VIETNAM NATIONAL UNIVERSITY HCM CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Introduction to Big Data

Spark Streaming

Lab04

Student:

Mai Trần Nguyễn Khang

Uông Minh Nguyên Khôi

Nguyễn Đức Phúc

Nguyễn Tuấn Phong

Instructor:

Nguyễn Ngọc Thảo

Lê Ngọc Thành

Huỳnh Lâm Hải Đăng

Vũ Công Thành

Contents

1 Introduction

1.1 Overview

In this subsection, briefly introduce the purpose and scope of the lab. Mention that the lab focuses on implementing an Extract-Transform-Load (ETL) pipeline for real-time analysis of cryptocurrency price data (BTCUSDT from Binance) using Spark Streaming. We can also briefly mention the key technologies used (Kafka, Spark Structured Streaming, MongoDB). Refer to the "Statements" section on page 2 of the PDF: "In this lab, we and our team are going to implement an Extract-Transform-Load pipeline that performs streamed analysis on time-series data of cryptocurrencies' prices, in particular the price of symbol BTCUSDT from the Binance trading platform." Also, refer to the "Report" grading criteria on page 7: "Overview: source code's structure, components, and implemented methods."

1.2 Team Members

STT	**MSSV**	**Name**
1	22127177	Mai Trần Nguyễn Khang
2	22127212	Uông Minh Nguyên Khôi
3	22127326	Nguyễn Tuấn Phong
4	22127109	Nguyễn Đức Phúc

1.3 Contribution Table

Task	Assigned Team Member(s)	Contributions
Implement Kafka Producer (Extract stage).	Mai Trần Nguyễn Khang	100%
Document Extract stage and include screenshots in report.	Mai Trần Nguyễn Khang	100%
Set up MongoDB.	Mai Trần Nguyễn Khang	100%
Implement Spark Streaming for Moving Statistics (Transform).	Uông Minh Nguyên Khôi	100%
Moving Statistics: Subscribe to Kafka 'btc-price' topic.	Uông Minh Nguyên Khôi	100%
Moving Statistics: Implement sliding windows and calculations.	Uông Minh Nguyên Khôi	100%
Moving Statistics: Handle edge cases and format output.	Uông Minh Nguyên Khôi	100%
Moving Statistics: Publish results to Kafka 'btc-price-moving'.	Uông Minh Nguyên Khôi	100%
Document Moving Statistics and include screenshots in report.	Uông Minh Nguyên Khôi	100%
Implement Spark Streaming for Z-score calculation (Transform).	Nguyễn Tuấn Phong	100%
Z-score: Listen to Kafka 'btc-price' and 'btc-price-moving'.	Nguyễn Tuấn Phong	100%
Z-score: Match records by timestamp and compute Z-scores.	Nguyễn Tuấn Phong	100%
Z-score: Handle edge cases and format output.	Nguyễn Tuấn Phong	100%
Z-score: Publish results to Kafka 'btc-price-zscore'.	Nguyễn Tuấn Phong	100%

Table 1: Team Contribution Table

Task	Assigned Team Member(s)	Contributions
Document Z-score sections and include screenshots in report.	Nguyễn Tuấn Phong	100%
Implement Spark Streaming for Load stage.	Nguyễn Đức Phúc	100%
Load: Subscribe to Kafka 'btc-price-zscore'.	Nguyễn Đức Phúc	100%
Load: Write stream to MongoDB collections ('btc-price-zscore-<window>').	Nguyễn Đức Phúc	100%
Load: Define and document MongoDB schema.	Nguyễn Đức Phúc	100%
Document Load stage and include screenshots in report.	Nguyễn Đức Phúc	100%
Write Report Overview section.	Nguyễn Đức Phúc	100%
Write Report Detailed Explanation section.	Nguyễn Đức Phúc	100%
Write Report Contribution Table section.	Mai Trần Nguyễn Khang	100%
Ensure code is well-documented.	Uông Minh Nguyên Khôi	100%
Include all necessary files and screenshots in submission.	Nguyễn Đức Phúc	100%
Provide code running instructions (if needed).	Nguyễn Đức Phúc	100%
Compile final report and submission ZIP file.	Mai Trần Nguyễn Khang	100%

Table 2: Team Contribution Table

2 Extract Stage

2.1 Objective

The primary objective of the Extract stage is to retrieve real-time price data for the BTCUSDT cryptocurrency symbol from the Binance trading platform. This data serves as the input for the subsequent stages of the ETL pipeline, which involve transformation and loading for analysis.

2.2 Requirements

As specified in the lab requirements, the Extract stage must fulfill the following criteria:

- Implement a Kafka producer to fetch time-series data about the BTCUSDT symbol from

Binance API.

- The fetched data should contain a floating-point value representing the symbol's price.
- Upon receiving a response from the API, the producer must check if the JSON conforms to the expected format: `{"symbol": <a string>, "price": <a floating-point value>}`.
- Insert event-time information (timestamp) associated with when the response was received by the crawler. The timestamp should adhere to ISO8601 standards, preferably in UTC.
- The data fetching and publishing process must run with a frequency of at least once per 100 milliseconds.
- Push the collected records to a Kafka topic named `btc-price`.
- Take screenshots of significant steps and include them in the report with detailed explanations.

2.3 Implementation Details

The provided Python code implements the Kafka producer for the Extract stage, addressing the specified requirements.

2.3.1 Data Fetching

The code utilizes the `requests` library to fetch data from the Binance API endpoint for the BTCUSDT symbol: `https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT`. A timeout is configured to prevent indefinite waiting for a response. Error handling is included to catch potential issues like timeouts and general request exceptions.

2.3.2 Timestamping

Immediately after receiving a successful response from the Binance API, the current timestamp is captured using `datetime.now(timezone.utc)`. This ensures that the event-time recorded is as close as possible to the moment the data is received by the system. The timestamp is then formatted into the ISO8601 string format with millisecond precision and the 'Z' indicator for UTC, as required.

2.3.3 Data Validation and Formatting

The code parses the API response as JSON. It includes validation steps to check if the response is a JSON object and if it contains the expected "symbol" and "price" fields. It specifically verifies that the fetched symbol is "BTCUSDT" and attempts to convert the price string to a floating-point value. Errors during JSON decoding, validation, or type conversion are caught and logged, and the problematic record is skipped.
The valid symbol and price, along with the generated ISO8601 timestamp, are combined into a new JSON payload.

2.3.4 Kafka Production

A Kafka producer is initialized using the `confluent_kafka` library, configured with the Kafka broker address obtained from an environment variable (‘KAFKA_BROKERS‘). The prepared JSON payload, encoded as bytes, is then published to the Kafka topic specified by ‘KAFKA_TOPIC‘, which is configured as ‘btc-price‘. The symbol is used as the message key. A delivery report callback is implemented to log successful deliveries or errors.

2.3.5 Frequency and Loop Control

The main logic runs within a `while run:` loop, controlled by a global flag that can be set to ‘False‘ by signal handlers for graceful shutdown (e.g., upon receiving SIGINT or SIGTERM). Inside the loop, the time taken for fetching and processing is measured. The code then calculates a sleep duration to ensure that the next fetch occurs after the specified ‘Workspace_INTERVAL_SECONDS‘ (0.1 seconds, meeting the 100ms requirement), effectively controlling the fetching frequency. The loop checks the ‘run‘ flag before sleeping to allow for a prompt exit upon receiving a shutdown signal.

2.3.6 Graceful Shutdown

Signal handlers are registered for SIGINT and SIGTERM to catch interruption signals. When a signal is caught, the ‘run‘ flag is set to ‘False‘, which causes the main fetch loop to exit after the current iteration or after sleeping. Before the program exits, the Kafka producer is flushed to attempt delivery of any queued messages within a specified timeout.

2.4 Screenshots

```
(venv) (base) → Extract git:(main) ✗ python main.py
2025-05-07 22:30:31,763 [INFO] Kafka producer connected to brokers: localhost:9092
2025-05-07 22:30:31,763 [INFO] Starting price fetch loop...
```

Figure 1: The message shows that we have successfully connected to the Kafka broker at port `localhost:9092`. It's now fetching and sending data to a Kafka topic `btc-price`.

```
(base) → kafka_2.13-4.0.0 bin/kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic btc-price \
  --from-beginning \
  --max-messages 10
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:31.891Z"}
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:32.008Z"}
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:32.130Z"}
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:32.407Z"}
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:32.531Z"}
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:32.651Z"}
{"symbol": "BTCUSDT", "price": 97009.02, "timestamp": "2025-05-07T15:30:32.774Z"}
{"symbol": "BTCUSDT", "price": 97009.03, "timestamp": "2025-05-07T15:30:32.892Z"}
{"symbol":"BTCUSDT", "price": 97009.03, "timestamp": "2025-05-07T15:30:33.014Z"}
{"symbol": "BTCUSDT", "price": 97009.03, "timestamp": "2025-05-07T15:30:33.128Z"}
Processed a total of 10 messages
```

Figure 2: The output on the Consumer side.

On the command lines: Each JSON object contains the price of the BTCUSDT in time. We run **Kafka Consumer** by using built-in script `kafka-console-consumer.sh`. We connect to the Kafka broker at `localhost:9092`, subsribe for topic `btc-price`, read `from-begining` from the start of the topic and limit by `10 messages`.

On the output lines: We get the JSON objects as: `{ "symbol": "BTCUSDT", "price": <floating-point value>, "timestamp": <timestamp in ISO8601 standards> }`, which satisfied the stage's requirements.

3 Transform Stage

3.1 Objective

The Transform stage is responsible for processing the raw cryptocurrency price data received from the Extract stage to derive meaningful statistics for analysis. This involves two primary computations: calculating moving averages and standard deviations over defined time windows, and subsequently computing Z-scores based on these statistics and the latest price.

3.2 Requirements

The Transform stage has two main components with specific requirements:

3.2.1 Moving Average and Standard Deviation Calculation

- Implement a Spark Structured Streaming program.
- Subscribe to the `btc-price` Kafka topic.
- Use event-time processing to group messages into sliding windows of lengths: 30 seconds, 1 minute, 5 minutes, 15 minutes, 30 minutes, and 1 hour.
- Compute the moving averages and moving standard deviations of prices for each window.
- Handle late data with a tolerance of up to 10 seconds.
- Handle edge cases (e.g., windows with insufficient data).
- Output results in a specified JSON format including the window name, average price, and standard deviation.
- Publish the results to a Kafka topic named `btc-price-moving` using append mode.
- Include screenshots and detailed explanation in the report.

3.2.2 Z-score Calculation

- Implement a separate Spark Structured Streaming program.

- Listen to both the `btc-price` and `btc-price-moving` Kafka topics.
- Match price records from `btc-price` with corresponding moving statistics records from `btc-price-movin` based on timestamp information.
- Compute the Z-score of the price with respect to the moving average and standard deviation for each sliding window provided in the statistics record.
- Handle edge cases (e.g., zero standard deviation).
- Output results in a specified JSON format including the window name and computed Z-score.
- Publish the results to a new Kafka topic named `btc-price-zscore` using append mode.
- Include screenshots and detailed explanation in the report.

3.3 Implementation Details

3.3.1 Moving Statistics (`moving.py`)

The `moving.py` script implements the first part of the Transform stage using Spark Structured Streaming.

1. **Input Stream:** It reads the incoming price data from the `btc-price` Kafka topic. The data, expected to be in JSON format, is parsed into a DataFrame with a defined schema including ‘symbol‘, ‘price‘, and ‘timestamp‘.
2. **Watermarking:** Watermarking is applied to the event-time (*event_timestamp*) with a threshold of 10 seconds (‘LATE_DATA_THRESHOLD‘) to handle late arriving data, allowing Spark to manage state cleanup correctly.
3. **Windowing and Aggregation:** The core logic iterates through the predefined window durations (30s, 1m, 5m, etc.). For each window size, it groups the data by a tumbling window based on the ‘event_timestamp‘ and by ‘symbol‘. Within each window group, it calculates the average price (‘avg_price‘) and the population standard deviation of the price (‘stddev_pop‘).
4. **Schema Structuring and Combining:** The results for each window size are structured into a Spark DataFrame containing the window end time, symbol, window name (as a literal

string), average price, and standard deviation. A check is included to handle potential null or NaN values in the standard deviation by replacing them with 0.0. The DataFrames for all window sizes are then combined into a single stream using ‘union‘.

5. **Output Formatting:** The combined stream is further processed to group results by window end time and symbol, collecting the statistics for all windows that end at the same time into a list. This aggregated data is then formatted into a JSON string conforming to the required output structure, including the window end timestamp (formatted as ISO8601), the symbol, and a list of structures, each containing the window name and its calculated ‘avg_price‘ and ‘std_price‘.

6. **Output Stream:** The final formatted JSON strings are written as the ‘value‘ to the `btc-price-moving` Kafka topic in ‘append‘ mode. A checkpoint location is configured for state management and fault tolerance.

3.3.2 Z-score Calculation (`zscore.py`)

The `zscore.py` script handles the second part of the Transform stage, calculating Z-scores.

1. **Input Streams:** It reads two input streams: the raw price data from the `btc-price` topic and the moving statistics data from the `btc-price-moving` topic. Both streams are parsed from JSON into DataFrames with appropriate schemas. Timestamps from the string format in JSON are parsed into Spark TimestampType.

2. **Watermarking:** Watermarks are applied to both streams based on their respective timestamps (‘price_ts‘ and ‘stats_ts_end‘) to handle late data. The watermark for the statistics stream (‘LATE_DATA_THRESHOLD_STATS‘) is set considering the window durations used in the moving statistics calculation.

3. **Stream Joining:** The two streams are joined based on matching symbols (‘price_symbol = stats_symbol‘) and timestamps. The join condition currently checks if the price timestamp is within 5 seconds of the window end timestamp from the stats stream. For a robust, event-time join, Spark’s join on ‘event_timestamp‘ with appropriate windowing constraints or stateful processing would be required to correctly match a price point with *all* relevant window statistics ending around that time. The provided code includes a debug join condition which

might need refinement to align strictly with joining a price record to *all* window stats relevant to its event time.

4. **Z-score Computation:** After joining, the ‘stats‘ array from the moving statistics record is ‘explode‘d to process each window's statistics individually. For each price record and corresponding window statistic, the Z-score is calculated using the formula: $Z = \frac{X-\mu}{\sigma}$, where X is the current price, μ is the window's average price (‘avg_price‘), and σ is the window's standard deviation (‘std_price‘). Handling for zero or NaN standard deviation is included to prevent division errors, resulting in a Z-score of 0.0 in such cases.

5. **Output Formatting:** The calculated Z-scores are then structured. Results are grouped by the original event timestamp and symbol, collecting all the computed Z-scores for different windows related to that specific price point into a list of structures. Each structure in the list contains the window name and the calculated ‘zscore_price‘. The final output is formatted as a JSON string, including the original event timestamp (as ISO8601 string), the symbol, and the list of Z-score structures.

6. **Output Stream:** The resulting JSON strings, representing the Z-scores for each price point across different windows, are written to the `btc-price-zscore` Kafka topic in ‘append‘ mode. A checkpoint location is also configured for this stream.

3.4 Screenshots

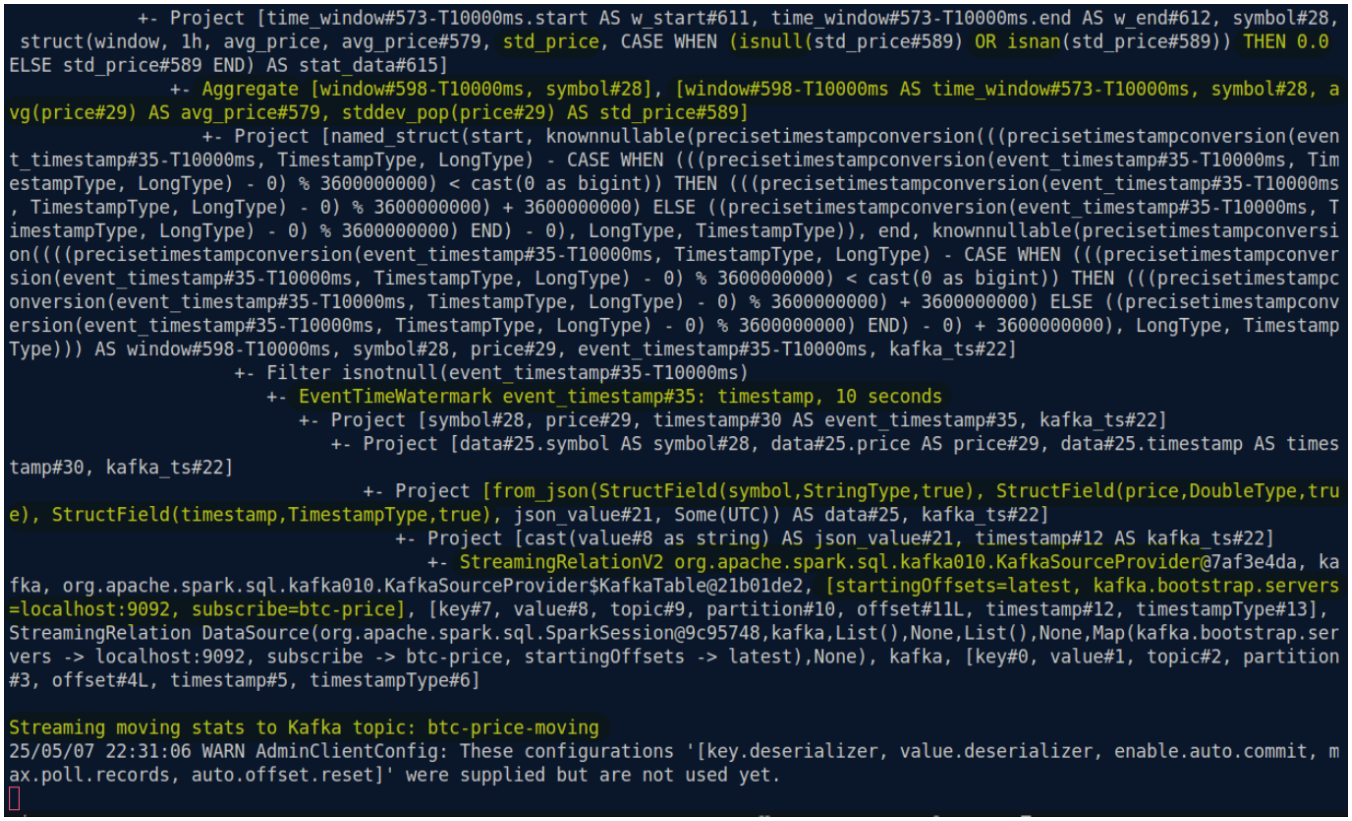

Figure 3: Output log on the Moving Stage.

The above output log has some key points:

1. `StreamingRelationV2 org.apache.spark.sql.kafka010.KafkaSourceProvider`: Use Spark's Kafka integration to read BTC price data at `localhost:9092`, topic `btc-price` from the `latest` offset.

2. `from_json(StructField(symbol, ...), StructField(price, ...), StructField(timestamp, ...))`: Kafka messages are parsed as JSON and converted to a structured schema with 3 fields: `symbol, price and timestamp`.

3. `EventTimeWatermark event_timestamp: timestamp, 10 seconds`: Use the field `timestamp` as event time, and allow late data up to `10 seconds`.

4. `Aggregate [window, symbol], [window, symbol, avg(price), stddev_pop(price)]`: Group by `window, symbol` and calculate `avg_price, std_price` (`std_price` is 0 if NULL or NaN).

5. `Streaming moving stats to Kafka topic: btc-price-moving`: The resulting aggregated data is being written to another Kafka topic called `btc-price-moving`.

Figure 4: Output on Kafka Consumer side after reading from topic `btc-price-moving`.

On the command lines: We run **Kafka Consumer** by using built-in script `kafka-console-consumer.sh`. We connect to the Kafka broker at `localhost:9092`, subsribe for topic `btc-price-moving`, read `from-begining` from the start of the topic and limit by `5 messages`.

On the output lines: Each JSON object contains the time when the window started, the time when the object produced, average price and its standard deviation on corresponding window sizes. We get the JSON objects which satisfied the stage's requirements as:

```
{
  "timestamp": <timestamp in ISO8601 standards>,
  "window_start": <timestamp in ISO8601 standards>,
  "symbol": "BTCUSDT",
  "stats": [
    {
        "window":  <a string among 30s, 1m, 5m, 15m, 30m, 1h>,
        "avg_price": <a floating-point value>,
        "std_price": <a floating-point value>
    },
    {
        "window":  <a string among 30s, 1m, 5m, 15m, 30m, 1h>,
        "avg_price": <a floating-point value>,
        "std_price": <a floating-point value>
    }
    ...
  ]
}
```

Listing 1: JSON object receive from topic btc-price-moving

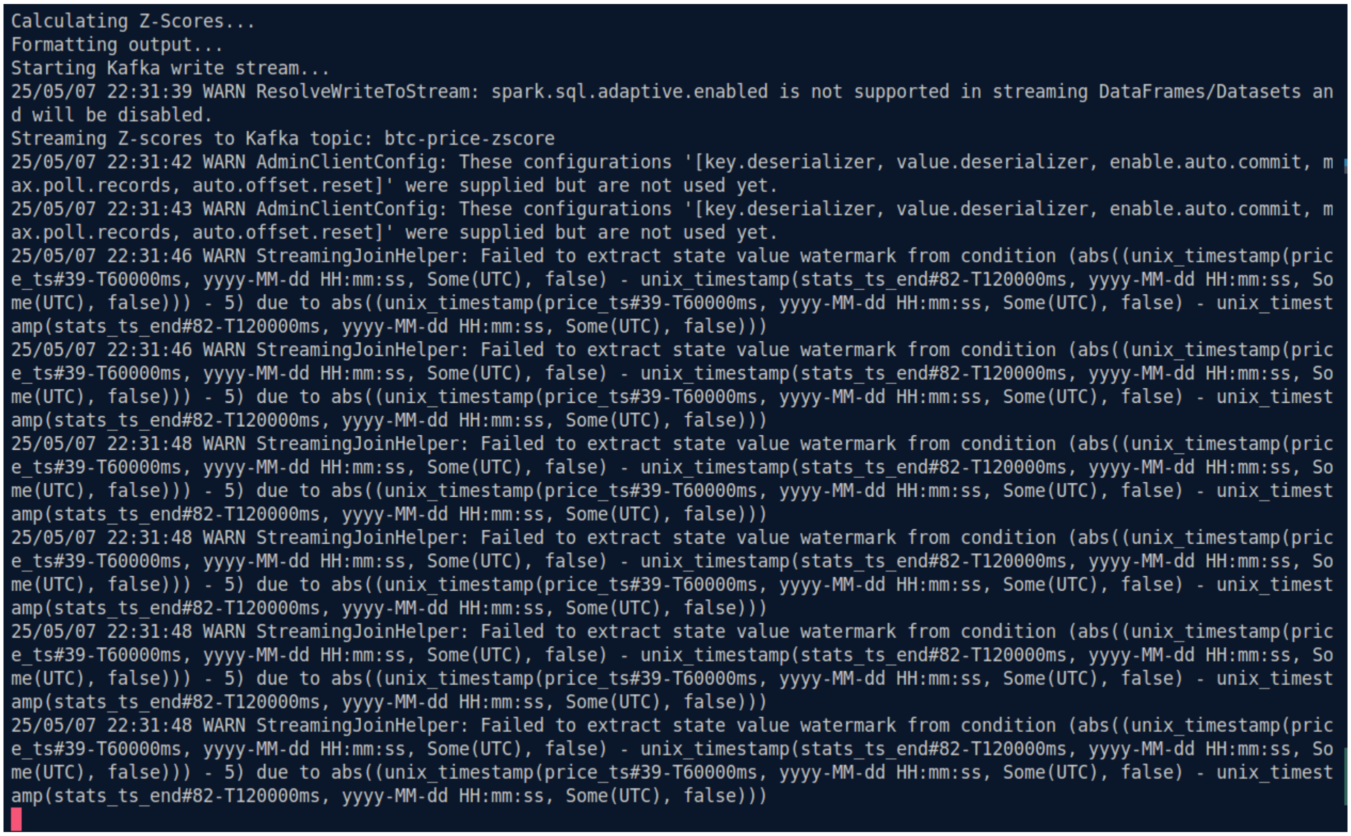

Figure 5: The heading lines indicate that the Z-score is calculated, formatted, and written to Kafka topic `btc-price-zscore`. The rest are system warnings, but they are harmless in our case.

Figure 6: Results of Kafka topic `btc-price-zscore`. Z-score price of BTC/USDT on different window size.

On the command lines: We run **Kafka Consumer** by using built-in script `kafka-console-consumer.sh`. We connect to the Kafka broker at `localhost:9092`, subsribe for topic `btc-price-zscore`, read `from-begining` from the start of the topic and limit by `5 messages`.

On the output lines: Each JSON object shows the Z-score standardization value of the BTCUSDT price on various window sizes. We get the JSON objects which satisfied the stage's requirements as:

```
{
  "timestamp": <timestamp in ISO8601 standards>,
  "symbol": "BTCUSDT",
```

```
  "zscore": [
    {
        "window":  <a string among 30s, 1m, 5m, 15m, 30m, 1h>,
        "zscore_price": <a floating-point value>
    },
    {
        "window":  <a string among 30s, 1m, 5m, 15m, 30m, 1h>,
        "zscore_price": <a floating-point value>
    }
    ...
  ]
}
```

Listing 2: JSON object receive from topic btc-price-moving

4 Load stage

4.1 Objective

The Load stage stores the Z-score streaming results into a NoSQL database (MongoDB) in near real-time. This enables persistent storage for the computed data. Each Z-score window is saved into a separate MongoDB collection, allowing modular and efficient data organization.

4.2 Requirements

Consume the transformed Z-score stream from Kafka topic btc-price-zscore.
Parse incoming JSON strings containing:

- timestamp (ISO format),
- symbol (e.g., BTCUSDT),
- zscores: a list of window, zscore_price.

Flatten the Z-score array, turning one record with multiple windows into multiple rows.
For each Z-score window (e.g., 5s, 1min, 10min), dynamically write to a dedicated MongoDB collection (e.g., btc-price-zscore-5s).

Handle streaming in micro-batches using foreachBatch and persist intermediate data for performance.

Set up Spark checkpointing to enable fault tolerance and state tracking.

4.3 Implementation Details

4.3.1 Reading from Kafka

The script consumes messages from Kafka using Spark Structured Streaming. Kafka value payloads are JSON strings, parsed using a custom schema.

```
zscore_schema = StructType([
    StructField("timestamp", StringType(), True),
    StructField("symbol", StringType(), True),
    StructField("zscores", ArrayType(
        StructType([
            StructField("window", StringType(), True),
            StructField("zscore_price", DoubleType(), True)
        ])
    ), True)
])
```

Listing 3: Z-score JSON schema definition

4.3.2 Streaming Query Setup

- Kafka topic: `btc-price-zscore`
- Starting offset: `latest`
- Watermark: `10 seconds` to tolerate late data

4.3.3 Flattening the Data

Each record is exploded using `explode(zscores)` so that each window's Z-score becomes an individual row. This allows writing to different MongoDB collections by window name.

```
exploded_df = batch_df \
```

```
    .select("event_timestamp", "symbol", F.explode("zscores").alias("zscore_info
")) \
    .select(
        F.col("event_timestamp").alias("timestamp"),
        "symbol",
        F.col("zscore_info.window").alias("window"),
        F.col("zscore_info.zscore_price").alias("zscore_price")
    )
```

Listing 4: Exploding the Z-score array

4.3.4 Writing to MongoDB

The Streaming Query with `foreachBatch` logic is as follows:

- Extract distinct Z-score windows for the batch.
- For each window, filter rows, then write to a collection named `btc-price-zscore-{window}`.
- Uses Spark's MongoDB connector with `.write.format("mongodb")`.

Checkpointing is enabled via:

```
.option("checkpointLocation", checkpoint_dir)
```

4.4 Screenshots

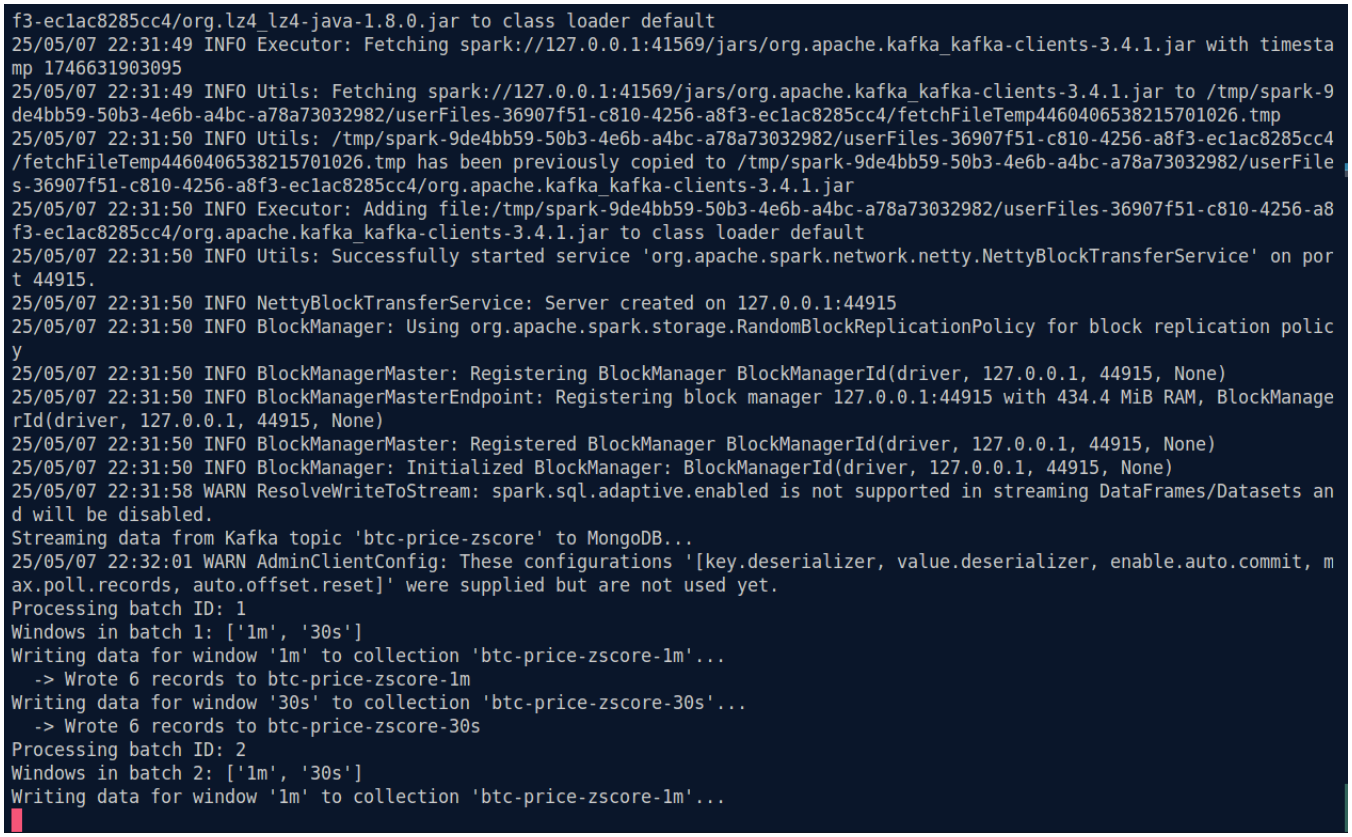

Figure 7: Spark Structured Streaming log output. The system processes incoming Kafka batches, dynamically filters by Z-score window values (e.g., `1m`, `30s`), and writes the filtered results to corresponding MongoDB collections.

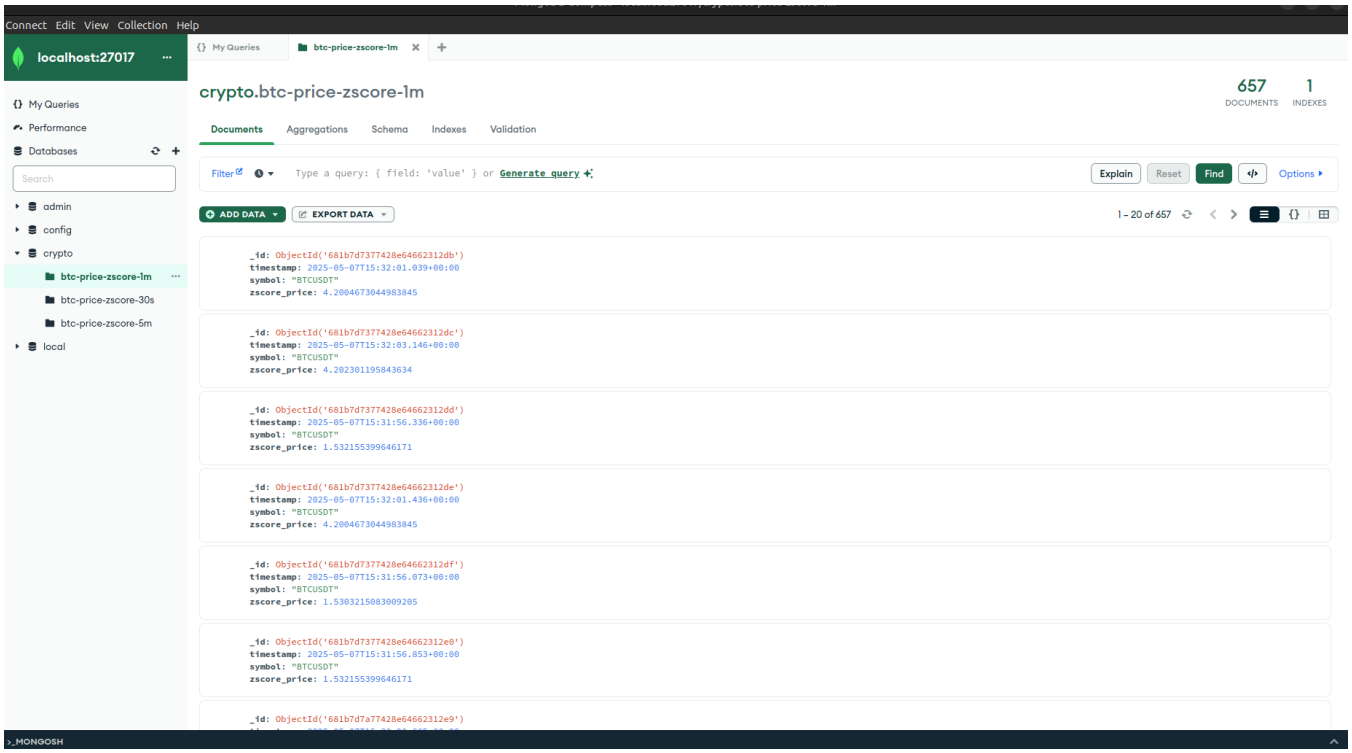

Figure 8: MongoDB Compass showing documents in the `btc-price-zscore-1m` collection. Each document includes timestamped Z-score results, which correspond to the processed batch data ingested from Kafka.

References

[1] Apache Kafka, Apache Kafka Documentation, The Apache Software Foundation

[2] MongoDB, MongoDB Manual, MongoDB Inc.

[3] MongoDB Spark Connector, MongoDB Spark Connector Guide, MongoDB Inc.

[4] Apache Spark Structured Streaming + Kafka Integration, Structured Streaming + Kafka Integration, The Apache Software Foundation

[5] Baeldung, Streaming Data from Kafka to MongoDB using Apache Spark, Baeldung