

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY



Nhập Môn Dữ Liệu Lớn

ASSIGNMENT:

Lab 04: Spark Streaming

Advisor(s): Huỳnh Lâm Hải Đăng
Student(s): Nguyễn Hoàng Anh - 22120014
Lê Bảo - 22120022
Hồ Khánh Duy - 22120076
Nguyễn Lê Nguyên - 22120237

Ho Chi Minh City, 6/2025

Mục lục

I	Giới thiệu	2
1	Về thông tin nhóm và công việc	2
1.1	Thông tin nhóm	2
1.2	Phân chia công việc và mức độ hoàn thành	2
2	Về thông tin bài Lab	2
2.1	Mô tả sơ lược	2
2.2	Mô tả Data Source:	3
3	Setup:	3
3.1	Kafka:	3
3.2	Dựng kafka trên EC2(optional):	5
3.3	Spark kết nối Kafka:	6
3.4	MongoDB:	6
3.5	Spark kết nối MongoDB:	6
II	Báo cáo bài tập	7
1	Extract: Trích xuất dữ liệu giá BTC thời gian thực vào Kafka	7
1.1	Tổng quan: Cấu trúc mã nguồn và các thành phần chính	7
1.2	Giải thích chi tiết phương pháp	7
1.3	Hướng dẫn chạy chương trình	9
2	Transform:	10
2.1	Transform Moving Statistics:	10
2.2	Transform Zscore:	13
3	Load:	15
4	Bonus:	17
4.1	Overview: Mục tiêu và kiến trúc	17
4.2	Detailed Explanation: Giải thích chi tiết phương pháp	18
4.3	Hướng dẫn chạy chương trình	19

Phần I

Giới thiệu

1 Về thông tin nhóm và công việc

1.1 Thông tin nhóm

STT	Họ Tên	MSSV
1	Nguyễn Hoàng Anh	22120014
2	Lê Bảo	22120022
3	Hồ Khánh Duy	22120076
4	Nguyễn Lê Nguyên	22120237

Bảng 1: Thông tin các thành viên trong nhóm

1.2 Phân chia công việc và mức độ hoàn thành

STT	Họ Tên	Công việc được giao	Hoàn thành	Vấn đề gặp phải
1	Nguyễn Hoàng Anh	Transform(moving, zscore), viết report	100%	Không có
2	Lê Bảo	Extract, Transform(moving), viết report	100%	Không có
3	Hồ Khánh Duy	Bonus, Transform(moving), viết report	100%	Không có
2	Nguyễn Lê Nguyên	Load, viết report	100%	Không có

Bảng 2: Công việc nhóm

2 Về thông tin bài Lab

2.1 Mô tả sơ lược

Xây dựng một pipeline xử lý dữ liệu thời gian thực (real-time streaming pipeline) để:

- Thu thập dữ liệu giá Bitcoin (BTCUSDT) từ Binance API.
- Phân tích thống kê theo thời gian (sliding window) bằng Spark Structured Streaming.
- Tính toán độ lệch chuẩn (Z-score) theo thời gian.
- Lưu trữ kết quả vào MongoDB.
- (Bonus) Phân tích xu hướng giá tăng/giảm trong 20s sau mỗi điểm dữ liệu.

Mục đích:

Mục tiêu	Ý nghĩa
Tìm hiểu streaming pipeline	Áp dụng Spark trong xử lý dữ liệu thời gian thực
Làm việc với Kafka	Phân tán dữ liệu giữa các bước
Áp dụng window + stateful + watermark	Nắm vững kỹ thuật xử lý dòng thời gian
Tích hợp MongoDB	Lưu trữ dữ liệu đã phân tích
Hiệu Z-score và thống kê trượt	Phân tích và phát hiện biến động bất thường

Bảng 3: Mục tiêu và ý nghĩa của bài Lab Streaming

2.2 Mô tả Data Source:

Dữ liệu được thu thập thông qua REST API chính thức của sàn giao dịch Binance, sử dụng endpoint:

`https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT`

API này trả về dữ liệu giá hiện tại của cặp giao dịch Bitcoin với USDT (BTCUSDT) dưới dạng JSON với các trường chính bao gồm:

- **symbol:** Tên cặp giao dịch (ví dụ: "BTCUSDT")
- **price:** Giá hiện tại của Bitcoin theo USDT, dạng chuỗi số thực (string)

Trong pipeline, giá trị **price** được chuyển đổi sang kiểu số thực (float) để thuận tiện cho việc tính toán và phân tích.

Ngoài ra, mỗi bản ghi dữ liệu được bổ sung thêm trường **event_time** đánh dấu thời gian UTC (dưới dạng ISO 8601 với mili giây), thể hiện thời điểm lấy dữ liệu giá.

Dữ liệu liên tục được lấy mới mỗi 100ms và gửi vào Kafka topic **btc-price** để phục vụ cho các bước xử lý tiếp theo trong pipeline streaming.

3 Setup:

3.1 Kafka:

Để thiết lập Kafka phục vụ cho quá trình **streaming dữ liệu**, chúng ta thực hiện các bước sau:

1. Tải Kafka:

- Truy cập trang chủ Kafka tại: <https://kafka.apache.org/downloads>
- Chọn phiên bản Kafka phù hợp. Nhóm tải theo dạng binary downloads và tải phiên bản Scala 2.13.

3.9.1

- Released May 21, 2025
- [Release Notes](#)
- Docker image: [apache/kafka:3.9.1](#).
- Docker Native image: [apache/kafka-native:3.9.1](#).
- Source download: [kafka-3.9.1-src.tgz](#) ([asc](#), [sha512](#))
- Binary downloads:
 - Scala 2.12 - [kafka_2.12-3.9.1.tgz](#) ([asc](#), [sha512](#))
 - Scala 2.13 - [kafka_2.13-3.9.1.tgz](#) ([asc](#), [sha512](#))

- Giải nén file ZIP vào một thư mục cụ thể, ví dụ: C:\kafka

2. Khởi động Zookeeper: (Kafka yêu cầu Zookeeper để hoạt động)

Mở cmd lên và chạy lệnh:

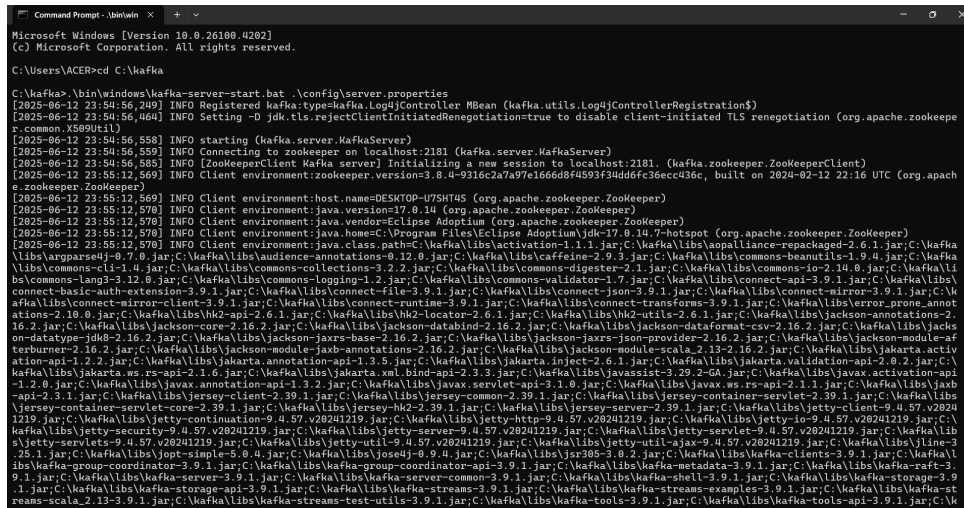
```
cd C:\kafka
.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```



3. Khởi động Kafka:

Mình sẽ khởi động Kafka ở một cmd khác và chạy lệnh:

```
.\bin\windows\kafka-server-start.bat .\config\server.properties
```



4. (Tuỳ chọn) Tạo các topic Kafka cần thiết: Nếu cấu hình auto.create.topics.enable=false, cần tạo topic thủ công:

```
.\bin\windows\kafka-topics.bat --create --topic btc-price \
--bootstrap-server localhost:9092 \
--partitions 1 --replication-factor 1

.\bin\windows\kafka-topics.bat --create --topic btc-price-moving \
```

```
--bootstrap-server localhost:9092 \  
--partitions 1 --replication-factor 1
```

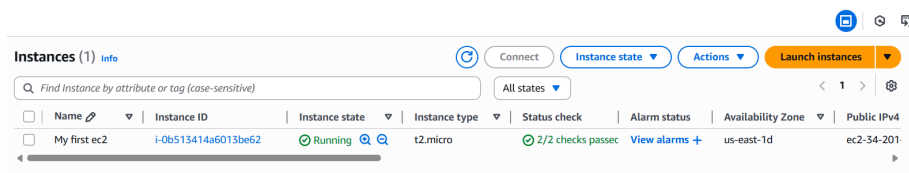
5. Kiểm tra hoạt động của topic (tùy chọn):

Mở một consumer để xem dữ liệu được gửi đến topic:

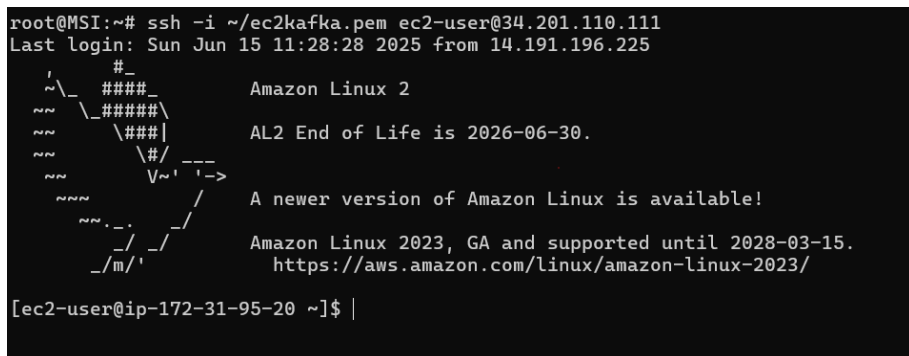
```
.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 ^  
--topic btc-price --from-beginning
```

3.2 Dựng kafka trên EC2(optional):

Trước tiên cần đăng kí tài khoản AWS và đi vào AWS console. Sau đó thực hiện dựng 1 máy ảo EC2 (type t2.micro vì nằm trong diện free tier).



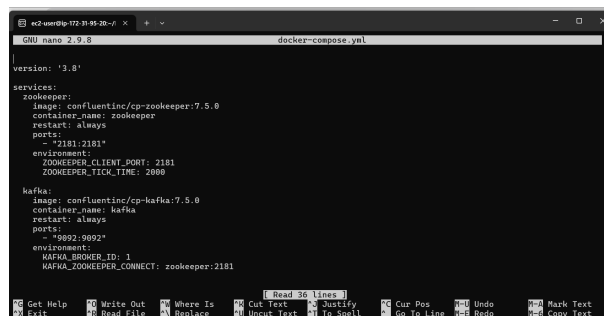
Thực hiện *ssh* vào máy ảo đã tạo.



Thực hiện cài đặt *docker* trên máy ảo.

```
# Cài Docker  
sudo apt install -y docker.io  
  
# Cho phép chạy Docker không cần sudo  
sudo usermod -aG docker $USER  
newgrp docker  
  
# Cài Docker Compose  
sudo apt install -y docker-compose  
  
# Kiểm tra lại  
docker --version  
docker-compose --version
```

Cấu hình *docker-compose.yml* để dựng kafka trên máy ảo ec2 này.



```
version: '3.8'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: zookeeper
    restart: always
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    container_name: kafka
    restart: always
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

Khi khởi chạy thì ta sẽ nhập lệnh `docker exec -it kafka bash` để mở **container** của kafka và thực hiện kết nối tạo *topic* từ bên ngoài.

Việc dựng kafka ở trên máy ảo mục đích ban đầu là để có thể tận dụng tài nguyên trên máy cá nhân của các thành viên thực hiện các job *transform*, kết nối giữa các giai đoạn do nhiệm vụ được phân công cho nhiều người và thuận lợi cho việc *scale* sau này.

3.3 Spark kết nối Kafka:

Spark chỉ cần chạy ở local không cần thiết phải chạy chung **Network** với kafka.

Kết nối từ spark đến kafka thì cần quan tâm **kafka.bootstrap.servers**. Nếu kafka chạy local với spark thì là `localhost:9092` còn nếu kafka trên cloud thì nó sẽ là `<PUBLIC_IP>:9092` với 9092 là port giao tiếp mặc định khi cấu hình của kafka.

Để chạy 1 spark job có kết nối với kafka thì cần lưu ý mô-đun kết nối của chúng chưa được đóng gói sẵn trong spark core. Vì thế khi chạy spark job chúng ta cần phải thêm tham số

```
--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.5
```

Nếu không khi chạy sẽ bị lỗi **ClassNotFoundException**.

3.4 MongoDB:

- Đảm bảo **firewall MongoDB Atlas** mở cổng cho địa chỉ IP máy chạy Spark hoặc thiết lập “Allow access from anywhere” để chạy.

3.5 Spark kết nối MongoDB:

Để Spark có thể ghi dữ liệu vào MongoDB, cần Cài đặt MongoDB Spark Connector:

- **Thêm thư viện MongoDB Spark Connector khi chạy chương trình:**
 - Khi chạy chương trình, sử dụng tùy chọn `-packages` để Spark tự động tải thư viện:
`--packages org.mongodb.spark:mongo-spark-connector_2.12:10.5.0`

Phần II

Báo cáo bài tập

1 Extract: Trích xuất dữ liệu giá BTC thời gian thực vào Kafka

1.1 Tổng quan: Cấu trúc mã nguồn và các thành phần chính

Chương trình được thiết kế để lấy dữ liệu giá Bitcoin (BTC/USDT) từ Binance API một cách liên tục và gửi vào Kafka topic tên là `btc-price` theo thời gian thực.

Kiến trúc mới áp dụng mô hình xử lý song song với hai luồng (thread) chính để tối ưu hóa hiệu suất: một luồng chuyên lấy dữ liệu từ API và một luồng chuyên gửi dữ liệu vào Kafka. Việc này giúp loại bỏ độ trễ và đảm bảo dòng dữ liệu luôn thông suốt.

Các thành phần chính:

- **Thư viện sử dụng:**

- `requests`: Gửi HTTP GET request đến Binance API.
- `json`: Serialize dữ liệu thành định dạng JSON.
- `kafka.KafkaProducer`: Gửi dữ liệu vào Kafka topic.
- `datetime`: Tạo timestamp UTC chính xác đến mili giây.
- `time`: Điều khiển tần suất hoạt động của các vòng lặp.
- `threading, queue`: Quản lý xử lý song song, tách biệt việc lấy và gửi dữ liệu.
- `os`: Đọc cấu hình từ biến môi trường.
- `logging`: Ghi lại hoạt động của chương trình một cách có cấu trúc.

- **Cấu trúc hàm và luồng chính:**

- `get_kafka_producer()`: Hàm khởi tạo và trả về một instance duy nhất của `KafkaProducer` (Singleton Pattern).
- `price_fetcher()` (Luồng 1): Chạy trong một luồng riêng, liên tục lấy giá từ Binance API và đưa vào một hàng đợi (Queue) để luồng gửi xử lý.
- `message_sender()` (Luồng 2): Vòng lặp chính, lấy dữ liệu từ hàng đợi, thêm timestamp, và gửi vào Kafka.

- **Kafka topic**: `btc-price` là topic nơi dữ liệu sẽ được gửi đến.

1.2 Giải thích chi tiết phương pháp

- **Khởi tạo Kafka Producer:**

- Producer được khởi tạo thông qua hàm `get_kafka_producer` để đảm bảo chỉ có một kết nối duy nhất trong suốt vòng đời của ứng dụng.
- Địa chỉ Kafka broker được cấu hình linh hoạt qua biến môi trường `KAFKA_BOOTSTRAP_SERVERS`, với giá trị mặc định là `localhost:9092`.
- Producer được cấu hình với các tham số cần thiết như cơ chế tự động thử lại (`retries=3`) và xác nhận từ broker (`acks=1`) để đảm bảo độ tin cậy khi gửi tin.


```
def get_kafka_producer():
    global producer
    if producer is None:
        kafka_servers = os.getenv('KAFKA_BOOTSTRAP_SERVERS', 'localhost:9092')
        logger.info(f"Connecting to Kafka at: {kafka_servers}")

        producer = KafkaProducer(
            bootstrap_servers=kafka_servers,
            value_serializer=lambda v: json.dumps(v).encode('utf-8'),
            buffer_memory=33554432,
            retries=3,
            acks=1
        )
    return producer
```

- **Luồng lấy giá BTC từ Binance API (price_fetcher):**

- Hàm này chạy trên một luồng (**thread**) riêng biệt để không làm gián đoạn quá trình gửi dữ liệu.
- Nó liên tục gọi đến endpoint của Binance API:
`https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT`
- Dữ liệu giá sau khi lấy về sẽ được đưa vào một hàng đợi (**Queue**). Hàng đợi này hoạt động như một bộ đệm, giúp tách biệt và điều phối dữ liệu giữa luồng lấy và luồng gửi.

```
def price_fetcher():
    global latest_price
    while True:
        try:
            price_data = get_price()
            if price_data:
                with price_lock:
                    latest_price = price_data
                if not price_queue.empty():
                    price_queue.get_nowait()
                price_queue.put_nowait(price_data)
        except Exception as e:
            logger.error(f"Price fetcher error: {e}")
            time.sleep(0.05)
```

- **Luồng gửi dữ liệu vào Kafka (message_sender):**

- Đây là vòng lặp chính của chương trình. Nó lấy dữ liệu giá từ **Queue** mà luồng **price_fetcher** đã cung cấp.
- Hàm **create_message_with_timestamp** được gọi để thêm trường **event_time** (timestamp UTC, chính xác đến mili giây) vào dữ liệu.
- Dữ liệu hoàn chỉnh sau khi xử lý sẽ được gửi vào topic **btc-price** đều đặn mỗi 100ms.

- Dữ liệu cuối cùng có dạng:

```
{  
  "symbol": "BTCUSDT",  
  "price": 67321.85,  
  "event_time": "2025-06-13T09:30:15.100Z"  
}
```

```
def message_sender():  
    while True:  
        try:  
            current_price = None  
            if not price_queue.empty():  
                current_price = price_queue.get_nowait()  
            else:  
                with price_lock:  
                    current_price = latest_price  
            if current_price:  
                message = create_message_with_timestamp(current_price)  
                if message:  
                    get_kafka_producer().send('btc-price', value=message)  
            else:  
                dummy_message = {  
                    "symbol": "BTCUSDT",  
                    "price": 0.0,  
                    "event_time": datetime.now(timezone.utc)\  
                        .isoformat(timespec='milliseconds')\  
                        .replace('+00:00', 'Z')  
                }  
                get_kafka_producer().send('btc-price', value=dummy_message)  
        except Exception as e:  
            logger.error(f"Message sender error: {e}")  
  
        time.sleep(0.1)
```

1.3 Hướng dẫn chạy chương trình

- **Yêu cầu môi trường:**

- Kafka đã cài đặt và đang chạy tại localhost:9092 (hoặc địa chỉ khác được cấu hình qua biến môi trường).
- Cài đặt các thư viện cần thiết:

```
pip install kafka-python requests
```

- **Chạy Producer:**

- Mở terminal và chạy lệnh: `python extract.py`.

2 Transform:

2.1 Transform Moving Statistics:

Cấu hình *sparkSession*

```
spark = SparkSession.builder \
    .appName("MovingStats_Tumbling_WithJoins") \
    .config("spark.sql.shuffle.partitions", "4") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") \
    .config("spark.driver.memory", "1g") \
    .config("spark.executor.memory", "1g") \
    .config("spark.driver.maxResultSize", "512m") \
    .getOrCreate()

spark.sparkContext.setLogLevel("OFF")
```

- `spark.serializer = org.apache.spark.serializer.KryoSerializer`

KryoSerializer là bộ tuần tự hoá hiệu năng cao hơn nhiều so với `JavaSerializer` mặc định. Lợi ích:

- Giảm kích thước dữ liệu đã tuần tự hoá.
- Tăng tốc độ truyền dữ liệu giữa *driver* *executor* (hoặc giữa các *task*).
- Thích hợp cho các ứng dụng *streaming* hoặc xử lý nhiều *object* phức tạp.

- `spark.sql.adaptive.enabled = true`

Kích hoạt **Adaptive Query Execution (AQE)** — Spark sẽ điều chỉnh *execution plan* ngay trong thời gian chạy dựa trên thống kê dữ liệu thực tế. Lợi ích:

- Tự động chọn chiến lược *join* phù hợp (broadcast, shuffle, ...).
- Giảm thiểu hiện tượng *data skew* hoặc chọn nhầm phương pháp *join*.
- Tăng hiệu suất truy vấn khi phân bố dữ liệu không đồng đều.

- `spark.sql.adaptive.coalescePartitions.enabled = true`

Tự động gộp (coalesce) các *partition* quá nhỏ sau giai đoạn *shuffle*.

- Giảm overhead vì phải quản lý hàng trăm *partition* tí hon (tốn chi phí *task scheduling*).

- `spark.sql.shuffle.partitions = 4`

Giảm số *partition* sinh ra sau *groupBy*, *join*, *window*, *agg* từ mặc định 200 → 4.

- Trong môi trường *local* hoặc bộ dữ liệu nhỏ (ví dụ lab), 200 *partition* là dư thừa, gây tốn bộ nhớ và *context switching*.
- Đặt về 4 giúp xử lý nhanh hơn mà không ảnh hưởng đến tính đúng.

So với cấu hình đơn giản ban đầu thì cấu hình như trên nhanh hơn rất nhiều, giảm việc *compute* từ vài phút đến còn vài chục giây.

```
spark = SparkSession.builder \
    .appName("MovingStats_Tumbling_WithJoins") \
    .getOrCreate()
spark.sparkContext.setLogLevel("WARN")
```

Tiếp theo là bước thực hiện đọc dữ liệu từ topic **"btc-price"** và chuyển nó sang **json**.

```
df_raw = (
    spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "btc-price")
        .option("startingOffsets", "latest")
        .load()
        .selectExpr("CAST(value AS STRING) as json_str")
        .select(from_json("json_str", schema).alias("data"))
        .select("data.symbol", "data.price", "data.event_time")
)
```

Tạo các luồng thống kê theo 6 mốc thời gian và thực hiện tính toán. *GroupBy* có **"symbol"** chủ yếu là để mở rộng sau này nếu không chỉ tính cho mình cổ phiếu **BTCUSDT**

```
stats30s = (
    df_raw
        .withWatermark("event_time", "10 seconds")
        .groupBy(window("event_time", "30 seconds", "10 seconds"), col("symbol"))
        .agg(avg("price").alias("avg_30s"), stddev("price").alias("std_30s"))
        .select(
            col("symbol"),
            col("window.end").alias("timestamp"),
            col("avg_30s"),
            col("std_30s")
        )
)
```

Gắn watermark tại statsDF (10 s)

Mục đích	Lý do chọn event_time & 10 s
Giới hạn <i>state</i> ngay tại phép tính cửa sổ	Sau <code>groupBy(window,...)</code> Spark sinh thêm cột <code>window.{start,end}</code> metadata watermark trên <code>event_time</code> sẽ mất tác dụng nếu gắn trước. Do đó phải gắn trực tiếp vào mỗi <code>statsDF</code> để Spark biết khi nào xả <i>state</i> của cửa sổ vừa khép.
Độ trễ bảo vệ tối đa	Các bản tin giá bị lệch mạng thường $\leq 2-3s$; đặt 10 s đủ an toàn nhưng vẫn bảo đảm độ trễ $< 10s$.
Đồng bộ giữa sáu kích thước	Tất cả <code>statsDF</code> đều dùng watermark 10 s bản ghi của mọi cửa sổ sẽ khép cùng lúc và sẵn sàng cho phép join.

Nếu chỉ gắn watermark duy nhất ở `df_raw`, Spark phải giữ *state* cửa sổ tới hơn 1h (với window 1h) trước khi xả bộ nhớ phình lên và kết quả ra chậm. Việc “đánh lại” 10 s trên từng `statsDF` khắc phục triệt để.

Cửa sổ trượt — slide 10 s

Bởi $LCM = 10s$, mọi `window_end` của sáu kích thước “rơi” vào các mốc

00:00:00, 00:00:10, 00:00:20, ...

Chỉ cần so khóa (`symbol`, `timestamp`) khi join, không cần “đồng hồ” phức tạp. Trượt 10 s bảo đảm *dashboard* cập nhật đều ~ 6–10 bản tin / phút, nhẹ hơn `slide 1s`.

Chọn 10s tức sẽ **no tumbling**.

Chọn `timestamp = window_end` vì

- **Định nghĩa rõ ràng `window_end`** chính là mốc thời gian đóng của cửa sổ trượt (t), mô tả rõ: “số liệu này đã tính xong cho khoảng $(t - window_duration, t]$ ”.
- **Đồng bộ hoá sáu kích thước:** Cả sáu cửa sổ đều dùng bước trượt 10 s; do đó, tại mỗi lần phát dữ liệu (mỗi 10 s) chúng đều có chung giá trị `window_end`. Nhờ vậy có thể *inner-join* trên khoá (`symbol`, `timestamp`) mà không cần logic khớp phức tạp.
- **Tại sao không dùng `window_start`:** Nếu sử dụng `window_start` mỗi độ dài cửa sổ sẽ có điểm bắt đầu khác nhau dẫn đến khó bảo đảm đồng bộ giữa các kích thước.

Tính trung bình & độ lệch chuẩn

```
avg("price").alias("avg_30s"),  
stddev("price").alias("std_30s")
```

Spark dùng hàm built-in, độ phức tạp $O(1)$ theo kích thước cửa sổ.

Thực hiện kết hợp các luồng thống kê theo cửa sổ

```
joined = (  
  stats30s  
    .join(stats1m, on=["symbol", "timestamp"], how="inner")  
    .join(stats5m, on=["symbol", "timestamp"], how="inner")  
    .join(stats15m, on=["symbol", "timestamp"], how="inner")  
    .join(stats30m, on=["symbol", "timestamp"], how="inner")  
    .join(stats1h, on=["symbol", "timestamp"], how="inner")  
    .select(  
      col("symbol"),  
      col("timestamp"),  
      array(  
        struct(lit("30s").alias("window"),  
              col("avg_30s").alias("avg_price"),  
              col("std_30s").alias("std_price")),  
        struct(lit("1m").alias("window"),  
              col("avg_1m").alias("avg_price"),  
              col("std_1m").alias("std_price")),  
        struct(lit("5m").alias("window"),  
              col("avg_5m").alias("avg_price"),  
              col("std_5m").alias("std_price")),  
        struct(lit("15m").alias("window"),  
              col("avg_15m").alias("avg_price"),  
              col("std_15m").alias("std_price")),  
        struct(lit("30m").alias("window"),  
              col("avg_30m").alias("avg_price"),  
              col("std_30m").alias("std_price")),  
        struct(lit("1h").alias("window"),  
              col("avg_1h").alias("avg_price"),  
              col("std_1h").alias("std_price"))  
      ).alias("windows")  
    )  
)
```

Ở đây, nhóm đã chọn kết hợp tất cả lại bằng join (inner) bằng việc so sánh *symbol* và *timestamp*. Sau đó format các thông tin *avg*, *std* bằng **array**, **struct** cho đúng với yêu cầu.

Tiêu chí	Inner-join trên khóa	$Union \rightarrow groupBy \rightarrow collect_list$
Độ trễ (latency)	Bản ghi được phát ngay khi sáu statistics cùng mốc timestamp xuất hiện trong một micro-batch.	Phải gom đủ sáu bản ghi rồi đợi watermark xác nhận “không còn thêm” \Rightarrow thường chậm hơn \geq watermark.
Kích thước state	<i>Join</i> chỉ giữ các cặp khóa chưa khớp ; nhờ watermark 10 s, state < 10 s dữ liệu \Rightarrow RAM nhỏ.	collect_list duy trì mảng tới khi đủ 6 phần tử \Rightarrow state lớn gấp 6 \times , phình đáng kể với nhiều symbol.
Đơn giản schema	Sau <i>join</i> đã có các cột avg_30s ... avg_1h; chỉ cần array(struct(...)) \rightarrow JSON.	Phải <i>groupBy</i> rồi collect_list, sau đó filter(size=6); thêm bước xử lý.
Khả năng sai sót	Không lo thiếu / thừa phần tử – <i>join</i> “tự động” bỏ key nếu thiếu bất kỳ cửa sổ.	Nếu quên filter hoặc filter sai, có thể xuất record thiếu cửa sổ.
Tối ưu vật lý	Spark dùng state join đã tinh chỉnh (hash-join + watermark pruning).	Aggregation thứ hai khó tối ưu, dễ rơi vào <i>shuffle-state</i> lớn.

Lí do chọn join thay vì union + collect là vì những tiêu chí sau:

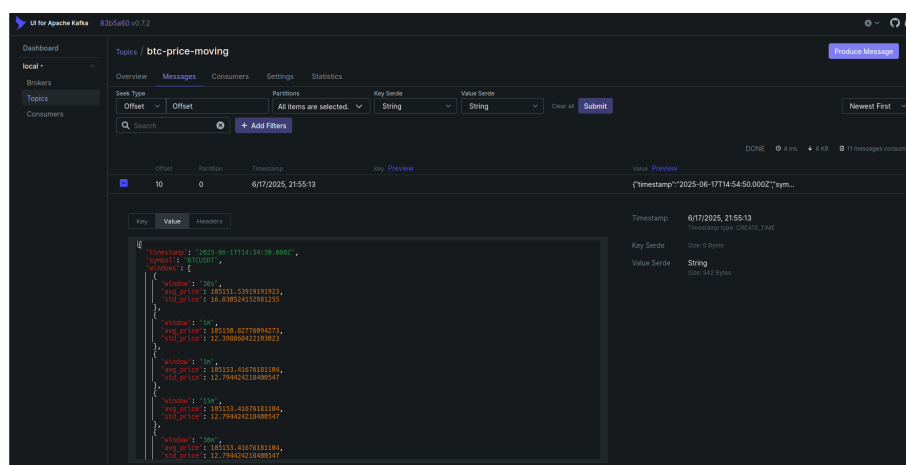
Cuối cùng là chuyển giá trị vừa tính toán tổng hợp được về đúng format và gửi vào topic "btc-price-moving" trong *kafka*.

```
output = joined.select(
    to_json(struct(col("timestamp"), col("symbol"), col("windows"))).alias("value")
)

query = output.writeStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", OUTPUT_TOPIC) \
    .option("checkpointlocation", "c:\users\admin\checkpoint\console") \
    .outputMode("append") \
    .start()

query.awaitTermination()
```

Kết quả của bước này thì ta sẽ được như sau:



2.2 Transform Zscore:

Trước tiên là đọc dữ liệu từ 2 topic trong kafka là "btc-price", "btc-price-moving"

```
df_raw = (
    spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "btc-price")
        .option("startingOffsets", "latest")
        .load()
        .selectExpr("CAST(value AS STRING) as json_str")
        .select(from_json("json_str", schema).alias("data"))
        .select("data.symbol", "data.price", "data.event_time")
        .withWatermark("event_time", "1 minute")
)

df_moving = (
    spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "btc-price-moving")
        .load()
        .selectExpr("CAST(value AS STRING) AS json_str")
        .select(from_json(col("json_str"), movingStatsSchema).alias("data"))
        .select("data.*")
        .withWatermark("timestamp", "30 seconds")
)
```

Ở đây chúng ta cần phải đặt watermark vì những dataframe muốn tham gia join thì điều đó là bắt buộc. (việc điều chỉnh thông số thời gian khác nhau sẽ phụ thuộc việc dữ liệu bên luồng nào đến trễ hơn và bên kia sẽ đợi)

```
raw = df_raw.alias("raw")
mov = df_moving.alias("mov")

joined = raw.join(mov, on="symbol", how="inner") \
    .where(
        (col("event_time") >= expr("mov.timestamp - INTERVAL 0.1 SECOND")) &
        (col("event_time") <= expr("mov.timestamp + INTERVAL 0.1 SECOND"))
    )
```

Thực hiện **join** 2 luồng lại với nhau và **filter** lọc ra những record có cùng *timestamp* và độ chênh lệch được chấp nhận là 100ms. Mục đích cho sự chênh lệch là bắt cặp chính xác hơn giữa 2 luồng.

```
result = joined.withColumn(
    "zscores",
    expr("""
        transform(
            windows,
            w -> struct(
                w.window AS window,
                (price - w.avg_price) / w.std_price AS zscore_price
            )
        )
    """)
).select(
    col("event_time").alias("timestamp"),
    col("symbol"),
    col("zscores")
)
```

Transform lặp qua mảng windows và thêm trường *zscore_price*.

Độ lệch chuẩn bằng 0 (trường hợp ngoại lệ) đã được thay 0 ở pha moving-stats nên không gây chia-0.

```
output = result.select(
    to_json(struct("timestamp", "symbol", "zscores")).alias("value")
)

output.writeStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "btc-price-zscore") \
    .option("checkpointLocation", "C:\\users\\admin\\checkpoint_1\\console") \
    .outputMode("append") \
    .start()

spark.streams.awaitAnyTermination()
```

Cuối cùng là chuyển về dạng chuẩn và gửi vào topic "**btc-price-zscore**"
Dưới đây là kết quả chi tiết sau quá trình *transform zscore*

```

- id: ObjectID("684fd938a6fe8c8666aac686")
- timestamp: 2025-06-16T08:42:00.000+00:00
- symbol: "BTCUSD"
- zscores: Array (6)
  - 0: Object
    - window: "30s"
    - zscore_price: 1.0461065161506193
  - 1: Object
    - window: "1m"
    - zscore_price: 0.8683683417147945
  - 2: Object
    - window: "5m"
    - zscore_price: -0.13821148461171373
  - 3: Object
    - window: "15m"
    - zscore_price: 0.862508269464267
  - 4: Object
    - window: "30m"
    - zscore_price: 0.4085298751781386
  - 5: Object
    - window: "1h"
    - zscore_price: 0.023321551053145984

```

3 Load:

3.1 Tạo SparkSession

```
# 1. Tạo SparkSession
spark = SparkSession.builder \
    .appName("KafkaZScoreToMongo") \
    .config("spark.mongodb.write.connection.uri", "mongodb+srv://nguyenlenguyen1712004:nguyen1712004@cluster0.mp61piv.mongodb.net/crypto?retryWrites=true") \
    .getOrCreate()
```

- Tạo một SparkSession để làm điểm khởi đầu cho việc xử lý dữ liệu Spark.
- appName("KafkaZScoreToMongo"): Tên ứng dụng Spark (dùng để hiển thị trong UI của Spark).
- .config(...): Khai báo URI kết nối đến MongoDB Atlas bằng mongo-spark-connector.
- URI dạng mongodb+srv://... là URI Atlas Cloud dùng account và mật khẩu.

3.2 Định nghĩa schema cho dữ liệu z-score

```
# 2. Schema tương ứng với dữ liệu zscore
zscore_schema = StructType([
    StructField("timestamp", TimestampType()),
    StructField("symbol", StringType()),
    StructField("zscores", ArrayType(
        StructType([
            StructField("window", StringType()),
            StructField("zscore_price", DoubleType())
        ])
    ))
])
```

- Schema định nghĩa cấu trúc JSON mà Spark sẽ parse

3.3 Đọc dữ liệu từ Kafka

```
# 3. Đọc dữ liệu từ Kafka topic "btc-price-zscore"
df = (
    spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "3.88.194.149:9092")
        .option("subscribe", "btc-price-zscore")
        .option("startingOffsets", "latest")
        .load()
        .selectExpr("CAST(value AS STRING) AS json_str")
        .select(from_json(col("json_str"), zscore_schema).alias("data"))
        .select("data.*")
)
```

- `format("kafka")`: Cho Spark biết sẽ đọc dữ liệu từ Kafka.
- `.option("kafka.bootstrap.servers", ...)`: Kafka broker (host:port).
- `.option("subscribe", ...)`: Tên Kafka topic muốn lắng nghe.
- `.option("startingOffsets", "latest")`: Đọc dữ liệu mới nhất từ thời điểm chạy trở đi.
- Kafka truyền dữ liệu ở dạng nhị phân (value là bytes) -> cần ép kiểu sang STRING.
- `from json(...)`: Chuyển chuỗi JSON thành cột có kiểu dữ liệu (theo schema đã định nghĩa).
- `.select("data.*")`: Lấy ra từng trường cụ thể

3.4 Ghi dữ liệu vào MongoDB

```
# 4. Ghi dữ liệu vào MongoDB
df.writeStream \
  .format("mongodb") \
  .option("checkpointLocation", "/home/nguyen1712004/checkpoint/mongo-zscore") \
  .option("spark.mongodb.write.connection.uri", "mongodb+srv://nguyenlenguyen1712004:nguyen1712004@cluster0.mp6lpiv.mongodb.net/crypto") \
  .option("spark.mongodb.write.database", "crypto") \
  .option("spark.mongodb.write.collection", "btc-price-zscore") \
  .outputMode("append") \
  .start()

spark.streams.awaitAnyTermination()
```

- Ghi dữ liệu đầu ra vào MongoDB.
- `checkpointLocation`: Vị trí lưu checkpoint để job Spark có thể khôi phục.
- `outputMode("append")`: Ghi thêm bản ghi mới vào MongoDB mà không ghi đè.
- `spark.streams.awaitAnyTermination()`: Giữ cho chương trình Spark tiếp tục chạy không bị thoát cho đến khi bị dừng bằng tay.

3.5 Kết quả:

```
{
  "_id": ObjectId("684fd91ba6fe8c9666aac683"),
  "timestamp": 2025-06-16T08:41:40.100+00:00,
  "symbol": "BTCUSD",
  "zscore": Array (6)
}
```

```
{
  "_id": ObjectId("684fd91ba6fe8c9666aac684"),
  "timestamp": 2025-06-16T08:41:49.900+00:00,
  "symbol": "BTCUSD",
  "zscore": Array (6)
}
```

```
{
  "_id": ObjectId("684fd938a6fe8c9666aac686"),
  "timestamp": 2025-06-16T08:42:00.000+00:00,
  "symbol": "BTCUSD",
  "zscore": Array (6)
}
```

```
{
  "_id": ObjectId("684fd938a6fe8c9666aac687"),
  "timestamp": 2025-06-16T08:42:10.100+00:00,
  "symbol": "BTCUSD",
  "zscore": Array (6)
}
```

Có thể xem rõ hơn với **connect string** sau:

`mongodb+srv://nguyenlenguyen1712004:nguyen1712004@cluster0.mp6lpiv.mongodb.net`

4 Bonus:

4.1 Overview: Mục tiêu và kiến trúc

Trong phần này, sẽ xây dựng một ứng dụng phân tích thời gian thực bằng PySpark Structured Streaming. Mục tiêu là xử lý dòng dữ liệu giá BTC từ Kafka, và với mỗi điểm dữ liệu, tính toán xem phải mất bao lâu để giá lần đầu tiên tăng lên hoặc lần đầu tiên giảm xuống trong một khoảng thời gian nhất định (20 giây).

Kết quả phân tích này, là các khoảng thời gian chờ (window), sẽ được gửi trở lại hai topic Kafka riêng biệt để các hệ thống khác có thể sử dụng.

Các thành phần chính:

- **Nền tảng xử lý:** PySpark Structured Streaming được sử dụng để xử lý dữ liệu theo thời gian thực.

- **Nguồn và đích dữ liệu:** Apache Kafka vừa là nguồn cung cấp dữ liệu giá (`btc-price`), vừa là nơi nhận kết quả phân tích (`btc-price-higher` và `btc-price-lower`).
- **Phương pháp:**
 - **Self-Join:** Kỹ thuật join một luồng dữ liệu với chính nó để so sánh mỗi sự kiện với các sự kiện xảy ra sau đó trong một cửa sổ thời gian.
 - **Stateful Aggregation:** Thực hiện các phép tính tổng hợp có trạng thái để tìm ra thời điểm sớm nhất mà giá thay đổi theo một hướng nhất định.
 - **Watermarking:** Xử lý dữ liệu trễ và quản lý bộ nhớ trạng thái một cách hiệu quả.

4.2 Detailed Explanation: Giải thích chi tiết phương pháp

- **Khởi tạo Spark Session:**

- Một `SparkSession` được tạo ra, cấu hình sẵn gói thư viện cần thiết (`spark-sql-kafka`) để có thể đọc và ghi dữ liệu với Kafka.
- Các cấu hình như `spark.sql.shuffle.partitions` giúp tối ưu hóa hiệu suất xử lý.

```
def create_spark_session():  
    return (  
        SparkSession.builder  
            .appName("BTC_Shortest_Windows_Analysis_Fixed")  
            .master(os.getenv('SPARK_MASTER', 'local[*]'))  
            .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.  
            # ... other configs  
            .getOrCreate()  
    )
```

- **Đọc và chuẩn bị dữ liệu từ Kafka:**

- Spark đọc dữ liệu từ topic `btc-price` dưới dạng một luồng streaming.
- Dữ liệu JSON thô được chuyển đổi thành một `DataFrame` có cấu trúc với các cột `symbol`, `price`, và `event_time`.
- **Watermark** 10 giây được áp dụng trên cột `event_time`. Điều này cho Spark biết rằng nó không cần phải chờ đợi những dữ liệu trễ hơn 10 giây, giúp giải phóng trạng thái (state) trong bộ nhớ và tránh việc state phát triển vô hạn.

- **Phân tích bằng kỹ thuật Self-Join:**

- Join luồng dữ liệu với chính nó. Một luồng được đặt bí danh là `left`, luồng còn lại là `right`.
- Điều kiện join được thiết kế để với mỗi sự kiện ở luồng `left`, nó sẽ được ghép cặp với tất cả các sự kiện ở luồng `right` mà có `event_time` xảy ra **sau** `left.event_time` nhưng không quá **20 giây**.
- Kỹ thuật này tạo ra một "cửa sổ nhìn về tương lai" 20 giây cho mỗi điểm dữ liệu, cho phép so sánh giá hiện tại với các mức giá sắp tới.

```
joined_stream = left_stream.join(  
    right_stream,  
    expr("""  
        left.join_key = right.join_key AND  
        right.event_time > left.event_time AND  
        right.event_time <= left.event_time + interval 20 seconds  
        """),  
    "leftOuter"  
)
```

- **Tính toán khoảng thời gian ngắn nhất:**

- Sau khi join, dữ liệu được nhóm theo sự kiện ban đầu (`left.event_time`, `left.price`).
- Với mỗi nhóm, tìm ra 2 giá trị:
 - * `first_higher_ts`: Timestamp **sớm nhất** trong cửa sổ 20 giây có giá cao hơn giá ban đầu.
 - * `first_lower_ts`: Timestamp **sớm nhất** trong cửa sổ 20 giây có giá thấp hơn giá ban đầu.
- Nếu không tìm thấy giá cao hơn/thấp hơn trong 20 giây, timestamp tương ứng sẽ là NULL. Sau đó, khoảng thời gian chờ (window) được tính bằng cách lấy timestamp tìm được trừ đi timestamp ban đầu. Nếu không tìm được (NULL), khoảng thời gian sẽ mặc định là **20.0** giây.

- **Gửi kết quả trở lại Kafka:**

- Kết quả cuối cùng bao gồm timestamp gốc và hai khoảng thời gian chờ (`higher_window`, `lower_window`).
- Dữ liệu được tách thành hai luồng riêng biệt, mỗi luồng chứa một loại "window" và được định dạng lại thành chuỗi JSON.
- Hai luồng này được ghi vào hai topic Kafka khác nhau: `btc-price-higher` và `btc-price-lower`.
- Việc sử dụng `checkpointLocation` là bắt buộc trong streaming có trạng thái, giúp Spark lưu lại tiến trình và có thể khôi phục lại từ đúng vị trí đó nếu có lỗi xảy ra.

4.3 Hướng dẫn chạy chương trình

- **Yêu cầu môi trường:**

- Một môi trường Spark đã được cài đặt và cấu hình (có thể là local hoặc trên một cluster).
- Kafka đang chạy.
- Python và các thư viện cần thiết.

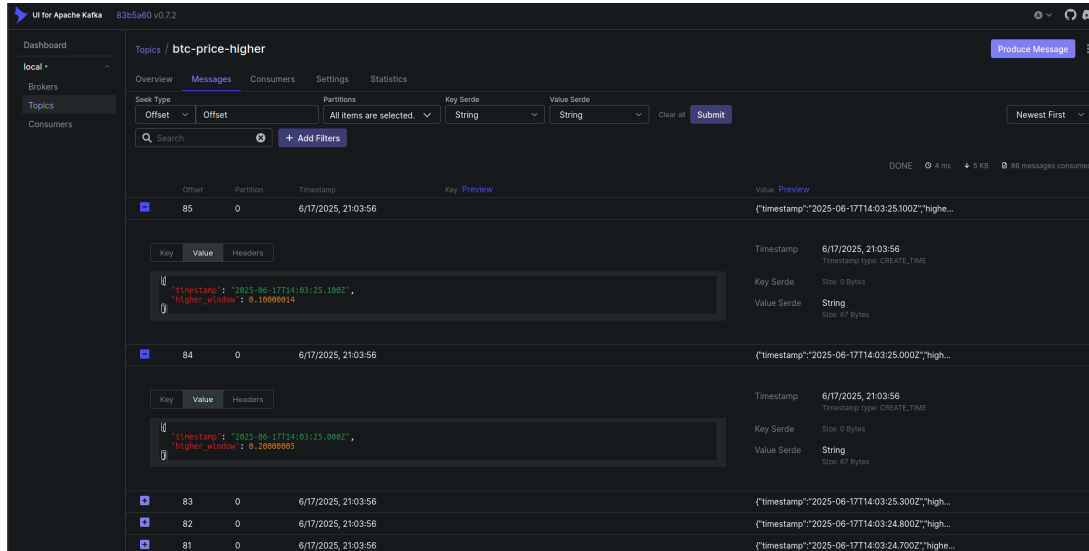
- **Chạy ứng dụng:**

- Vì đây là một ứng dụng Spark, nó nên được chạy bằng lệnh `spark-submit`. Lệnh này đảm bảo rằng tất cả các phụ thuộc và cấu hình của Spark được áp dụng đúng cách.

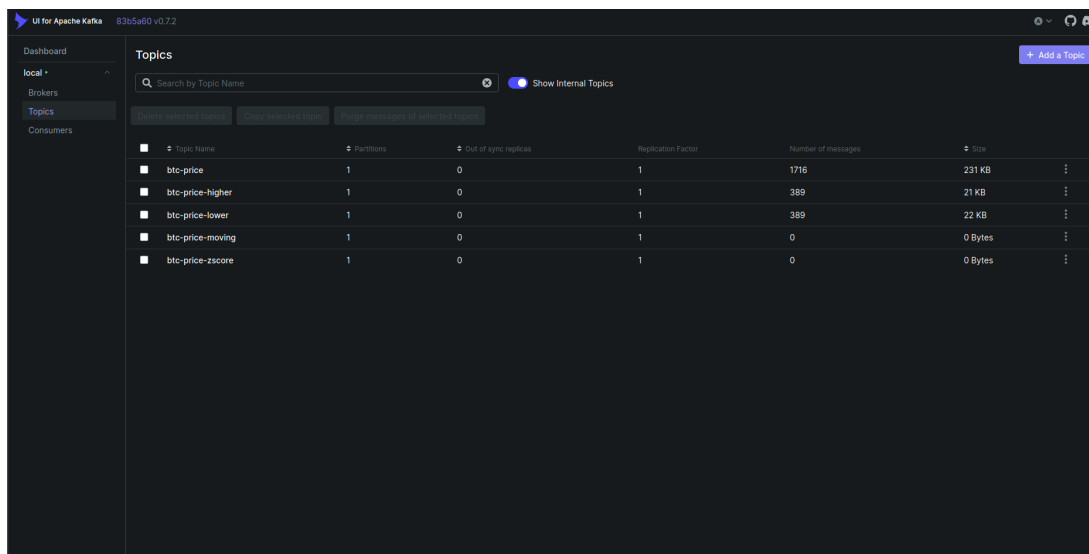
```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0 bonus.py
```

- Ứng dụng sẽ chạy liên tục ở chế độ nền, lắng nghe dữ liệu từ Kafka và gửi kết quả phân tích trở lại.

- **Kiểm tra kết quả:** Đây là hình ảnh trên giao diện:



The screenshot shows the Apache Kafka UI for the 'btc-price-higher' topic. The 'Messages' tab is selected, showing a list of messages. The table has columns for Offset, Partition, Timestamp, Key, and Value. The Value column shows a JSON object: {"timestamp": "2025-06-17T14:03:25.100Z", "higher_window": 0.1000014}. The interface also includes a search bar, filters, and a 'Produce Message' button.



The screenshot shows the Apache Kafka UI for the 'Topics' tab. It displays a table of topics with columns for Topic Name, Partitions, Out of sync replicas, Replication Factor, Number of messages, and Size. The topics listed are btc-price, btc-price-higher, btc-price-lower, btc-price-moving, and btc-price-zscore. The interface also includes a search bar and a 'Show Internal Topics' toggle.