity	supergroup	204.17 KB	Jul 05 15:44	1	128 MB	sales_transactions_SS_raw_2_12.csv
-rw-r--r--	itversity	supergroup	203.83 KB	Jul 05 16:44	1	128 MB	sales_transactions_SS_raw_3_13.csv
-rw-r--r--	itversity	supergroup	204.76 KB	Jul 05 17:45	1	128 MB	sales_transactions_SS_raw_4_14.csv
-rw-r--r--	itversity	supergroup	204.01 KB	Jul 05 18:44	1	128 MB	sales_transactions_SS_raw_5_15.csv

Showing 1 to 5 of 5 entries Previous 1 Next

Hadoop, 2020.

Also 11 means that the file was uploaded at 11AM and 15 means that the file was uploaded at 3AM

Why This Is Beneficial:

- **Automation:** You don't need to manually move files to HDFS, saving time and reducing the risk of errors.
- **Organization:** Files are organized into date-specific directories, making it easier to manage and locate data.
- **Consistency:** The script ensures that files are moved systematically and consistently, adhering to a set schedule.
- **Scalability:** As the amount of data grows, the script can handle large volumes and multiple directories efficiently.

Archiving Process for Uploaded Groups

1. **Archiving to the "Archive" Folder**:
    - After each group (folder) is successfully uploaded to the Data Lake, it will be moved to an "Archive" folder for safekeeping.
2. **Data Retention Policy**:
    - The "Archived" folder will be subject to a specific data retention policy, retaining the archived data for one month. This ensures that data is available for disaster recovery purposes during this period.

How Files Are Read and Processed:SPARK JOP1

Once the files are moved to HDFS (Hadoop Distributed File System) by the Python script, the Spark job takes over to read and process these files. Here's how this process works:

This data processing workflow for handling sales transactions data stored in hour- and day-based directories. The goal of this process is to clean, transform, and analyze the sales data to extract valuable business insights.

1. Setting Up the Spark Environment

To start, we initialize the Spark environment with the right configurations:

2. Reading Data from Hour and Day Directories

Data is stored in a structured directory format based on the date and hour of the transaction. We retrieve the data for the last hour using a function that computes the current date and hour

get_last_hour(): Gets the date and hour for the last hour.

directory_path1: Constructs the full path to the CSV files for the last hour's transactions.

3. Data Cleansing and Transformation

Handling Missing Values

We calculate the number of null values in each column and fill them with default values where applicable

- **null_counts**: Computes the count of null values per column.
- **fill_values**: Specifies default values for columns with missing data.
- **data.na.fill()**: Fills the null values in the DataFrame with specified defaults.

When handling nulls we used smart logic to check if the columns exit first before removing the nulls

Removing Duplicates

We identify and remove duplicate rows in the dataset

- **dropDuplicates()**: Removes duplicate rows from the DataFrame.

Calculating Discounts

We add a new column discounts based on the available offers

Consolidating Offers

We consolidate individual offer columns into a single offer column and drop the original columns:

```
df = data.withColumn('offer',

   when(col('offer_1') == 'True', 1)

   .when(col('offer_2') == 'True', 2)

   .when(col('offer_3') == 'True', 3)

   .when(col('offer_4') == 'True', 4)

   .when(col('offer_5') == 'True', 5)

   .otherwise(None))

data = df.drop(*['offer_1', 'offer_2', 'offer_3', 'offer_4', 'offer_5'])
```

Splitting Shipping Address

We split the shipping_address into city, state, and postal_code:

Calculating Total Paid

```
data = data.withColumn(

   "total_paid",
```

```
    (col("units") * col("unit_price")) - (col("units") * col("unit_price") * col("discounts"))
)
```

Correcting Email Format

```
from pyspark.sql.functions import regexp_replace

data = data.withColumn("cusomter_email", regexp_replace("cusomter_email", r"\.com.*", ".com"))
```

Adding Date Columns

We convert the transaction_date to a standard format and add extraction date columns:

- to_date(): Converts the transaction_date to a date format.
- current_date(): Adds the current extraction date.
- year(), month(): Extracts the year and month from the extraction date.

4. Creating Hive Tables

We create Hive tables to store the transformed data for further analysis

Creating Dimension Tables

We create dimension tables for customers, products, locations, and dates

```
customer_dim_df = data.select("customer_id", "customer_fname", "cusomter_lname", "cusomter_email").distinct()

product_dim_df = data.select("product_id", "product_name", "product_category").distinct()

sales_data_with_location = data.select(concat(col("city"), lit("-"), col("state"), lit("-"),
col("postal_code")).alias("location_id"), "city", "state", "postal_code").distinct()

date_dim_df = data.select("transaction_date", dayofmonth("transaction_date").alias("day"),
month("transaction_date").alias("month"), year("transaction_date").alias("year"),
quarter("transaction_date").alias("quarter"), date_format("transaction_date", "EEEE").alias("day_name")).distinct()

customer_dim_df.write.format("orc").mode("append").saveAsTable("TransactionsDB.Customer1_Dim")

product_dim_df.write.format("orc").mode("append").saveAsTable("TransactionsDB.Product1_Dim")

sales_data_with_location.write.format("orc").mode("append").saveAsTable("TransactionsDB.Location1_Dim")

date_dim_df.write.format("orc").mode("append").saveAsTable("TransactionsDB.Date1_Dim")
```

we also have to more dimensional table the **agent dimention** and the **branches dimentions**

```
try:
    # Save the DataFrame as a Hive table with multi-level partitioning and bucketing
    agents.write.format("orc") \
        .mode("overwrite") \
        .saveAsTable("TransactionsDB.AgentDIM")

    print("Table created successfully.")
except Exception as e:
    print(f"Error creating table: {e}")
```

```
Table created successfully.
```

```
try:
    # Save the DataFrame as a Hive table with multi-level partitioning and bucketing
    branches.write.format("orc") \
        .mode("overwrite") \
        .saveAsTable("TransactionsDB.branchsDIM")

    print("Table created successfully.")
except Exception as e:
    print(f"Error creating table: {e}")
```

```
Table created successfully.
```

Creating Fact Table

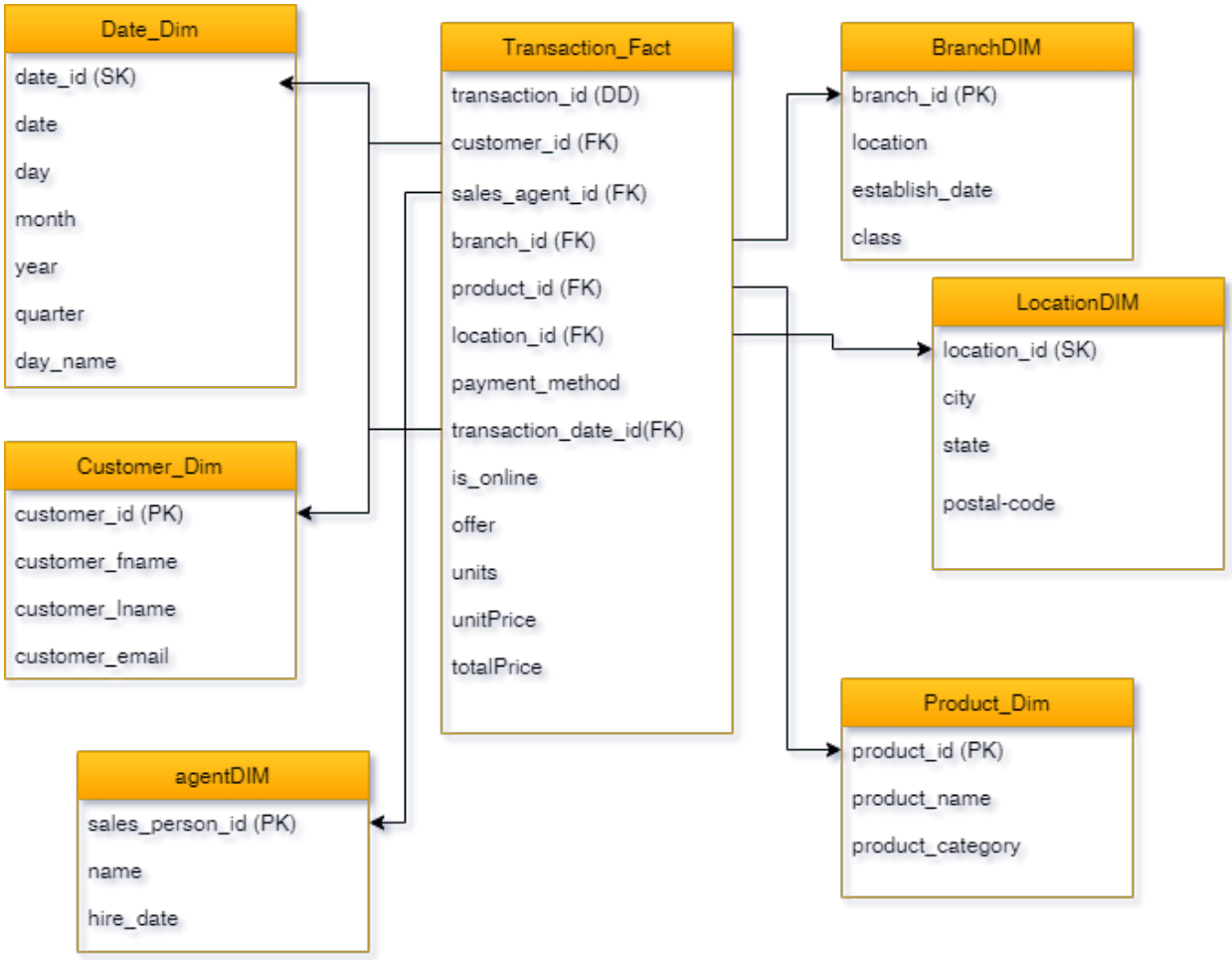
We create a fact table that stores transactional data, Saves the fact DataFrame as a Hive table with partitioning.

```
fact_df = data.select(
    "transaction_id", "customer_id", "sales_agent_id", "branch_id", "product_id",
    F.concat_ws("-", "city", "state", "postal_code").alias("location_id"),
    "transaction_date", "offer", "is_online", F.col("units").cast("int").alias("units"),
    F.col("unit_price").cast("double").alias("unit_price"), "total_paid",
    "Extraction_Year", "Extraction_Month"
)

fact_df.write.format("orc") \
    .partitionBy("Extraction_Year", "Extraction_Month") \
    .mode("append") \
    .saveAsTable("TransactionsDB.Transaction_FactTable")
```

Hive DWH model

offers a solid foundation for effective data storage, retrieval, and analysis. Its design promotes high performance



Key Strengths of the Hive DWH Model

1. **Clear Dimensional Modeling**:
    - **Fact Table (Transaction_Fact)**: Central to capturing sales metrics.
    - **Dimension Tables**: Provide contextual details (Date, Customer, Product, Location, Branch, Agent) for deeper insights.
2. **Separation of Concerns**:
    - Efficiently manages large volumes of numeric data (facts) separately from descriptive data (dimensions).
    - Simplifies updates and enhances usability for generating reports and insights.
3. **Performance Optimization**:
    - Uses smaller dimension tables to filter and aggregate data in the large fact table.
    - Supports efficient querying, leveraging Hive's indexing and partitioning.
4. **Comprehensive Business Analysis**:

- **Time Analysis**: The Date_Dim allows for time-based analysis like daily, monthly, and yearly trends. It supports seasonal, quarterly, and day-of-week analyses.
- **Customer Insights**: The Customer_Dim provides details about customer demographics and behavior, which are crucial for segmentation and personalized marketing.

- **Product Performance**: The Product_Dim helps analyze product sales and performance, aiding in inventory management and product strategy.
- **Geographical Insights**: The LocationDIM enables geographical analysis, which is essential for regional performance and market expansion strategies.
- **Sales Operations**: The BranchDIM and agentDIM provide insights into sales operations and agent performance, supporting resource allocation and branch management.

5. **Scalability and Flexibility**:
    - Easily extends to accommodate new data or business requirements.
    - Scales efficiently with growing data volumes, benefiting from Hive's robust data management.
6. **Data Consistency and Integrity**:
    - Ensures transactions link to valid dimension entries, maintaining data accuracy and reliability.
7. **Support for Complex Queries**:
    - Facilitates comprehensive reports and in-depth business insights by joining fact and dimension tables.
8. **Advanced Data Management**:
    - **Partitioning and Bucketing**: Enhances performance by organizing and retrieving data efficiently.
    - **Data Quality**: Manages data consistency and handles issues like missing values and duplicates.

Business Requirements

1.Most Selling Products:

Business Meaning: This query identifies the top 10 best-selling products based on the number of units sold. Understanding which products are the most popular helps in making informed decisions about inventory management, marketing strategies, and product development. Popular products can be promoted more aggressively, and inventory levels can be adjusted to ensure availability.

```
#total sold unites by product

most_selling_products_df = spark.sql("""

SELECT p.product_name, SUM(f.units) AS total_sold_units

FROM TransactionsDB.Transaction_FactTable f

JOIN TransactionsDB.Product1_Dim p ON f.product_id = p.product_id

GROUP BY p.product_name

ORDER BY total_sold_units DESC

LIMIT 10

""")


most_selling_products_df.show()
```

```
+-----------------+----------------+
|     product_name|total_sold_units|
+-----------------+----------------+
|            Boots|          204041|
|          Sandals|          203997|
|           Laptop|          103629|
|       Smartphone|          103275|
|Hair Straightener|          103257|
|            Skirt|          103004|
|          Toaster|          102779|
|        Microwave|          102702|
|          T-Shirt|          102469|
|            Dress|          102253|
+-----------------+----------------+
```

2. The most redeemed offers from the transaction per product

Identifying Most Popular Offers: This query helps the business understand which promotional offers.

```
spark.sql("""
    WITH offer_details AS (
        SELECT product_id,offer,COUNT(*) AS total_redeemed FROM TransactionsDB.Transaction_FactTable
        WHERE offer IS NOT NULL GROUP BY product_id, offer ),
    ranked_offers AS (
        SELECT  product_id,offer,total_redeemed,
            ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY total_redeemed DESC) AS rank
        FROM offer_details)
    SELECT  p.product_name,
        CASE
            WHEN ro.offer = 1 THEN 'offer_1'
            WHEN ro.offer = 2 THEN 'offer_2'
            WHEN ro.offer = 3 THEN 'offer_3'
            WHEN ro.offer = 4 THEN 'offer_4'
            WHEN ro.offer = 5 THEN 'offer_5'
            ELSE 'other_offer'
        END AS most_redeemed_offer,
        ro.total_redeemed
    FROM ranked_offers ro
    JOIN TransactionsDB.Product1_Dim p ON ro.product_id = p.product_id
    WHERE ro.rank = 1
    ORDER BY ro.total_redeemed DESC
""")
```

83]:

product_name	most_redeemed_offer	total_redeemed
Hair Dryer	offer_1	1981
Skirt	offer_3	1958
Toaster	offer_1	1941
T-Shirt	offer_2	1937
Laptop	offer_5	1933

3.Lowest Cities in Online Sales

Business Meaning: This query identifies the cities with the lowest online sales. By understanding which regions are underperforming, the marketing team can target these areas with specific campaigns to boost sales. This strategy can help increase market penetration and overall revenue by addressing and improving weak sales regions.

```
lowest_online_sales_cities_df = spark.sql("""

SELECT l.city, round(SUM(f.total_paid),2) AS online_sales

FROM  TransactionsDB.Transaction_FactTable f

JOIN TransactionsDB.Location1_Dim l ON f.location_id = l.location_id

WHERE f.is_online = 'yes'

GROUP BY l.city

ORDER BY online_sales ASC

LIMIT 10

""")



lowest_online_sales_cities_df.show()
```

```
|         city|online_sales|
+-------------+------------+
|        Linda|     2189.48|
|      Fairfax|     3328.56|
| Tyngsborough|     3717.47|
|     Oakhurst|     3802.57|
|Nichols Hills|     3887.52|
|       Palmer|     4116.54|
|  Wheat Ridge|      4287.4|
|     Redlands|     4485.43|
| West Windsor|     4533.49|
|  North Adams|     4580.31|
+-------------+------------+
```

Batch Job2

We have another daily Spark job is designed to aggregate sales data by integrating information from agents and transactions. This process involves:

1. **Data Retrieval**: The job reads data from the agents' details and transaction records.
2. **Data Integration**: It joins these datasets to combine agent information with their respective sales transactions.
3. **Insights Generation**: The output highlights the agent's name, the products sold, and the total units sold for each product.
4. **Local Storage**: The results are then written to a local CSV file, ensuring easy access and further analysis.

Business Value: This operation is crucial for tracking sales performance at the agent level. By understanding which agents are driving sales for specific products, we can tailor business strategies and conduct thorough performance assessments.

Output Format:

- **Agent Name**: Identifies the sales agent.
- **Product Name**: Specifies the product sold.
- **Total Sold Units**: Aggregates the total number of units sold per product by each agent.

Storage Method: The compiled data is saved as a CSV file locally, providing a straightforward format for reviewing and integrating with other analytical tools or systems.

Project Documentation: Kafka-Spark Streaming Data Pipeline

1. Introduction

This document provides a detailed description of the implementation of a data pipeline using Apache Kafka and Apache Spark. The goal is to capture application logs, process them in real-time, and store the processed data on HDFS. The logs have a dynamic schema, requiring careful handling and processing

2. Technical Description

2.1 Kafka Producer

A Python-based Kafka producer is used to send application logs to the Kafka cluster. Given that the logs have a dynamic schema, it is essential to examine the producer code to identify all possible data elements that will be sent to Kafka.

Producer Configuration and Execution:

Script Path: /script/location/script.py
Command to Start Producer:

python /script/location/script.py

Running from Jupyter Notebook: The producer can also be initiated from a Jupyter notebook environment, providing flexibility for interactive development and testing.

2.2 Kafka Configuration

Ensure that the Kafka topic is properly configured to receive the logs. The topic name, partitioning, and replication factors should be defined based on the expected load and reliability requirements.

2.3 Spark Streaming Job

Create a Spark Streaming job to receive data from Kafka, process it, and store the results in HDFS.

Steps:

1. **Read Data from Kafka:** Use Spark's Kafka integration to read the stream of logs.
2. **Process Data:** Perform necessary transformations and aggregations on the streaming data.

    # Define schema for the incoming JSON data

```
# Define schema based on the producer's data generating part

schema = StructType([

    StructField("eventType", StringType(), True),

    StructField("customerId", StringType(), True),

    StructField("productId", StringType(), True),

    StructField("timestamp", StringType(), True),

    StructField("quantity", IntegerType(), True),

    StructField("totalAmount", DoubleType(), True),

    StructField("paymentMethod", StringType(), True),

    StructField("recommendedProductId", StringType(), True),

    StructField("algorithm", StringType(), True),

    StructField("metadata", StructType([

        StructField("category", StringType(), True),

        StructField("source", StringType(), True)

    ]), True)

])
```

3. **Store Data on HDFS:** Write the processed data to HDFS in a structured format (e.g., Parquet, ORC).

3. Business Requirements

3.1 Data Storage and Organization

As a representative of the business team, it is crucial to read and understand the data carefully. Store the data in HDFS in a manner that maximizes its value for various teams.

3.2 Queries and Reports

To derive value from the stored data, at least two queries or reports should be created:

Query 1: Average Total Amount by Payment Method

Generate a report showing the average total transaction amount for each payment method.

Business Meaning: This report helps the business understand the spending patterns associated with different payment methods. It can inform decisions about payment processing partnerships, promotional strategies, and customer preferences.

```
from pyspark.sql.functions import avg



average_total_amount = data.groupBy("paymentMethod").agg(avg("totalAmount"))

average_total_amount.show()
```

+-------------+------------------+

|paymentMethod| avg(totalAmount)|

+-------------+------------------+

```
|  Credit Card|            145.29|
|      Unknown|               0.0|
|       PayPal|            436.84|
|   Debit Card|320.97333333333336|
+-------------+------------------+
```

Query 2: Total Events by Type

Generate a report showing the total count of each type of event.

Business Meaning: This report provides insights into the frequency and distribution of various events within the application. It can help in identifying the most common user actions, monitoring application health, and prioritizing feature enhancements or bug fixes.

```
total_events_by_type = data.groupBy("eventType").count()

total_events_by_type.show()
```

```
+-------------------+-----+
|          eventType|count|
+-------------------+-----+
|           purchase|    2|
|          addToCart|    9|
|        productView|    7|
|recommendationClick|   11|
+-------------------+-----+
```