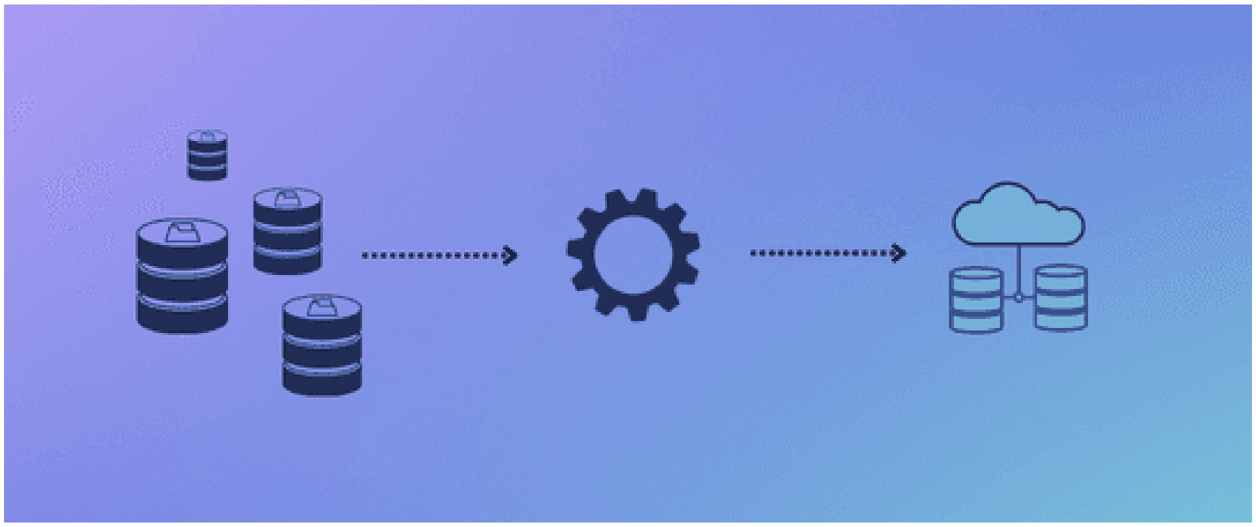


Real-Time E-commerce Data Pipeline with Spark ETL Documentation

Project Overview

This project aims to design and implement an ETL pipeline for ShopEase, a hypothetical e-commerce platform. The pipeline will handle large-scale data processing in batch and real-time modes using Apache Spark. The data includes user interactions, transactions, inventory updates, and customer feedback

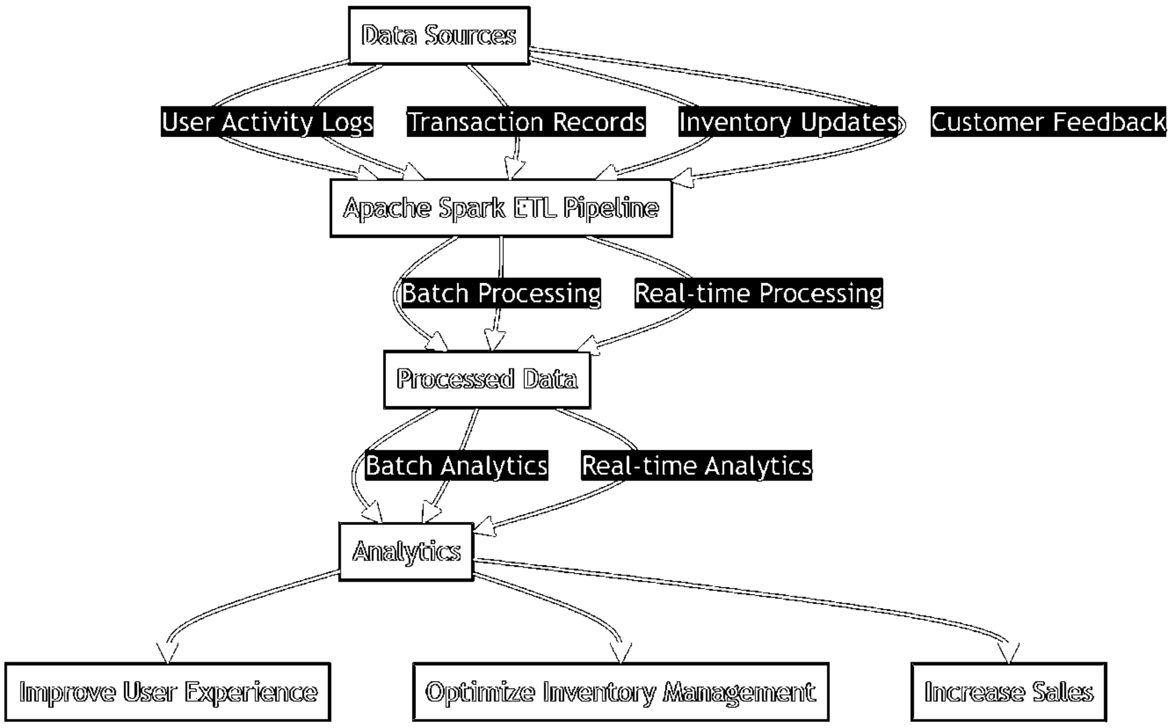
Scenario

Shop Ease wants to enhance its analytics capabilities to improve user experience, optimize inventory management, and increase sales. The data pipeline will process the following types of data:

- **User Activity Logs**: Captures clickstream data representing user interactions.
- **Transaction Records**: Details about purchases, refunds, and returns.

- **Inventory Updates**: Information about stock levels, restocks, and product changes.
- **Customer Feedback**: Reviews and ratings from customers.

The goal is to create a robust ETL pipeline to process this data, perform necessary transformations, and make it available for both batch and real-time analytics.



Project Requirements

The ETL pipeline consists of the following components:

1. **Data Ingestion**
2. **Data Processing and Transformation**
3. **Real-Time Streaming Processing (Optional)**
4. **Data Storage**
5. **Performance Optimization**
6. Documentation and Reporting

all will be shown below

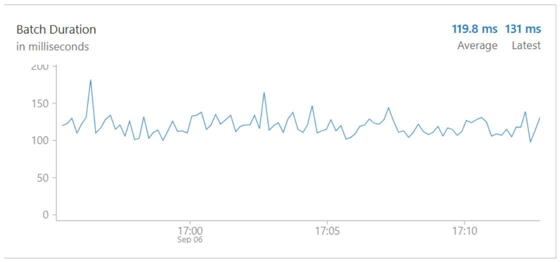
1. Data Ingestion

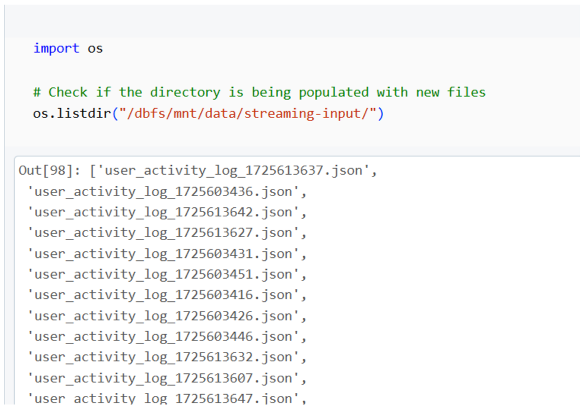
- **Q1: How did you ingest the batch and real-time data? Provide code snippets demonstrating the loading of data using RDDs and Data Frames.**

Batch Data Ingestion:

- Loaded large historical datasets (CSV and JSON) from a local file system into Apache Spark Data Frames.
- Datasets included transaction records, inventory data, and customer feedback.
- **Used `spark.read.csv()` and `spark.read.json()` methods.**

💡 **Generate a batch of 10 random user activity logs**





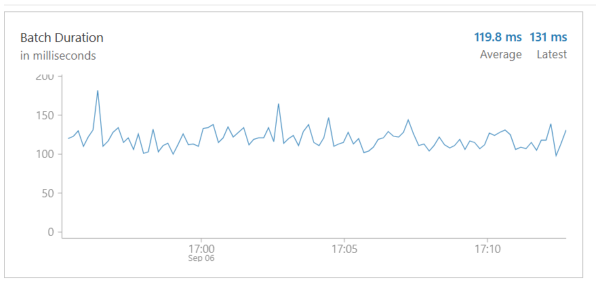
```
import os

# Check if the directory is being populated with new files
os.listdir("/dbfs/mnt/data/streaming-input/")
```

```
Out[98]: ['user_activity_log_1725613637.json',
 'user_activity_log_1725603436.json',
 'user_activity_log_1725613642.json',
 'user_activity_log_1725613627.json',
 'user_activity_log_1725603431.json',
 'user_activity_log_1725603451.json',
 'user_activity_log_1725603416.json',
 'user_activity_log_1725603426.json',
 'user_activity_log_1725603446.json',
 'user_activity_log_1725613632.json',
 'user_activity_log_1725613607.json',
 'user_activity_log_1725613647.json',
```

Real-Time Data Ingestion:

- Simulated live user activity logs using spark
- Data included user activity logs like clickstream data capturing interactions the Shop Ease platform.



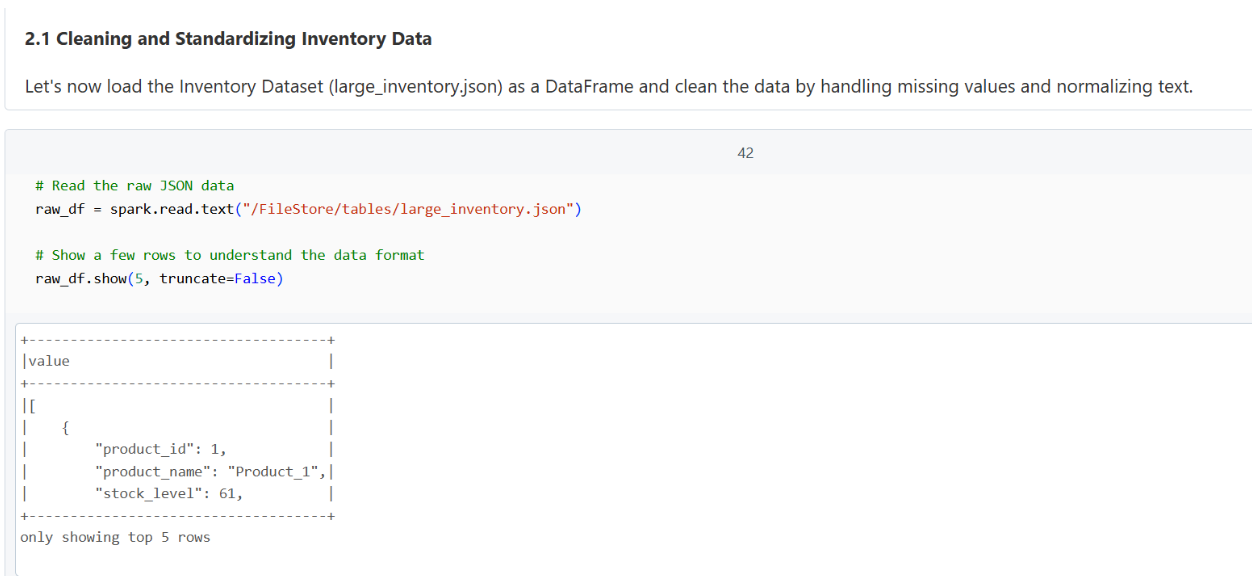
💡 **Ingest Real-Time Data Using Spark Structured Streaming**

2. Data Cleaning and Transformation

- **Q2: Describe the transformations applied to the transaction data using RDDs. How did you ensure data quality and privacy?**

RDD Transformations:

- **Filtered Corrupted Records**: Removed records missing `transaction_id` or `amount`.
- **Anonymized User IDs**: Used a hashing function to replace user IDs with hashed values.
- **Ensured Data Quality**: Removed invalid/incomplete records and handled exceptions during processing.


2.1 Cleaning and Standardizing Inventory Data

Let's now load the Inventory Dataset (large_inventory.json) as a DataFrame and clean the data by handling missing values and normalizing text.

42

```python
# Read the raw JSON data
raw_df = spark.read.text("/FileStore/tables/large_inventory.json")

# Show a few rows to understand the data format
raw_df.show(5, truncate=False)
```

```
+------------------------------------+
|value                               |
+------------------------------------+
|[                                   |
|    {                               |
|        "product_id": 1,            |
|        "product_name": "Product_1",|
|        "stock_level": 61,          |
+------------------------------------+
only showing top 5 rows
```

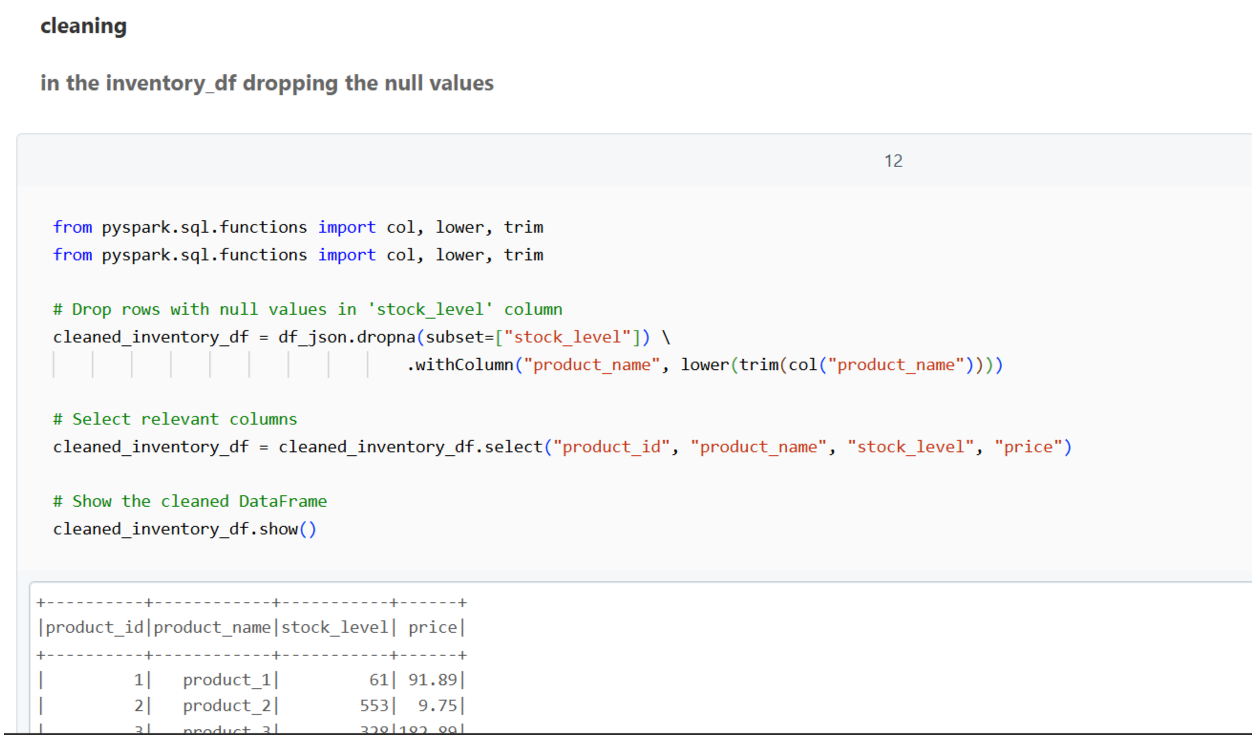
Data Frame Operations

Q3: How did you clean and standardize the inventory data using Data Frames? Provide examples of handling missing values and normalizing text fields.

Inventory Data Cleaning:

- **Handled Missing Values**: Used `DataFrame.dropna()` to remove records with missing critical values.
- **Normalized Text Fields**: Applied `lower()` and `trim()` to standardize text fields.

The cleaned Data Frame was then ready for further analysis.



cleaning

in the inventory_df dropping the null values

12

```python
from pyspark.sql.functions import col, lower, trim
from pyspark.sql.functions import col, lower, trim

# Drop rows with null values in 'stock_level' column
cleaned_inventory_df = df_json.dropna(subset=["stock_level"]) \
                                .withColumn("product_name", lower(trim(col("product_name"))))

# Select relevant columns
cleaned_inventory_df = cleaned_inventory_df.select("product_id", "product_name", "stock_level", "price")

# Show the cleaned DataFrame
cleaned_inventory_df.show()
```

```
+----------+------------+-----------+------+
|product_id|product_name|stock_level| price|
+----------+------------+-----------+------+
|         1|   product_1|         61| 91.89|
|         2|   product_2|        553|  9.75|
|         3|   product_3|        328|182.99|
```

Spark SQL Queries

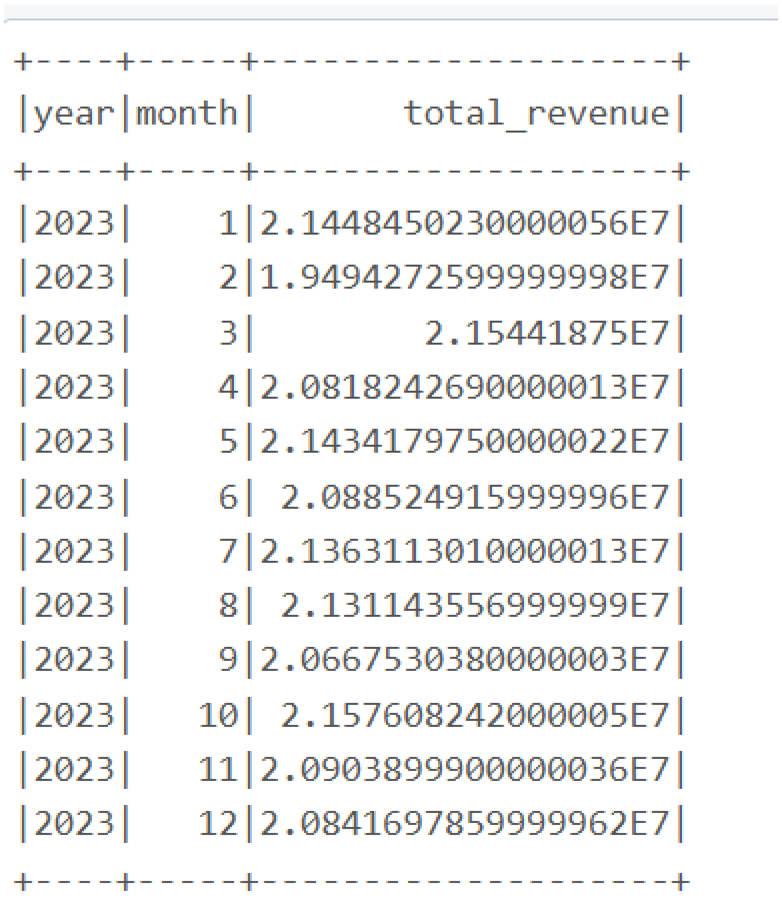
Q4: Present the Spark SQL queries used to calculate the top 10 most purchased products, monthly revenue trends, and inventory turnover rates.

Top 10 Most Purchased Products:

```
spark.sql("""
    SELECT product_id, COUNT(*) AS purchase_count
    FROM transactions
    WHERE transaction_date >= DATE_SUB(CURRENT_DATE(), 30)  -- L
    GROUP BY product_id
    ORDER BY purchase_count DESC
    LIMIT 10
""").show()
```

Monthly Revenue Trends:

```
spark.sql("""
    SELECT
        YEAR(transaction_date) AS year,
        MONTH(transaction_date) AS month,
        SUM(amount) AS total_revenue
    FROM transactions
    GROUP BY YEAR(transaction_date), MONTH(transaction_date)
    ORDER BY year, month
""").show()
```



```
+----+-----+--------------------+
|year|month|       total_revenue|
+----+-----+--------------------+
|2023|    1|2.1448450230000056E7|
|2023|    2|1.9494272599999998E7|
|2023|    3|         2.15441875E7|
|2023|    4|2.0818242690000013E7|
|2023|    5|2.1434179750000022E7|
|2023|    6| 2.088524915999996E7|
|2023|    7|2.1363113010000013E7|
|2023|    8| 2.131143556999999E7|
|2023|    9|2.0667530380000003E7|
|2023|   10| 2.157608242000005E7|
|2023|   11|2.0903899900000036E7|
|2023|   12|2.0841697859999962E7|
+----+-----+--------------------+
```

Inventory Turnover Rates:

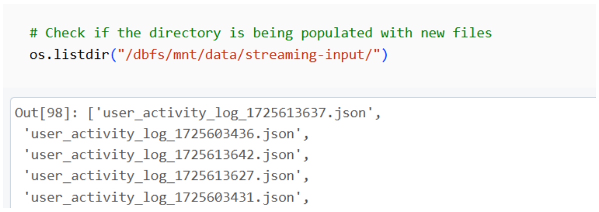
```
SELECT product_name, (SUM(quantity) / MAX(stock_level)) AS turno
FROM transactions
JOIN inventory ON transactions.product_id = inventory.product_i
GROUP BY product_name;
```

Real-Time Processing

Q5: If implemented, explain how the real-time streaming was set up. What metrics were computed in real-time, and how were they stored/displayed?

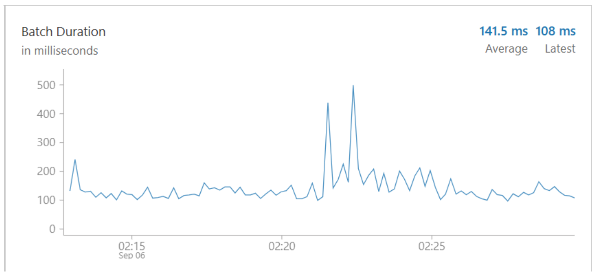
Setup:

- Used Apache Spark Structured Streaming to read live user activity logs



```
# Check if the directory is being populated with new files
os.listdir("/dbfs/mnt/data/streaming-input/")
```

```
Out[98]: ['user_activity_log_1725613637.json',
 'user_activity_log_1725603436.json',
 'user_activity_log_1725613642.json',
 'user_activity_log_1725613627.json',
 'user_activity_log_1725603431.json',
```



- Continuously processed incoming data to compute real-time metrics.

Real-Time Metrics:

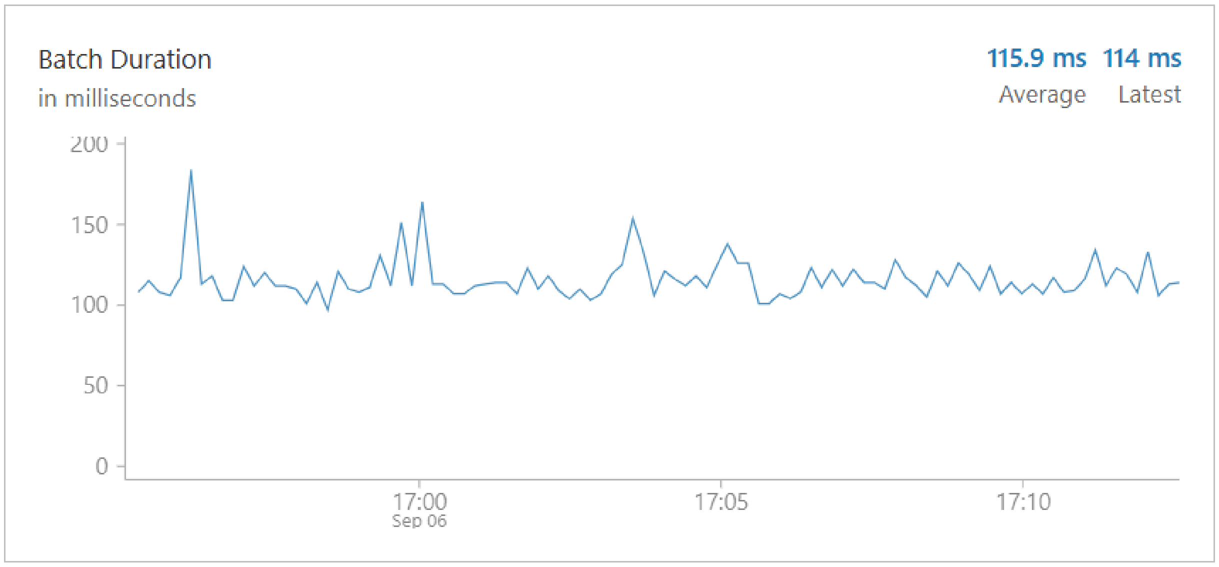
- **Active Users per Minute**: Counted distinct user IDs using a sliding window.

```
from pyspark.sql.functions import window, approx_count_distinct

# Compute active users per minute using approximate distinct co
active_users_per_minute = (
    streaming_df
    .withWatermark("event_time", "1 minute")  # Watermark to ha
    .groupBy(window("event_time", "1 minute"))
    .agg(approx_count_distinct("user_id").alias("active_users")
)

# Start the streaming query with a new name
query = (active_users_per_minute
         .writeStream
         .format("memory")  # Store in-memory for easy display
         .outputMode("complete")
         .queryName("active_users_new")  # Use a different name
         .start())

# Display the results
display(spark.sql("SELECT * FROM active_users_new"))
```



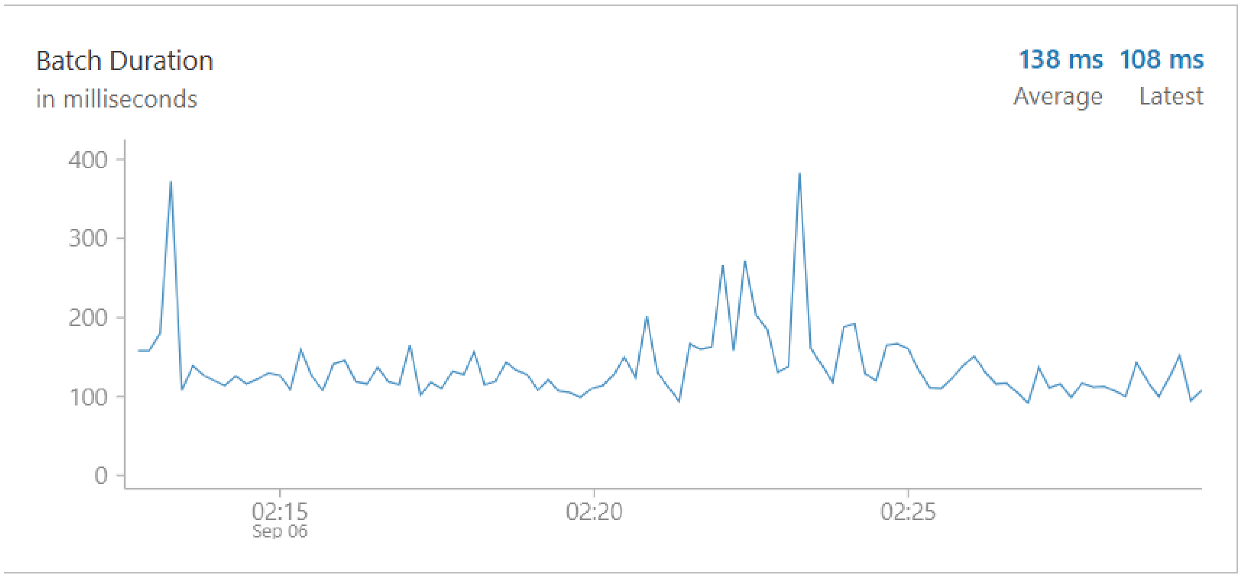
- **Real-Time Conversion Rates**: Calculated by dividing completed transactions by total sessions.

```
from pyspark.sql.functions import expr, count, col, when

# Calculate the conversion rate
conversion_rate = (
    streaming_df
    .withWatermark("event_time", "1 minute")  # Watermark to har
    .groupBy(window("event_time", "1 minute"))
    .agg(
        count(when(col("event_type") == "purchase", 1)).alias("p
        count(when(col("event_type") == "view", 1)).alias("view_
    )
    .withColumn(
        "conversion_rate",
        expr("purchase_count / view_count")
    )
    .select("window.start", "window.end", "conversion_rate")
)

# Start the streaming query to compute the conversion rate per m
query = (conversion_rate
         .writeStream
         .format("memory")  # Store in-memory for easy display
         .outputMode("complete")
         .queryName("conversion_rate")
         .start())

# Display the results
display(spark.sql("SELECT * FROM conversion_rate"))
```



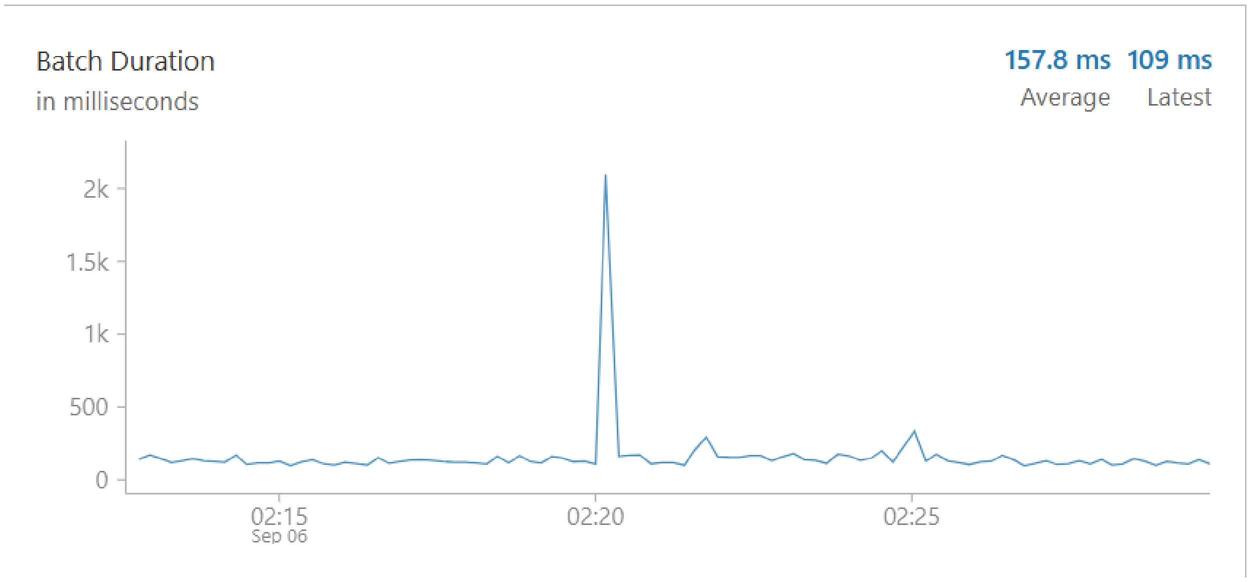
- **Anomaly Detection**: Identified unusual spikes in user activities with threshold-based alerts.

```
rom pyspark.sql.functions import count

# Detect unusual spikes in user activity
activity_spikes = (
    streaming_df
    .withWatermark("event_time", "1 minute")  # Watermark to har
    .groupBy(window("event_time", "1 minute"), "event_type")
    .agg(count("event_type").alias("event_count"))
    .filter("event_count > 100")  # Example threshold for spike
)

# Start the streaming query to detect activity spikes
query = (activity_spikes
         .writeStream
         .format("memory")  # Store in-memory for easy display
         .outputMode("append")
         .queryName("activity_spikes")
         .start())
```

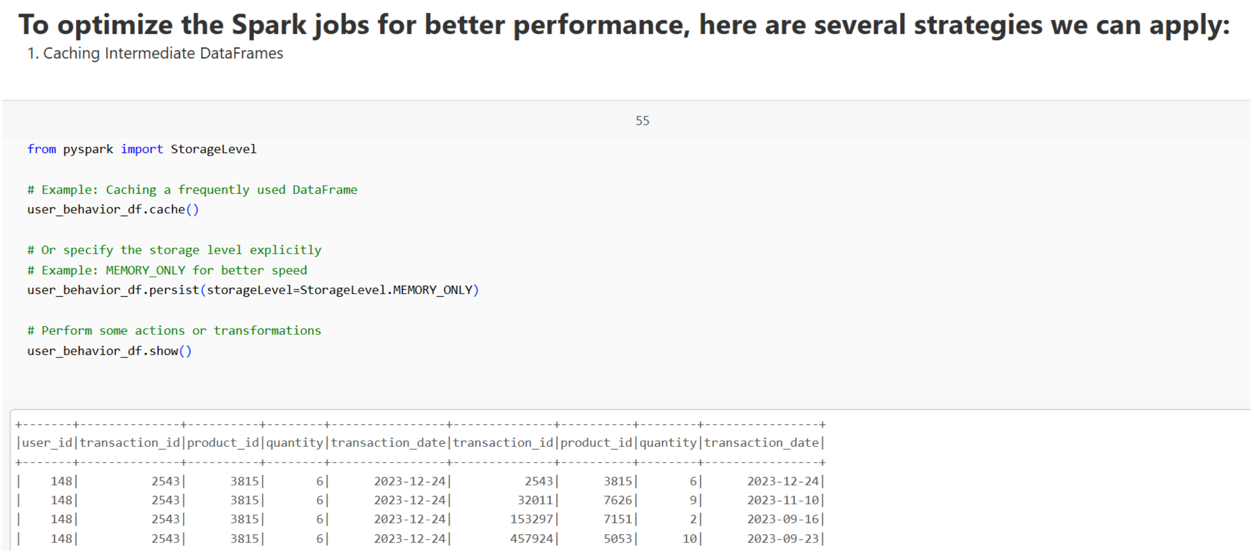
```
# Display the results
display(spark.sql("SELECT * FROM activity_spikes"))
```



6. Performance Optimization

Q6: What strategies did you employ to optimize the performance of your Spark jobs? Provide examples of configuration settings or code optimizations.

- **Caching**: Cached intermediate DataFrames to avoid recomputation during multiple transformations, reducing execution time.



To optimize the Spark jobs for better performance, here are several strategies we can apply:

1. Caching Intermediate DataFrames

55

```
from pyspark import StorageLevel

# Example: Caching a frequently used DataFrame
user_behavior_df.cache()

# Or specify the storage level explicitly
# Example: MEMORY_ONLY for better speed
user_behavior_df.persist(storageLevel=StorageLevel.MEMORY_ONLY)

# Perform some actions or transformations
user_behavior_df.show()
```

```
+-------+--------------+----------+--------+----------------+--------------+----------+--------+----------------+
|user_id|transaction_id|product_id|quantity|transaction_date|transaction_id|product_id|quantity|transaction_date|
+-------+--------------+----------+--------+----------------+--------------+----------+--------+----------------+
|    148|          2543|      3815|       6|      2023-12-24|          2543|      3815|       6|      2023-12-24|
|    148|          2543|      3815|       6|      2023-12-24|         32011|      7626|       9|      2023-11-10|
|    148|          2543|      3815|       6|      2023-12-24|        153297|      7151|       2|      2023-09-16|
|    148|          2543|      3815|       6|      2023-12-24|        457924|      5053|      10|      2023-09-23|
```

- **Partitioning**: Repartitioned DataFrames to optimize the performance of join operations, ensuring data was evenly distributed across Spark partitions.

```
# Repartition DataFrame to increase/decrease the number of partitions
user_behavior_df = user_behavior_df.repartition(10)  # Example: repartitioning to 10 partitions
```

- **Tuning Spark Configurations**: Adjusted configurations such as `spark.sql.shuffle.partitions` and `spark.executor.memory` to balance between memory usage and parallel processing efficiency.

💡 spark.executor.memory: Total amount of memory to allocate per executor. spark.driver.memory: Memory size for the Spark driver. spark.executor.cores: Number of CPU cores per executor.

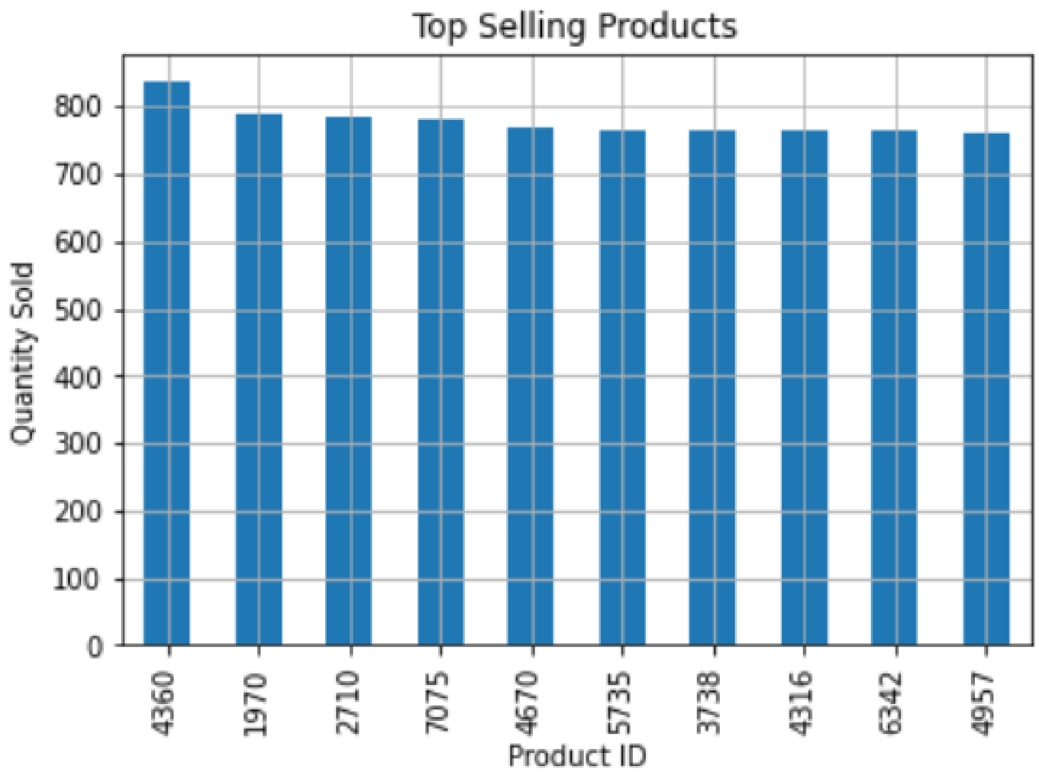
```
# Use the Spark configuration
spark.conf.set("spark.executor.memory", "4g")
spark.conf.set("spark.driver.memory", "4g")
spark.conf.set("spark.executor.cores", "2")
```

Reporting

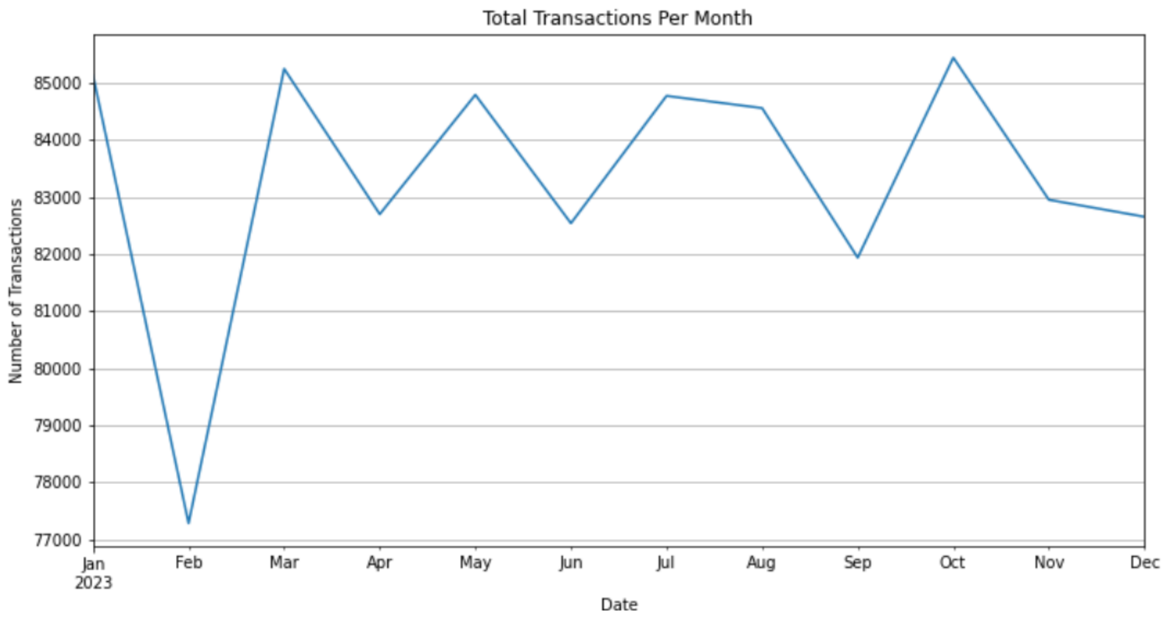
Q7: Show sample outputs or dashboards that visualize the insights derived from the ETL pipeline.

- **Top Products Bar Chart**: Displaying the top 10 products with the highest sales.

```
<Figure size 864x432 with 0 Axes>
```



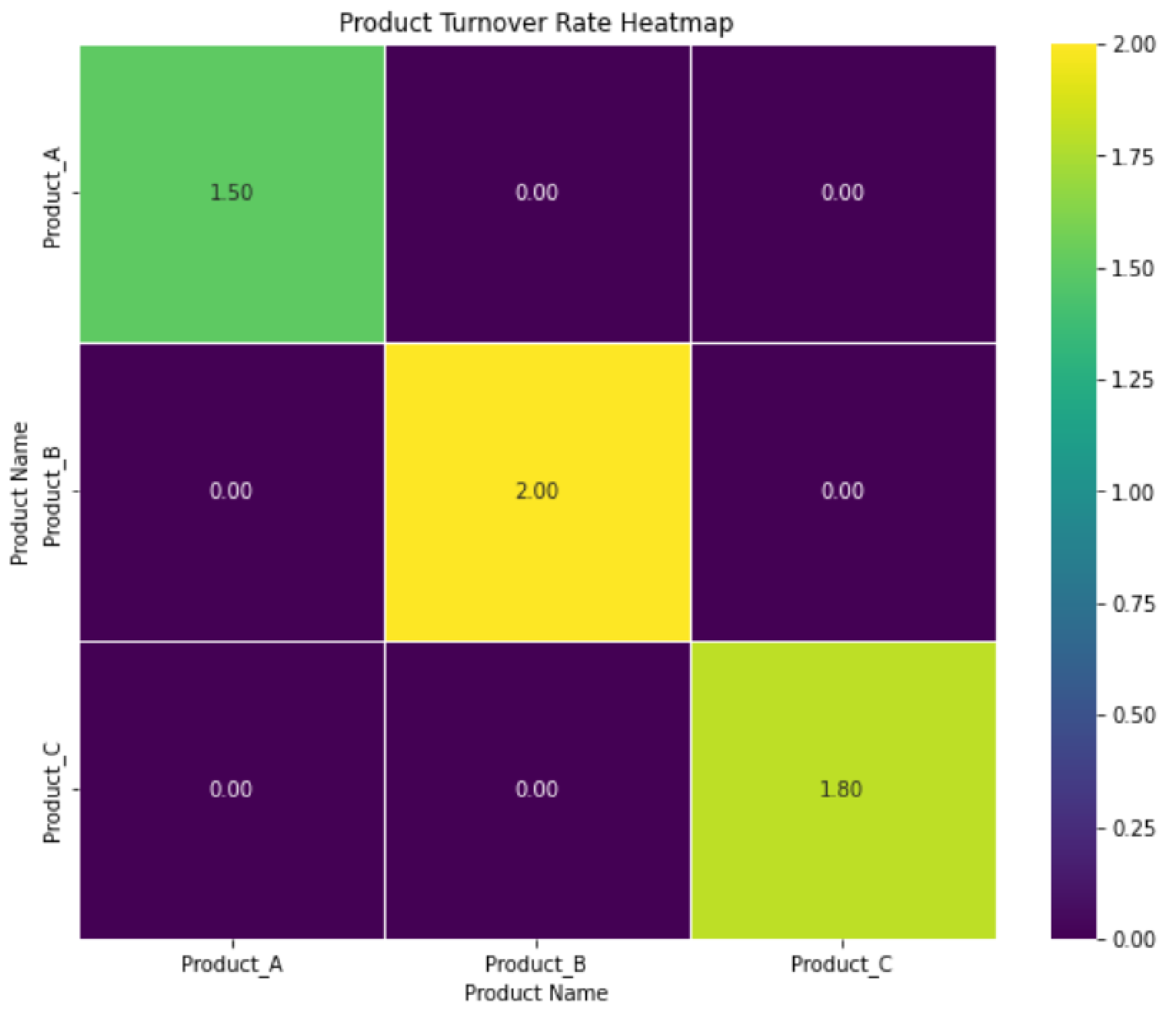
- **Revenue Trend Line Chart**: Showing monthly revenue over the past year.



- **Inventory Turnover Heatmap**: Visualizing turnover rates across different product categories, using a heatmap to show products with the highest turnover rates.

💡 1 Product Categories on One Axis: The x-axis or y-axis should represent different product categories.

2 Turnover Rates: The cells should represent the turnover rates for each product category.

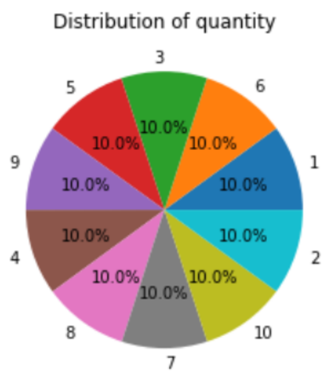


Additional Visualizations

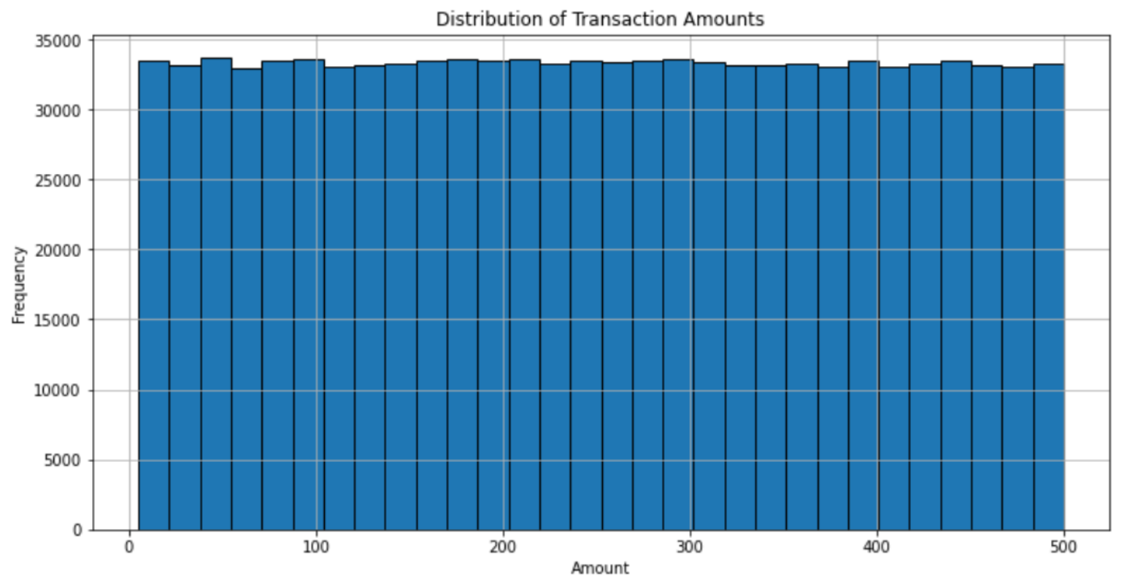
1.Pie Chart of quantity: Number of items purchased.

```
# Plot
plt.figure(figsize=(8, 8))
payment_methods_df.set_index('quantity').plot(kind='pie', y='count', autopct='%1.1f%%', legend=False)
plt.title('Distribution of quantity')
plt.ylabel('')
plt.show()
```

```
<Figure size 576x576 with 0 Axes>
```

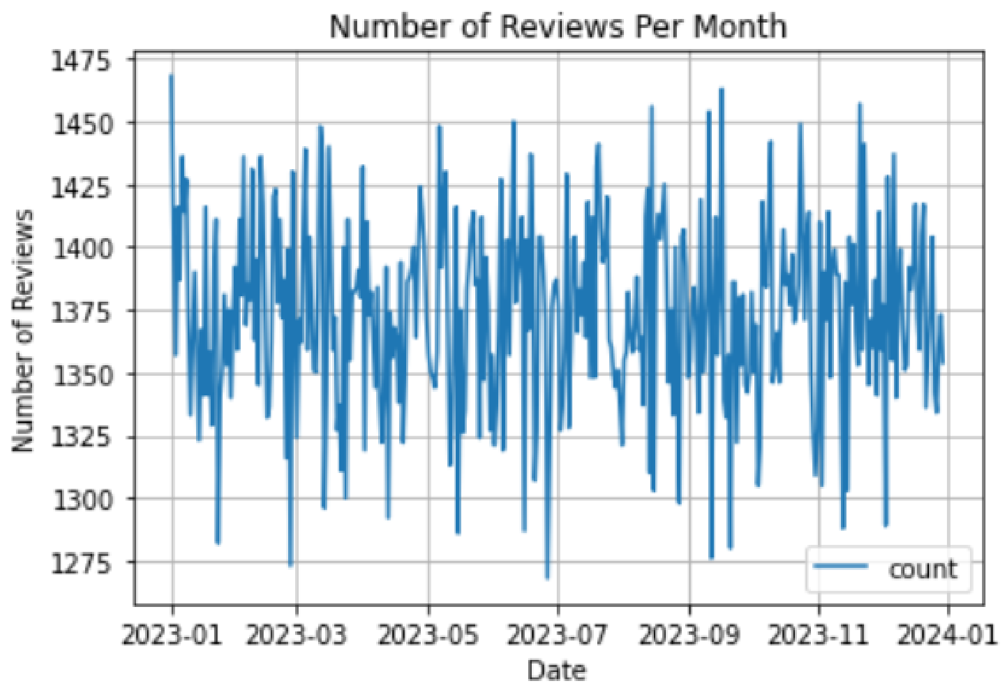


2.**Histogram of Transaction Amounts:**



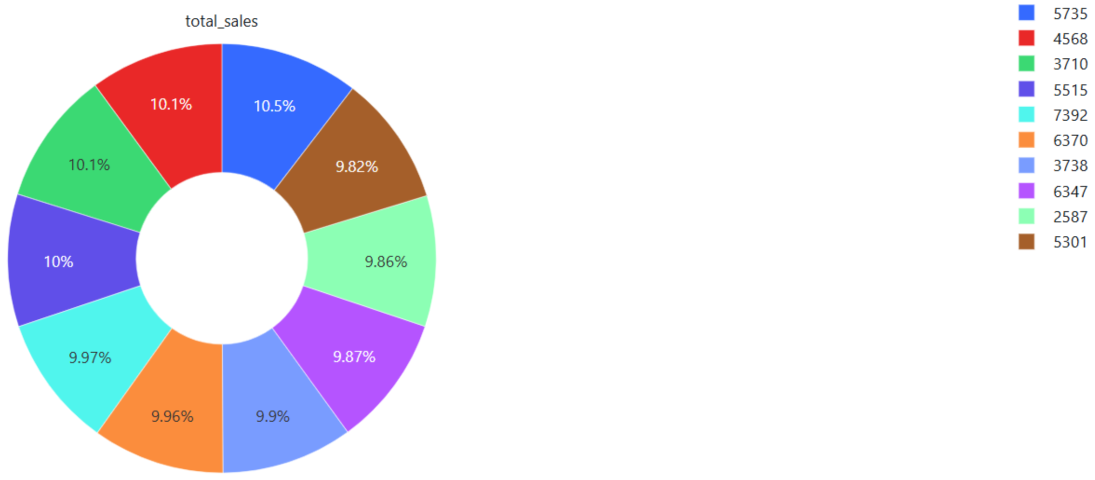
3.Line Chart of Review Volume Over Time:

```
<Figure size 864x432 with 0 Axes>
```

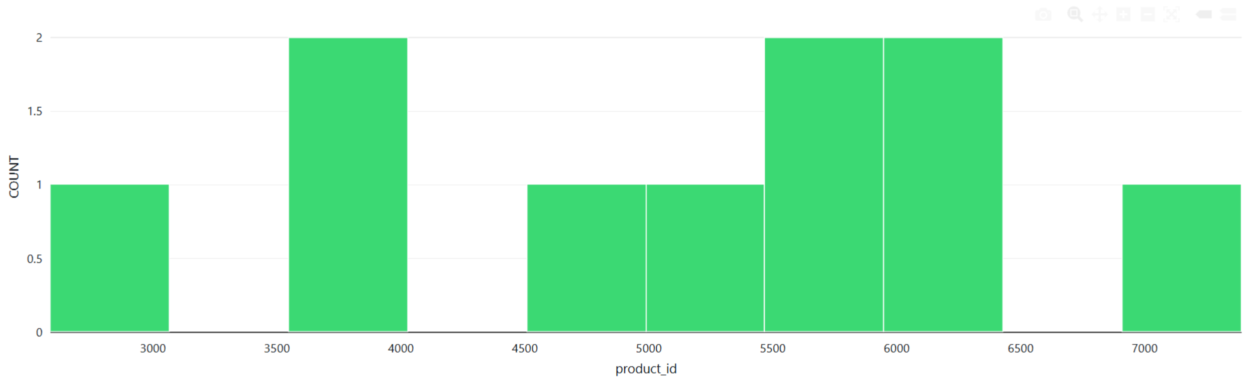


4.Top Products by Sales

💡 pie chart

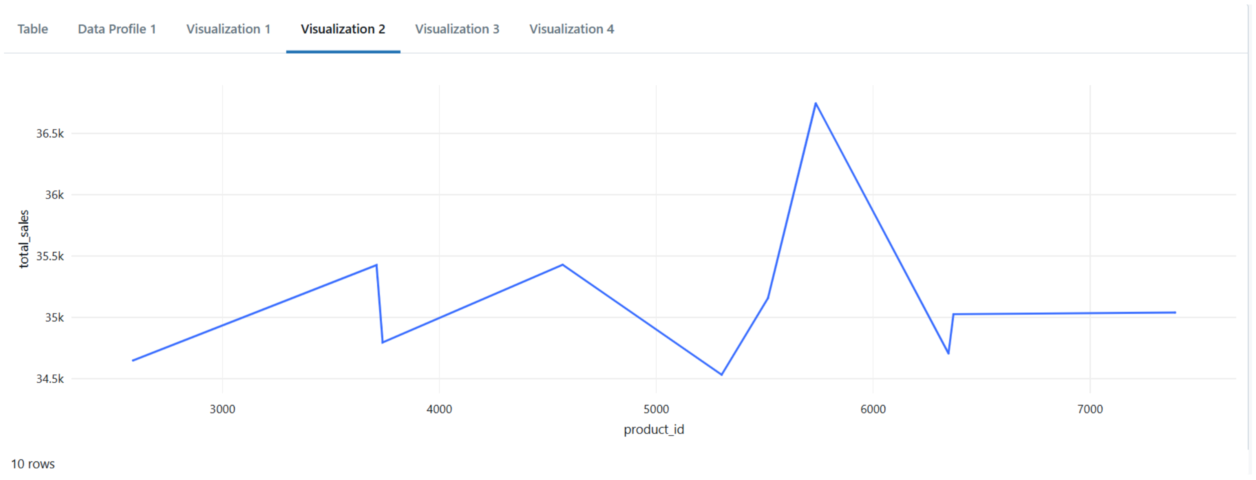


💡 bar plot



10 rows

💡 line chart



Conclusion

The “Real-Time E-commerce Data Pipeline with Spark ETL” project successfully designed and implemented a scalable data pipeline for Shop Ease. Using Apache Spark, the pipeline efficiently processed both batch and real-time data, including user activity logs, transaction records, inventory updates, and customer feedback. It ensured optimal data ingestion, cleaning, transformation, and storage.

▼ Key Achievements Summary:

1. **Effective Data Ingestion**: Handled high volumes of historical and real-time data from multiple sources, ensuring up-to-date insights and comprehensive data history.
2.
Data Transformation and Quality Assurance: Cleaned, standardized, and anonymized data using Apache Spark, ensuring data integrity and consistency.
3.
Real-Time Analytics and Processing: Enabled real-time metrics computation, enhancing insights into user behavior and platform performance.
4.
Optimized Performance: Implemented caching, partitioning, and efficient joins to improve Spark job performance, making the pipeline scalable and

cost-effective.
5.
Comprehensive Reporting and Visualization: Generated key reports and visualizations to support strategic decision-making on sales performance, revenue growth, and inventory management.

▼ **Final Thoughts**

Overall, this project showcases the power of Apache Spark in building a comprehensive ETL pipeline capable of supporting real-time analytics and large-scale data processing for e-commerce platforms. The pipeline's flexibility and scalability make it well-suited for handling the evolving needs of ShopEase, ensuring the company can continuously leverage its data to enhance user experience, optimize operations, and drive business growth. Future enhancements could involve integrating more advanced machine learning models for predictive analytics and further automating the real-time data processing capabilities.

link of the whole notebook you can import it on data bricks

Hadeer Dowidar