Report on Real-Time Stock Price Analysis Using PySpark and Kafka Streaming
2024F-T3 BDM 3603 - Big Data Framework 01
(DSMM Group 1)

Submitted By:
Nirmala Regmi (c0903616)
Sujata Gurung (c0903143)
Prawesh Pokharel (c0913577)
Aishwarya Karki (c0903073)
Sijal Shrestha (c0910639)

Submitted to : Dr. Ishant Gupta

Objective

The goal of this project is to develop a real-time data pipeline for ingesting, processing, analyzing, and visualizing live stock prices. The pipeline leverages distributed and scalable tools like Confluent Cloud for data streaming and PySpark for data processing. The project also includes the implementation of alerts and visualizations to monitor stock performance in real-time.

Tools and Technologies

1. Confluent Cloud

- A cloud-based Kafka platform used for managing streaming stock price data.
- Provides advanced features like schema registry, monitoring, and security for seamless operations.

2. PySpark

- Processes and analyzes streaming data in a distributed and scalable manner.

3. Databricks or Local PySpark Environment

- Acts as the development and execution platform for PySpark jobs.

4. Visualization Tools

- Matplotlib/Seaborn: For static visualizations.
- Databricks Dashboards: For dynamic and interactive visual analytics.

Project Workflow

Step 1: Confluent Kafka Setup

1. **Setup Kafka Cluster:**
    - Use Confluent Cloud for setting up a Kafka cluster with predefined configurations.
        - Steps for Kafka Installation
            i. Create an account in Confluent.
            ii. Create a Kafka Cluster.
            iii. Create a topic.
            iv. Download API From Topic:
            v. Set up Kafka Configuration.
                - Use Confluent's Kafka API credentials (e.g., bootstrap server, API key, secret) to configure your producer or consumer.
            vi. Start Working.

```
=== Confluent Cloud API key ===

API key:
O3LJJAT5NL2RRB6G

API secret:
wTc8QOLiu7ZU5PEXIUNB2BlHp23S5LLEJnOBbK4TeRWGjYVsDFHeN9RPYD8WN/Oo

Resource:
lkc-87gr7m

Bootstrap server:
pkc-619z3.us-east1.gcp.confluent.cloud:9092
```

Fig: API keys for Stock Topic using Confluent

- Utilize Confluent's managed services for better scalability and fault tolerance.

2. **Kafka Producer Setup:**
   - Develop a Kafka producer script to:
     - Pull live data via APIs (e.g., Alpha Vantage, Yahoo Finance).
   - Published data to the Kafka topic stock_prices with the following structure:
     - Ticker: Stock symbol (e.g., AAPL, MSFT).
     - Price: Floating-point stock price.
     - Timestamp: Timestamp of the stock price (ISO format).
   - Key Features

- Configured producer properties such as acks for reliable delivery.
- Partitioned messages by ticker for scalability.

```
Published: {"ticker": "AAPL", "price": 220.78, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "TSLA", "price": 155.73, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "GOOG", "price": 259.58, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "AMZN", "price": 396.91, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "MSFT", "price": 197.36, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "AAPL", "price": 296.63, "timestamp": "2024-12-09 23:09:27"}
Published: {"ticker": "TSLA", "price": 242.99, "timestamp": "2024-12-09 23:09:27"}
Published: {"ticker": "GOOG", "price": 309.02, "timestamp": "2024-12-09 23:09:27"}
Published: {"ticker": "AMZN", "price": 287.97, "timestamp": "2024-12-09 23:09:27"}
Published: {"ticker": "MSFT", "price": 491.9, "timestamp": "2024-12-09 23:09:27"}
Published: {"ticker": "AAPL", "price": 397.64, "timestamp": "2024-12-09 23:09:32"}
Published: {"ticker": "TSLA", "price": 498.41, "timestamp": "2024-12-09 23:09:32"}
Published: {"ticker": "GOOG", "price": 342.97, "timestamp": "2024-12-09 23:09:32"}
Published: {"ticker": "AMZN", "price": 104.55, "timestamp": "2024-12-09 23:09:32"}
Published: {"ticker": "MSFT", "price": 397.66, "timestamp": "2024-12-09 23:09:32"}
Published: {"ticker": "AAPL", "price": 214.94, "timestamp": "2024-12-09 23:09:37"}
Published: {"ticker": "TSLA", "price": 164.92, "timestamp": "2024-12-09 23:09:37"}
Published: {"ticker": "GOOG", "price": 203.75, "timestamp": "2024-12-09 23:09:37"}
Published: {"ticker": "AMZN", "price": 206.71, "timestamp": "2024-12-09 23:09:37"}
Published: {"ticker": "MSFT", "price": 399.07, "timestamp": "2024-12-09 23:09:37"}
Published: {"ticker": "AAPL", "price": 450.9, "timestamp": "2024-12-09 23:09:42"}
Published: {"ticker": "TSLA", "price": 247.12, "timestamp": "2024-12-09 23:09:42"}
Published: {"ticker": "GOOG", "price": 100.52, "timestamp": "2024-12-09 23:09:42"}
Published: {"ticker": "AMZN", "price": 188.04, "timestamp": "2024-12-09 23:09:42"}
Published: {"ticker": "MSFT", "price": 139.66, "timestamp": "2024-12-09 23:09:42"}
...
Published: {"ticker": "TSLA", "price": 127.38, "timestamp": "2024-12-09 23:44:34"}
Published: {"ticker": "GOOG", "price": 286.78, "timestamp": "2024-12-09 23:44:34"}
Published: {"ticker": "AMZN", "price": 164.65, "timestamp": "2024-12-09 23:44:34"}
Published: {"ticker": "MSFT", "price": 483.01, "timestamp": "2024-12-09 23:44:34"}
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Fig: Valued produced by the producer

Step 2: PySpark Streaming Integration

1. **Library Imports:**
   - Import PySpark libraries and Confluent Kafka dependencies.

2. **Spark Session Configuration:**
    - Set up a Spark session with necessary configurations for integrating with Confluent Kafka.

Step 3: Schema Definition

1. Define the schema for the stock price data:
    - Fields might include symbol, price, volume, timestamp, and exchange.
2. Use PySpark's StructType for schema enforcement during JSON parsing.

Step 4: Data Ingestion from Confluent Kafka

1. **Kafka Consumer in PySpark:**
    - Use PySpark to subscribe to the stock_prices Kafka topic.
    - Utilize Confluent's schema registry for seamless data serialization and deserialization.
2. **Processing:**
    - Parsed data using a predefined schema (ticker, price, timestamp) and converted JSON messages into PySpark DataFrames.
    - Processed data included:
        - Real-Time Metrics: Moving averages, volatility, and price change percentage.
        - Alerts: Triggered conditions like a 5% price drop or rise within a short period.

```
Successfully subscribed to topic: stock
Partitions assigned: [TopicPartition{topic=stock,partition=0,offset=-1001,leader_epoch=None,error=None}, TopicPartition{topic=stock,part
Assigned partition TopicPartition{topic=stock,partition=0,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=1,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=2,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=3,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=4,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=5,offset=-1001,leader_epoch=None,error=None}
Consumed message: {'ticker': 'AAPL', 'price': 345.09, 'timestamp': '2024-12-09 22:00:22'}
+------+------+-------------------+
|ticker| price|          timestamp|
+------+------+-------------------+
|  AAPL|345.09|2024-12-09 22:00:22|
+------+------+-------------------+

Appended data to Delta table: /mnt/delta/stock_value
Consumed message: {'ticker': 'AAPL', 'price': 208.25, 'timestamp': '2024-12-09 22:00:27'}
+------+------+-------------------+
|ticker| price|          timestamp|
+------+------+-------------------+
|  AAPL|208.25|2024-12-09 22:00:27|
+------+------+-------------------+

Appended data to Delta table: /mnt/delta/stock_value
Consumed message: {'ticker': 'AAPL', 'price': 246.16, 'timestamp': '2024-12-09 22:00:32'}
...
+------+------+-------------------+
|  MSFT|275.22|2024-12-09 19:57:26|
+------+------+-------------------+

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Fig: Delta Table

Step 5: Real-Time Data Processing

1. **Metrics Calculation:**
   - Compute key metrics in real-time, such as:
     - Moving Averages: Smooth fluctuations using a sliding window.
     - Price Changes: Measure percentage change between consecutive time intervals.
     - Volatility: Evaluate the price variation over a specific period.
2. **Alert System:**
   - Set up real-time alerts triggered by conditions, such as:

    - A stock price increase or decrease exceeding 5% within a minute.
    - Volatility breaching predefined thresholds.
- Alerts can be logged or sent via notifications to stakeholders.

Step 6: Data Sink and Visualization

1. **Data Sink:**
    - Wrote processed data to a Delta table located at /mnt/delta/stock_value.
    - The Delta table supported:
        - Real-time updates for dynamic analytics.
        - Efficient querying for historical analysis.

```
df = spark.read.format("delta").load(delta_table_path)

# Display the contents of the Delta table
df.show()
```

```
+------+------+-------------------+
|ticker| price|          timestamp|
+------+------+-------------------+
|  MSFT| 353.1|2024-12-09 19:08:30|
|  AAPL|192.62|2024-12-09 23:11:52|
|  MSFT|189.91|2024-12-09 19:08:35|
|  AAPL|392.65|2024-12-09 22:25:25|
|  AAPL|450.49|2024-12-09 22:02:17|
|  AAPL|287.03|2024-12-09 22:09:52|
|  MSFT|275.22|2024-12-09 19:57:26|
|  MSFT|458.14|2024-12-09 19:22:20|
|  AAPL|377.68|2024-12-09 22:23:15|
|  AAPL|407.88|2024-12-09 22:08:02|
|  AAPL| 453.1|2024-12-09 22:02:07|
|  AAPL|489.52|2024-12-09 22:23:25|
|  AAPL|326.37|2024-12-09 22:07:07|
|  AAPL|324.13|2024-12-09 22:21:25|
|  AAPL|214.94|2024-12-09 23:09:37|
|  MSFT|395.99|2024-12-09 19:14:00|
|  MSFT|185.56|2024-12-09 19:09:45|
|  MSFT|305.84|2024-12-09 19:23:50|
|  AAPL|427.88|2024-12-09 22:22:40|
|  MSFT|439.97|2024-12-09 19:18:40|
+------+------+-------------------+
only showing top 20 rows
```

Fig: Delta Table

2. Visualization Dashboard:

- Use visualization tools to create:
  - Real-Time Charts: Display moving averages, price trends, and alerts.
  - Interactive Dashboards: Built in Databricks to allow dynamic exploration of metrics.

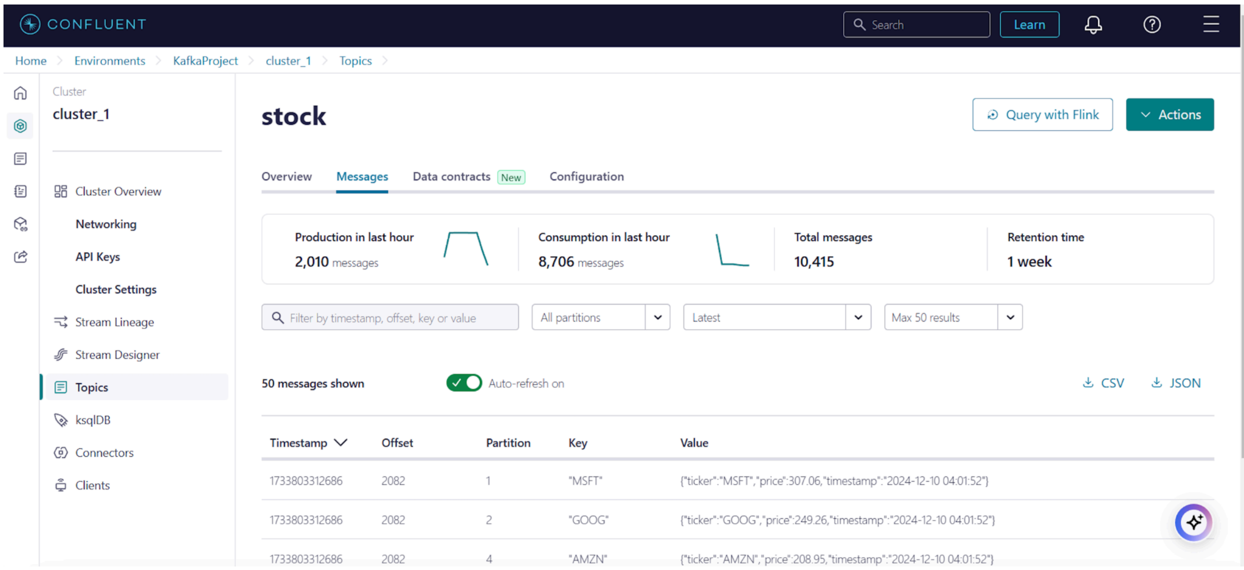

Fig: Confluent Dashboard

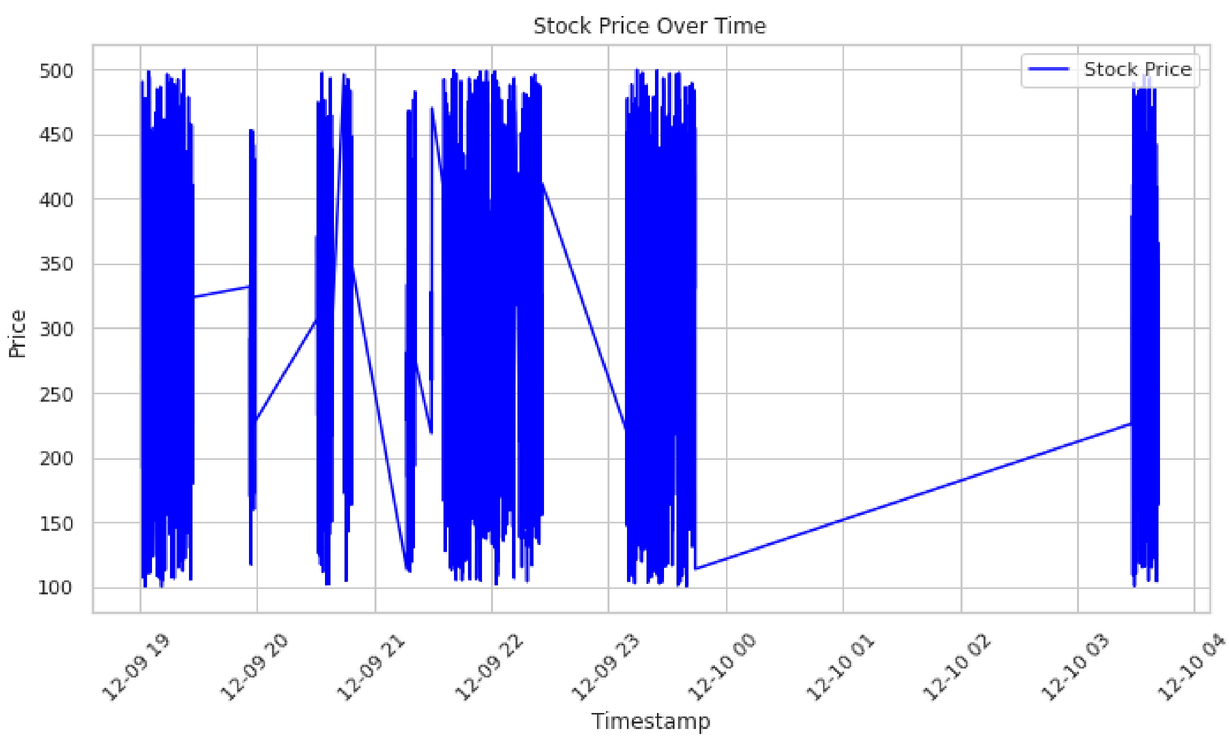

Fig: Average stock price over time

Fig: Average Stock Price by Day for two stock price

Step 7: Batch Processing for Summaries

1. **Daily Summary Aggregation:**
    - Aggregate data into daily summaries, including:
        - Average, minimum, and maximum prices.
        - Total traded volume.
        - Count of significant price changes (e.g., over 5%).
2. **Storage of Summaries:**
    - Persist daily summaries in Delta tables or relational databases for historical trend analysis.

Key Features

1. **Scalability:**
    - Confluent Kafka ensures robust data streaming with minimal setup overhead.
2. **Real-Time Insights:**
    - Continuous computation of metrics and real-time visualization of stock performance.
3. **Alerts:**
    - Notifications for significant events, allowing users to respond quickly.
        - Calculates the percentage change in stock prices for each ticker over consecutive timestamps. It uses a window specification (Window.partitionBy("ticker").orderBy("timestamp")) to group data by ticker and order it by timestamp for sequential analysis. The lag function retrieves the previous price for each ticker, enabling the computation of the price change percentage with (current_price - previous_price) / previous_price * 100. Rows with significant changes (above 5% or below -5%) are filtered using a condition on the calculated column price_change_percent. This helps identify noteworthy market movements that could indicate volatility or unusual activity. The resulting dataset, alerts,

highlights these significant price fluctuations for further analysis.

4. **Batch Analytics:**
    - Daily summaries provide insights into long-term trends and market behaviors.

Challenges and Solutions

1. **Challenge:** Handling schema evolution in data streams.
    - **Solution**: Utilize Confluent's schema registry to manage and evolve schemas.
2. **Challenge:** Ensuring real-time latency with large data volumes.
    - **Solution**: Optimize PySpark processing with efficient partitioning and caching.
3. **Challenge:** Monitoring and troubleshooting in a distributed setup.
    - **Solution**: Leverage Confluent's monitoring tools for real-time diagnostics.

Future Enhancements

1. **Machine Learning Integration:**
    - Implement predictive analytics, such as forecasting stock prices or detecting anomalies using ML models.

2. **Cloud-Scale Deployment:**
    - Scale the pipeline using cloud-native tools and integrate with other Confluent services.
3. **Multi-Market Integration:**
    - Expand to handle data from multiple stock markets or financial instruments.
4. **Extended Visualization:**
    - Add interactive filters and drill-down capabilities in dashboards for better insights.

Conclusion

By leveraging Confluent Kafka and PySpark, this project delivers a robust and scalable solution for real-time stock price analysis. The integration of metrics computation, alerting, and visualization ensures timely and actionable insights, setting the foundation for advanced analytics and predictive modeling in the future.

Codes:

Producer Code:

```python
import random
import json
import time
from datetime import datetime
from confluent_kafka import Producer

# Kafka configuration
producer = Producer({
    'bootstrap.servers': 'pkc-619z3.us-east1.gcp.confluent.cloud:9092',  # Kafka server
    'security.protocol': 'SASL_SSL',
    'sasl.mechanism': 'PLAIN',
    'sasl.username': 'O3LJJAT5NL2RRB6G',  # Replace with your API Key
    'sasl.password': 'wTc8QOLiu7ZU5PEXIUNB2BlHp23S5LLEJnOBbK4TeRWGjYVsDFHeN9RPYD8WN/Oo',  # Replace with your API Secret
    'debug': 'security'  # Enable debugging for detailed logs
})

def delivery_report(err, msg):
    if err is not None:
        print(f"Message delivery failed: {err}")
    else:
        print(f"Message delivered to {msg.topic()} [{msg.partition()}] at offset {msg.offset()}")

# Simulate stock price data
def simulate_stock_data(ticker):
    return {
        "ticker": ticker,
        "price": round(random.uniform(100, 500), 2),  # Random price between 100 and 500
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }
```

```python
# Publish stock prices to Kafka
def publish_stock_prices(tickers, topic, interval=2):
    try:
        while True:
            for ticker in tickers:
                stock_data = simulate_stock_data(ticker)
                message = json.dumps(stock_data)

                # Produce message to Kafka topic
                try:
                    producer.produce(topic, key=ticker, value=message, callback=delivery_report)
                    print(f"Published: {message}")
                except Exception as e:
                    print(f"Error producing message: {e}")

            # Wait for the interval before sending the next batch
            time.sleep(interval)
    except KeyboardInterrupt:
        print("Stopping producer...")
    finally:
        print("Flushing any remaining messages...")
        producer.flush()
        print("Producer stopped.")

# List of stock tickers to simulate
stock_tickers = ['AAPL', 'TSLA', 'GOOG', 'AMZN', 'MSFT']

# Kafka topic
topic_name = 'stock'

# Run the producer
publish_stock_prices(stock_tickers, topic_name, interval=5)
```

```
Published: {"ticker": "AAPL", "price": 220.78, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "TSLA", "price": 155.73, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "GOOG", "price": 259.58, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "AMZN", "price": 396.91, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "MSFT", "price": 197.36, "timestamp": "2024-12-09 23:09:22"}
Published: {"ticker": "AAPL", "price": 296.63, "timestamp": "2024-12-09 23:09:27"}
Published: {"ticker": "TSLA", "price": 242.99, "timestamp": "2024-12-09 23:09:27"}
```

Consumer Code:

```python
import json
import time
from datetime import datetime, timedelta
import random
from confluent_kafka import Producer

# Kafka configuration
conf = {
    'bootstrap.servers': 'pkc-619z3.us-east1.gcp.confluent.cloud:9092',  # Your Kafka cluster's bootstrap server
    'security.protocol': 'SASL_SSL',
    'sasl.mechanisms': 'PLAIN',
    'sasl.username': 'O3LJJAT5NL2RRB6G',
    'sasl.password': 'wTc8QOLiu7ZU5PEXIUNB2BlHp23S5LLEJnOBbK4TeRWGjYVsDFHeN9RPYD8WN/Oo',  # Replace with your SASL password
}
```

```python
import json
from datetime import datetime
from confluent_kafka import Consumer, KafkaException, KafkaError
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, FloatType, TimestampType

# Kafka configuration
consumer = Consumer({
    'bootstrap.servers': 'pkc-619z3.us-east1.gcp.confluent.cloud:9092',  # Replace with your Kafka server
    'security.protocol': 'SASL_SSL',
    'sasl.mechanism': 'PLAIN',
    'sasl.username': 'O3LJJAT5NL2RRB6G',  # Replace with your API Key
    'sasl.password': 'wTc8QOLiu7ZU5PEXIUNB2BlHp23S5LLEJnOBbK4TeRWGjYVsDFHeN9RPYD8WN/Oo',  # Replace with your API Secret
    'group.id': 'python-group-1',  # Consumer group ID
    'auto.offset.reset': 'earliest'  # Start reading messages from the beginning if no offset is committed
})

# Kafka topic
topic_name = 'stock'  # Using the provided topic name

# Initialize Spark session
spark = SparkSession.builder \
    .appName("Real-Time Stock Price Analysis") \
    .getOrCreate()
```

```python
# Define schema for stock data
schema = StructType([
    StructField("ticker", StringType(), True),
    StructField("price", FloatType(), True),
    StructField("timestamp", TimestampType(), True)
])

# Delta table path
delta_table_path = "/mnt/delta/stock_value"  # Make sure to use the correct path

# Subscribe to the topic with error handling
def subscribe_to_topic():
    try:
        # Attempt subscription to the topic
        consumer.subscribe([topic_name], on_assign=on_partition_assign)
        print(f"Successfully subscribed to topic: {topic_name}")
    except Exception as e:
        print(f"Error during subscription: {e}")

def on_partition_assign(consumer, partitions):
    """
    This function is called when partitions are assigned to the consumer.
    It's good for handling custom partition assignment logic.
    """
    print(f"Partitions assigned: {partitions}")
    for partition in partitions:
        print(f"Assigned partition {partition}")

def consume_messages():
    try:
        subscribe_to_topic()  # Subscribe to topic before starting consumption
        while True:
            msg = consumer.poll(1.0)  # Poll for new messages (1-second timeout)

            if msg is None:
                continue  # No message received
            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    print(f"Reached end of partition: {msg.topic()} [{msg.partition()}]")
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
```

```
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                # Message successfully received
                message = json.loads(msg.value().decode('utf-8'))
                print(f"Consumed message: {message}")

                # Parse the timestamp string into a datetime object
                timestamp_str = message['timestamp']
                timestamp_obj = datetime.strptime(timestamp_str, '%Y-%m-%d %H:%M:%S')

                # Update message with the correct timestamp object
                message['timestamp'] = timestamp_obj

                # Create a DataFrame with the received message
                df = spark.createDataFrame([message], schema)

                # Show the dataframe (optional)
                df.show()

                # Append the data to Delta table
                df.write \
                  .format("delta").mode("append").save(delta_table_path)

                print(f"Appended data to Delta table: {delta_table_path}")

    except KeyboardInterrupt:
        print("Stopping consumer...")
    finally:
        # Close down consumer cleanly
        consumer.close()
        print("Consumer stopped.")

# Run the consumer
consume_messages()
```

```
Successfully subscribed to topic: stock
Partitions assigned: [TopicPartition{topic=stock,partition=0,offset=-1001,leader_epoch=None,error=None}, TopicPartition{topic=s
Assigned partition TopicPartition{topic=stock,partition=0,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=1,offset=-1001,leader_epoch=None,error=None}
Assigned partition TopicPartition{topic=stock,partition=2,offset=-1001,leader_epoch=None,error=None}
```

Main Code:

```
from pyspark.sql.types import StructType, StructField, StringType, FloatType, TimestampType, IntegerType

# Define schema with ticker, price, timestamp, and volume
schema = StructType([
    StructField("ticker", StringType(), True),
    StructField("price", FloatType(), True),
    StructField("timestamp", TimestampType(), True),
    StructField("volume", IntegerType(), True)
])
```

+ Code + Markdown

Step 4: Read Data from Kafka

```
kafka_stream_df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "pkc-619z3.us-east1.gcp.confluent.cloud:9092") \
    .option("subscribe", "stock") \
    .load()
```

```
stock_df = kafka_stream_df \
    .selectExpr("CAST(value AS STRING)") \
    .select("value") \
    .selectExpr("json_tuple(value, 'ticker', 'price', 'timestamp') as (ticker, price, timestamp)") \
    .withColumn("timestamp", col("timestamp").cast(TimestampType())) \
    .withColumn("price", col("price").cast(FloatType()))
```

```
# Example DataFrame loading (replace with your actual data source)


# Define the WindowSpec for moving average calculation (with no partitioning)
window_spec = Window.orderBy(F.col("timestamp"))

# Calculate the moving average of 'price'
processed_stream = parsed_stream.withColumn(
    "moving_avg",
    F.avg("price").over(window_spec)
)

# Print the schema to ensure the columns are correct
processed_stream.printSchema()
```

```
root
 |-- ticker: string (nullable = true)
 |-- price: float (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- volume: integer (nullable = true)
 |-- moving_avg: double (nullable = true)
```

```
parsed_stream.printSchema()
```

```
root
 |-- ticker: string (nullable = true)
 |-- price: float (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- volume: integer (nullable = true)
```

```python
from pyspark.sql import SparkSession
import pandas as pd
import matplotlib.pyplot as plt

# Initialize Spark session
spark = SparkSession.builder.appName("DeltaTableRead").getOrCreate()

# Path to the Delta table
delta_table_path = "/mnt/delta/stock_value/"

# Read the Delta table
df = spark.read.format("delta").load(delta_table_path)

# Convert to Pandas DataFrame for easier plotting
pandas_df = df.toPandas()

# Check the first few rows of the dataframe
print(pandas_df.head())

#
```

```
  ticker       price           timestamp
0   MSFT  153.729996 2024-12-09 22:08:57
1   AAPL  210.369995 2024-12-09 22:04:52
2   AAPL  438.109985 2024-12-09 23:39:59
3   AAPL  405.399994 2024-12-10 11:59:23
4   AAPL  330.880005 2024-12-09 23:34:28
```

References:

Confluent Kafka - https://www.confluent.io/cloud-kafka/

Stackoverflow- https://stackoverflow.com/questions/76888375/integrate-pyspark-structured-streaming-with-confluent-kafka

Databricks- https://docs.databricks.com/en/pyspark/index.html