Comparative Analysis of Kafka-Spark Pipelines for Real-Time and Batch Bitcoin Data Processing

1. Introduction

This report details a project designed to implement and compare two distinct data processing paradigms – real-time streaming and periodic batch processing – for analyzing financial market data. Specifically, it focuses on processing Bitcoin (BTC-USD) ticker data obtained from the Coinbase Pro WebSocket API. The system leverages Apache Kafka for robust data ingestion and buffering, Apache Spark for powerful distributed processing (both Spark Structured Streaming and Spark Batch), SQLite for intermediate data storage in the batch pipeline, and Streamlit for visualizing the comparative results.

The core components include:

- A Kafka producer (producer.py) subscribing to the Coinbase WebSocket feed and publishing BTC-USD ticker data to a Kafka topic.
- A Kafka consumer (consumer.py) reading from the topic and persisting the raw data into a SQLite database.
- A Spark Structured Streaming application (spark_processor.py) processing data directly from Kafka in near real-time, performing windowed aggregations.
- A Spark Batch application (batch_processor.py) processing data periodically from the SQLite database, performing global aggregations over the accumulated data.
- A simple scheduler (batch_scheduler.py) to trigger the batch job at regular intervals.
- A Streamlit dashboard (dashboard.py) to visualize and compare the outputs and performance characteristics of the streaming and batch modes.

The primary objective is to analyze key metrics like average price, price range, trading volume, trade counts, and market sentiment (derived from buy/sell ratios) using both approaches and evaluate the trade-offs in terms of latency, data granularity, processing time, and the nature of the insights derived.

2. Installation of Software

Core Systems:

- Python (3.x recommended)
- Apache Kafka (including Zookeeper)
- Apache Spark (compatible with the PySpark version used)
- SQLite3 (usually bundled with Python)

Python Libraries:

- kafka-python: For Kafka producer and consumer interaction.
- websocket-client: For connecting to the Coinbase WebSocket feed.
- pyspark: For Spark processing logic (Streaming and Batch).
- sqlite3: For database interaction in the consumer.
- pandas: Used within the Streamlit dashboard for data manipulation.
- matplotlib: Used within the Streamlit dashboard for plotting.
- streamlit: For building the interactive dashboard.

Spark Dependencies:

- SQLite JDBC Driver (sqlite-jdbc-*.jar, e.g., sqlite-jdbc-3.36.0.3.jar as seen in batch_processor.py and batch_scheduler.py): Required for Spark to connect to the SQLite database via JDBC in the batch processor. This JAR file needs to be accessible to the Spark driver and executors.

3. Input Data

a. Source:

- The primary data source is the real-time WebSocket feed provided by Coinbase Pro: wss://ws-feed.exchange.coinbase.com.
- Specifically, the system subscribes to the ticker channel for the BTC-USD product ID, as configured in producer.py.

- **b. Description:**

  The data consists of JSON messages representing ticker updates for Bitcoin trading against the US Dollar.

  The producer.py script filters these messages, ensuring they are of type: ticker, have the correct product_id: BTC-USD, and contain all required fields before publishing to Kafka.

  The key fields extracted and used in subsequent processing steps (persisted by consumer.py and processed by Spark) are:

  - trade_id: Unique identifier for the trade event (used as primary key in SQLite).
  - price: The price of the last trade.
  - best_bid: The best available bid price.
  - best_ask: The best available ask price.
  - side: The side of the taker order ('buy' or 'sell').
  - time: The timestamp of the trade event (ISO 8601 format).
  - last_size: The volume/quantity of the last trade.

  This data represents a continuous stream of individual trade events occurring on the Coinbase exchange for BTC-USD.

4. Streaming Mode Experiment

a. Description:

- The streaming pipeline is implemented using Spark Structured Streaming in spark_processor.py.
- It reads data directly from the raw_prices Kafka topic as soon as messages are produced.
- The pipeline performs the following steps:
  1. Connects to Kafka and subscribes to the raw_prices topic.
  2. Parses the incoming JSON messages based on a defined schema.
  3. Casts relevant fields (price, last_size, time) to appropriate data types (Float, Timestamp).
  4. Applies windowed aggregations based on event time.
  5. Calculates derived metrics within each window (e.g., price_range, volatility_flag, market_sentiment, buy_sell_ratio, volume_spike).
  6. Uses watermarking (withWatermark("time", "1 minute")) to handle late-arriving data, discarding state for windows older than the watermark.
- This approach processes data incrementally, providing low-latency insights into market activity within defined time intervals.

b. Windows:

- The streaming aggregation uses a **tumbling window** of **1 minute** duration, based on the time field (event time) of the incoming data (window(col("time"), "1 minute")). This means aggregations are calculated for distinct, non-overlapping 1-minute intervals.

c. Results:

- The output of the streaming process consists of aggregated metrics calculated for each 1-minute window.
- These results are continuously appended to CSV files located in the /app/data/stream_csv/ directory. Each file likely corresponds to a Spark micro-batch output, and each row represents the aggregated metrics for a specific 1-minute window.
- Key metrics produced per window
  include: avg_price, max_price, min_price, total_volume, trade_count, buy_sell_ratio, market_sentiment, volatilit etc.
- Results are also printed to the console (.outputMode("update")) for real-time monitoring during development/debugging.
- The dashboard (dashboard.py) reads these CSV files to display time-series trends and the latest window's metrics.
- A basic timing (Streaming Aggregation Logic Time) is printed, measuring the time taken to define the Spark streaming logic, not the end-to-end processing latency per window.

5. Batch Mode Experiment

a. Description:

- The batch processing pipeline is implemented using Spark's core batch processing capabilities in batch_processor.py.
- It reads data from the btc_prices table within the SQLite database (/app/data/crypto_prices.db), which is populated by the consumer.py script.
- The batch_scheduler.py script invokes this Spark job periodically (every 10 minutes).
- The pipeline performs the following steps upon each execution:
  1. Connects to the SQLite database using the JDBC driver.
  2. Loads the *entire* btc_prices table into a Spark DataFrame.
  3. Casts relevant fields to appropriate data types.
  4. Performs *global* aggregations across all the data read from the database (i.e., no windowing).
  5. Calculates derived metrics based on these global aggregates.
- This approach processes all accumulated data up to the point of execution, providing a historical summary.

- **b. Data Size:**
  - The size of the data processed in each batch run depends entirely on how long the producer.py and consumer.py scripts have been running and accumulating data in the SQLite database. The batch job processes *all rows* present in the btc_prices table at the time it is triggered. This size grows linearly over time, assuming a relatively constant influx of ticker data.
- **c. Results:**
  - The output of each batch run is a single row DataFrame containing globally aggregated metrics over the entire dataset processed.
  - These metrics (avg_price, max_price, min_price, total_volume, trade_count, etc.) represent a summary of *all* trades stored in the database up to that point.
  - The result is appended as a new CSV file (or overwrites if Spark's default naming is used without partitioning) in the /app/data/batch_output/ directory. The dashboard specifically loads the *latest* file found here.
  - The batch_scheduler.py logs the total execution time of each spark-submit command to /app/data/batch_log.txt, providing a measure of the batch job's performance.

6. Comparison of Streaming & Batch Modes

a. Results and Discussion:

- **Latency:** Streaming provides near real-time insights with results generated shortly after each 1-minute window closes (plus Spark processing overhead and potential watermark delays). Batch mode inherently has higher latency, determined by the scheduling interval (10 minutes) plus the actual batch processing time. The dashboard visually confirms this difference via execution time comparison.
- **Data Granularity:** Streaming offers fine-grained, time-windowed results (1-minute intervals), allowing analysis of trends, volatility bursts, and sentiment shifts *over time*. Batch mode produces a single, global summary statistic representing the entire history processed in that run, losing temporal detail but providing an overall picture. This is evident in the dashboard where streaming shows a price *trend* while batch shows a single average price line.
- **Processing Time & Scalability:** The streaming job processes smaller chunks of data continuously. Its processing time per micro-batch should ideally remain relatively stable if the data rate per window is constant. The batch job's processing time (Batch Aggregation Logic Time, logged in batch_log.txt) is expected to increase as the SQLite database grows over time, as it reads and processes more data in each run. The dashboard compares the *latest* batch execution time with the *average* streaming window duration (a slightly indirect comparison, but illustrative).
- **Resource Usage:** Streaming requires resources (Spark executors/cores) to be continuously available to process incoming data. Batch processing consumes resources intensively during its execution period (every 10 minutes) and might be idle in between (though local[*] mode implies Spark context stays potentially active if the scheduler script keeps running).
- **Use Cases:**
  - *Streaming:* Ideal for real-time dashboards, immediate alerting (e.g., based on volatility_flag or volume_spike), monitoring intra-day trends, and applications requiring low-latency updates.
  - *Batch:* Suitable for end-of-day reporting, generating periodic summaries, historical analysis over large datasets, training machine learning models that require a complete historical view, or when near real-time results are not critical.
- **Consistency & Completeness:** Batch processing operates on a static snapshot of the database at execution time, ensuring consistency for that run. Streaming uses watermarking to handle moderate data lateness, aiming for correctness, but extremely late data might be dropped from its window calculations.
- **Metrics Interpretation:** As seen in the dashboard, metrics like avg_price, total_volume, and trade_count have different meanings. In streaming, they refer to a specific 1-minute window. In batch, they refer to the *entire history* processed in that run.

7. Conclusion

This project successfully implemented and compared two distinct data processing architectures – streaming and batch – for analyzing real-time BTC-USD ticker data using Kafka and Spark.

The streaming pipeline demonstrated its ability to provide low-latency, windowed analytics suitable for real-time monitoring and trend detection. The use of Spark Structured Streaming with Kafka integration proved effective for handling continuous data flow and performing time-based aggregations.

The batch pipeline, scheduled periodically, provided comprehensive historical summaries by processing all accumulated data from a SQLite database. While exhibiting higher latency, it offers a complete overview suitable for periodic reporting or analysis where immediacy is less critical.

The comparison highlighted the fundamental trade-offs: streaming excels in low latency and temporal granularity, while batch provides simpler processing of complete historical datasets at the cost of delay. The choice between streaming and batch fundamentally depends on the specific requirements of the application regarding data freshness, the need for historical context versus time-sensitive trends, and resource allocation strategies. The Streamlit dashboard effectively visualized these differences in outputs and performance characteristics.

8. References

- Coinbase Pro API Documentation: **https://docs.pro.coinbase.com/** (Specifically WebSocket Feed section)
- Apache Kafka Documentation: **https://kafka.apache.org/documentation/**
- Apache Spark Documentation: **https://spark.apache.org/docs/latest/**
  - Structured Streaming Programming Guide: **https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html**
  - Spark SQL, DataFrames and Datasets Guide: **https://spark.apache.org/docs/latest/sql-programming-guide.html**
- kafka-python Documentation: **https://kafka-python.readthedocs.io/**
- websocket-client Documentation: **https://github.com/websocket-client/websocket-client**
- SQLite Documentation: **https://www.sqlite.org/docs.html**
- Xerial SQLite JDBC Driver: **https://github.com/xerial/sqlite-jdbc**
- Streamlit Documentation: **https://docs.streamlit.io/**