Full Name: Kirill Klimashenko

Student ID: 10622092

Course Name: Decision Support Systems

Date of Submission: February 27, 2025

Exploration and Implementation of Real-Time Data Streaming with Kafka and Pandas

Study Explanation

Objective

Today, I set out to deepen my understanding of real-time data streaming by implementing an Apache Kafka-based system integrated with Python’s pandas library. My primary goal was to build a functional pipeline that reads data from a JSON file, streams it through Kafka topics, processes it with multiple consumers, and logs the results. I aimed to simulate a real-world scenario where payment data is streamed, cleaned, transformed, and analyzed, while troubleshooting any issues that arose during setup and execution. This hands-on project was intended to solidify my skills in messaging systems and data manipulation for big data applications.

Methodology

I started by setting up my environment on Windows 10, using my Kafka installation at C:\Users\user\projects\kafka and Python 3 with pandas and confluent\_kafka installed. My dataset, merged\_payments\_customers.json, included payment records with fields like customer\_id, amount, payment\_type, and activebool. I kicked off with pandas to load and explore this JSON file. I imported it into a DataFrame using pd.read\_json(), which gave me a structured view of the data.

With pandas, I performed initial analysis: calculating the total payment per customer with df.groupby('customer\_id')['amount'].sum(), calculated mean and top five customers. I checked for missing values using df.isnull().sum()—noticing payment\_type was consistently null—and replaced them with "unknown" via df['payment\_type'].fillna('not\_disclosed'). This preprocessing ensured my data was clean and structured before streaming.

Next, I shifted to Kafka. I wrote a producer script to read the JSON file record-by-record with json.load(), streaming each to the dss-payment-stream topic using confluent\_kafka.Producer. I set bootstrap.servers='localhost:9092' and added a 0.5-second delay with time.sleep(0.5) to simulate real-time flow. I then designed a consumer group (dss-payment-group) with two consumers to read from this topic, applying the same pandas-inspired transformations: replacing null payment\_type with "unknown", discarding negative amounnts. A separate logging consumer (dss-logging-group) appended records to a CSV with timestamps.

Midway, Kafka crashed due to a file lock on \_\_consumer\_offsets-28, spotted in the logs at 00:45:07. I used pandas to re-analyze my sample data while troubleshooting, ensuring my transformations held. After clearing the log directory (C:\Users\user\projects\tmp\kafka-logs) and restarting Kafka and ZooKeeper, I confirmed stability and recreated topics with kafka-topics.bat.

Conclusion

Today’s work was a rollercoaster of progress and problem-solving. I successfully built and ran a Kafka Producer that streamed payment data, designed a consumer group for processing, and set up a logging mechanism—all integrated with pandas for data handling. The pipeline worked briefly, sending messages to `dss-payment-stream`, but crashed due to a file lock on `\_\_consumer\_offsets-28`, revealed in the Kafka logs at `00:45:07`. This was caused by a Windows process holding a `.timeindex` file, forcing Kafka to shut down. I resolved it by clearing the log directory (`C:\Users\user\projects\tmp\kafka-logs`) and restarting Kafka, confirming it restarted cleanly afterward.

Key findings include Kafka’s sensitivity to file locks on Windows, the importance of clean shutdowns, and the need for robust error handling in producers. My pandas preprocessing ensured data readiness, and Apache Kafka proved powerful for messaging once stabilized. I learned to navigate Kafka’s logs and configs deeply, overcoming initial SASL confusion (a red herring from an old setup) and settling on `PLAINTEXT` as intended.