PES UNIVERSITY
100 feet Ring Road, BSK 3rd Stage
Bengaluru 560085

Department of Computer Science and Engineering
B. Tech. CSE - 6th Semester
Jan – May 2023

UE20CS343
DATABASE TECHNOLOGIES (DBT)

Project Report

Analysis of real time price of Bitcoin using Kafka and Spark

TEAM
PES1UG20ME009: Alleti Rohan Reddy
PES1UG20CS043: Ananya Menon

Class of Prof. Suresh Jamadagni

Table of Contents

1. Introduction

Bitcoin is a digital cryptocurrency that has gained a lot of attention in the last few years. The price of bitcoin fluctuates constantly, and it is essential for traders and investors to keep track of the real-time price of bitcoin. In this report, we analyze the real-time price of bitcoin using Spark and Kafka.

Spark is a fast and flexible big data processing engine that allows for real-time analysis of large data sets. Kafka is a distributed streaming platform that is used for building real-time data pipelines and streaming applications. Together, they provide a powerful platform for analyzing real-time data.

The objective of this report is to analyze the real-time price of bitcoin using Spark and Kafka. We will build a real-time data pipeline that collects bitcoin price data, processes it in real-time using Spark, and produces insights that will help traders and investors make informed decisions.

2. Installation of Software

Services such as spark, kafka, zookeeper and postgres database are being run in docker containers connected over a single bridge docker network. Code is written in python along with few packages installed using pip.

source code:
https://github.com/RohanJnr/bitcoin-analysis-spark-kafka

The following are the required softwares to run the project.

- Docker
- Docker compose
- Python 3.11
- Python packages
  - websocket-client==1.5.1
  - pyspark==3.3.2
  - kafka-python==2.0.2
  - psycopg2-binary==2.9.6
- Softwares Images running in docker containers
  - confluentinc/cp-zookeeper:7.3.2
    - https://hub.docker.com/r/confluentinc/cp-zookeeper
  - confluentinc/cp-kafka:7.3.2
    - https://hub.docker.com/r/confluentinc/cp-kafka
  - bitnami/spark:3.3.
    - https://hub.docker.com/r/bitnami/spark
  - postgres:15-alpine
    - https://hub.docker.com/_/postgres
    - Database is created with 2 tables:
      - “stock_data”: This table stores real time bitcoin price and volume of purchase.
      - “stock_aggregates”: This table contains average data of price and sum of volume of purchase of bitcoin

between a certain time interval which is set by spark sliding window.

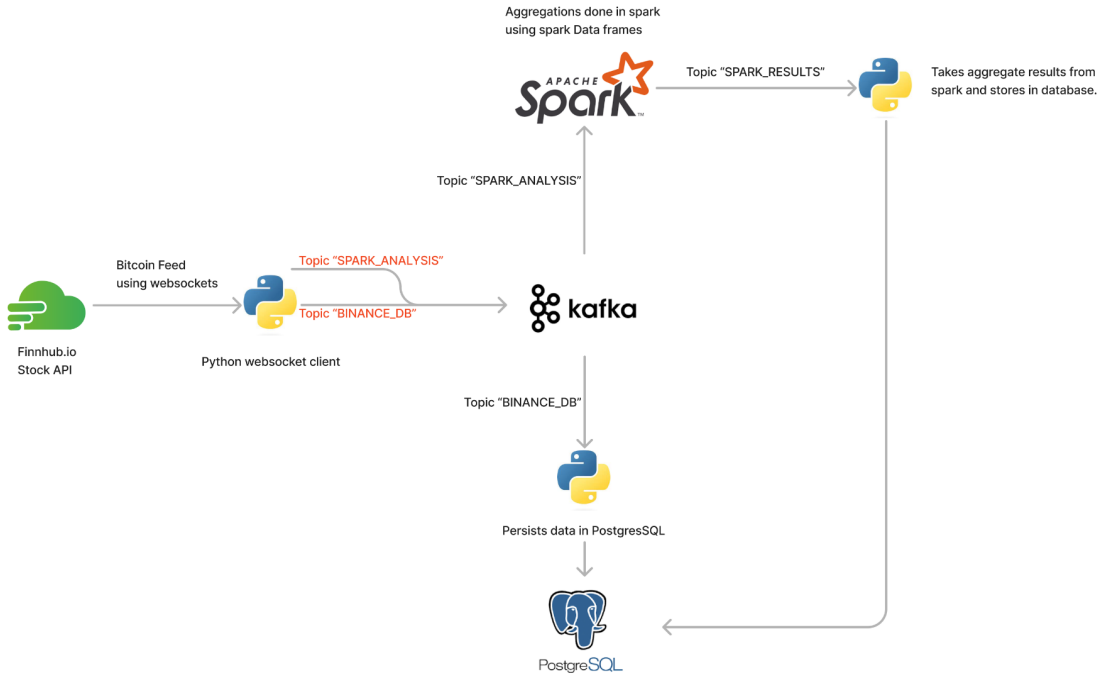
- Command to run all services:
  `docker compose up`

3. Problem Description

- Bitcoin price live / real time data is being fetched from https://finnhub.io. They provide a websocket URL from which the feed is received (data comes in every 500ms-1000ms).
- The feed is then published to 2 kafka topics
  - Topic “BINANCE_DB”
    - The consumer of this topic stores the real time data in a database under the table “stock_data”.
  - Topic “SPARK_ANALYSIS”
    - Spark is the consumer for this topic. Spark analysis this data, does aggregations and then sends the aggregated to the kafka topic “SPARK_RESULTS”
- The topic “SPARK_RESULTS” is received by another consumer which stores the results in the database.
- For Batch processing, the spark instances reads the database “stock_data” table and computes the following:
  - Overall average price
  - Average price per minute (since the data coming in high velocity: every 500ms-1000ms)
  - Total purchases
  - Total purchases per minute

4.1 Stream processing Flow Chart

Link to figma diagram:
https://www.figma.com/file/7vopZh1Q0YByug4QRINsr8/Spark-Streaming?node-id=0%3A1&t=uFjFYhnZ5HgkwZYx-1



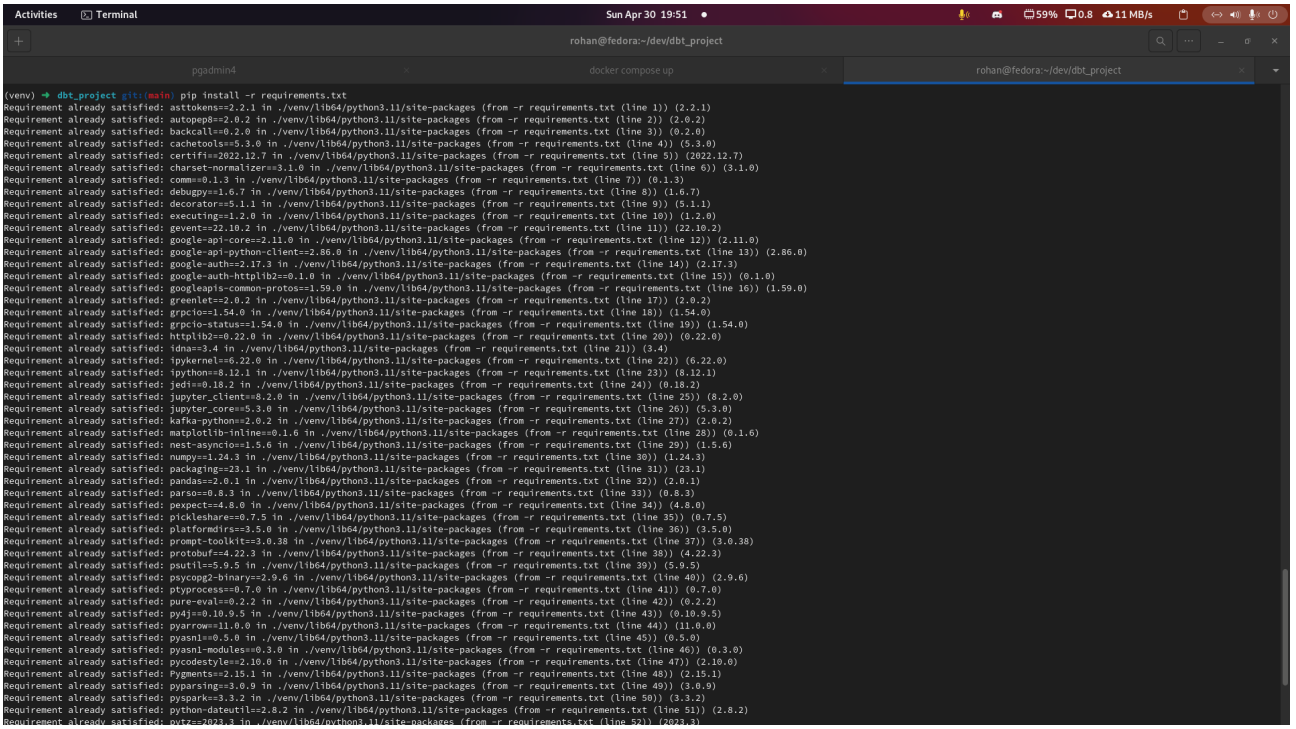
4.2 Batch Processing Flow Chart

5.Input Data

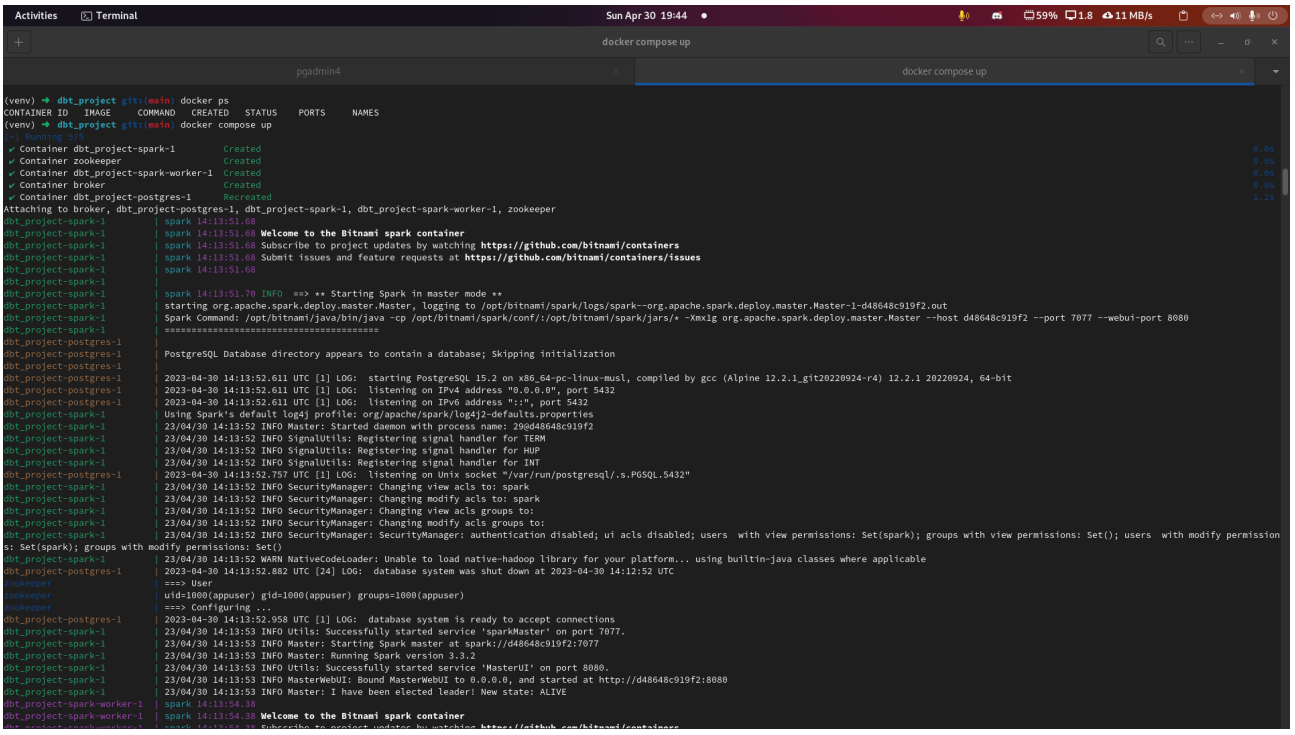
- Source and description
  - Finnhub.io site was used for live input data.
  - https://finnhub.io/docs/api/websocket-trades
  - The input data consists of
    - **Bitcoin price**
    - **timestamp**
    - **volume purchased**
    - symbol
    - conditions

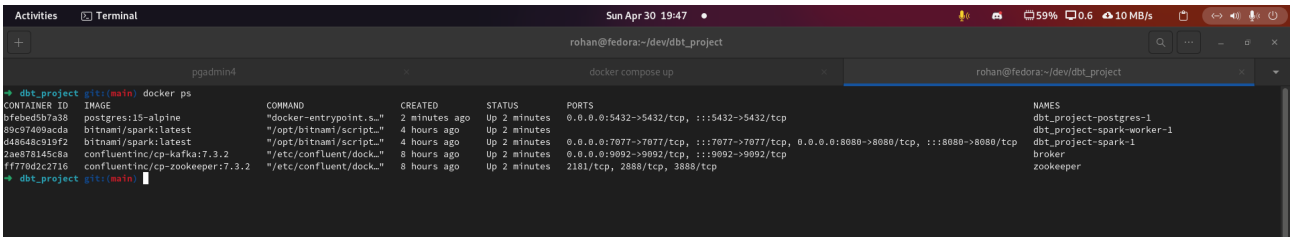
6. Streaming mode experiment

- Install all dependencies by creating a python and virtualenv and install packages using requirements.txt

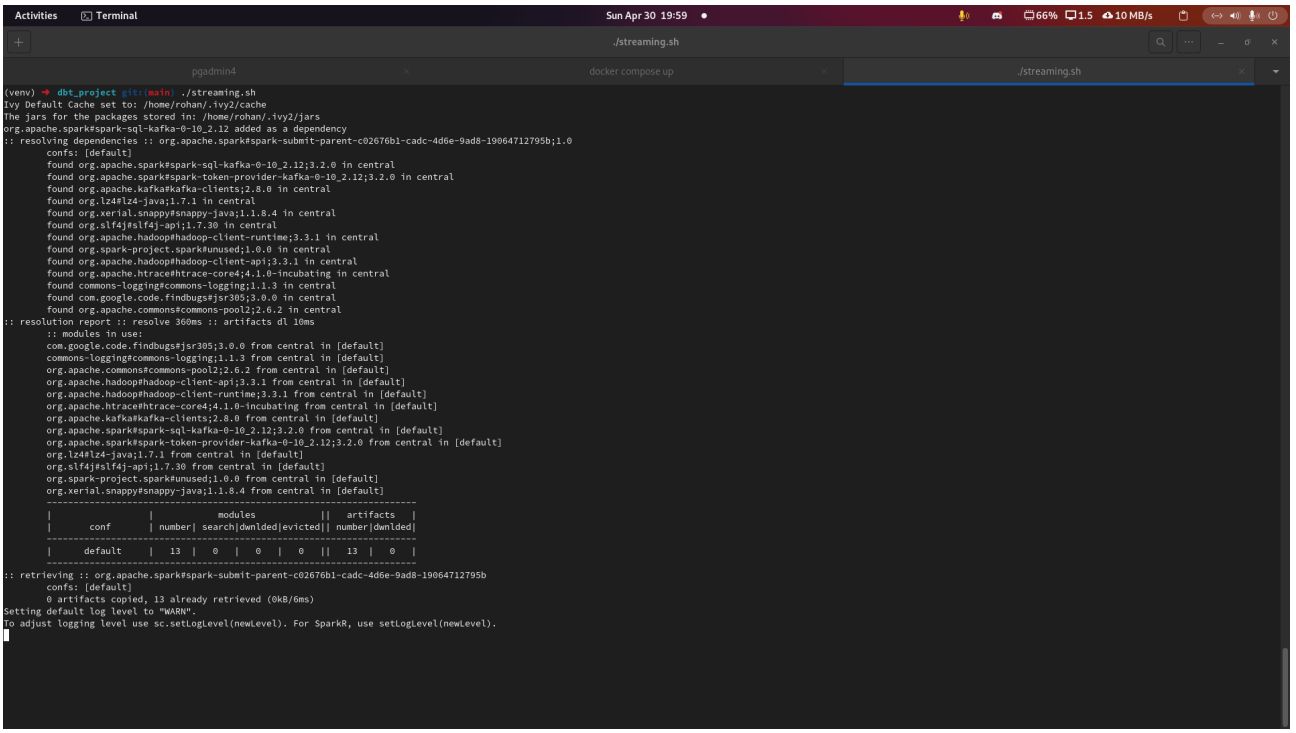


- Run all docker containers: Includes zookeeper, kafka, spark and postgres via docker compose

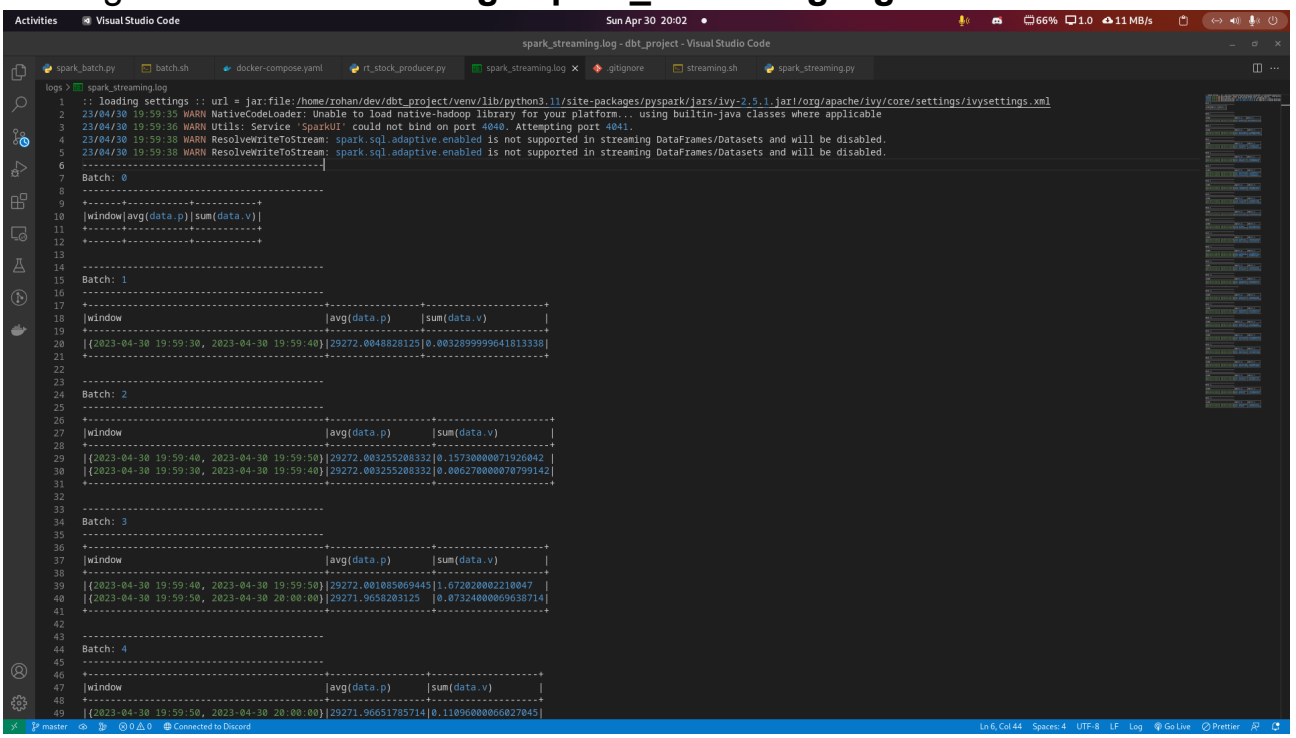




- We need to now run the following files:
  - **kafka/rt_stock_producer.py** (receives real time data and publishes to kafka topics)
  - **kafka/persists_db_consumer.py** (Persists data from topic “BINANCE_DB” to database)
  - **kafka/results_consumer.py** (Kafka consumer to receive results after calculations/aggregations from spark)
  - **spark/spark_streaming.py** (Listens to bitcoin data from kafka and performs aggregations)
- The above four files can be run by running “streaming.sh” file. The bash script is already configured to run the 4 files and logs the output to the “log/” directory.

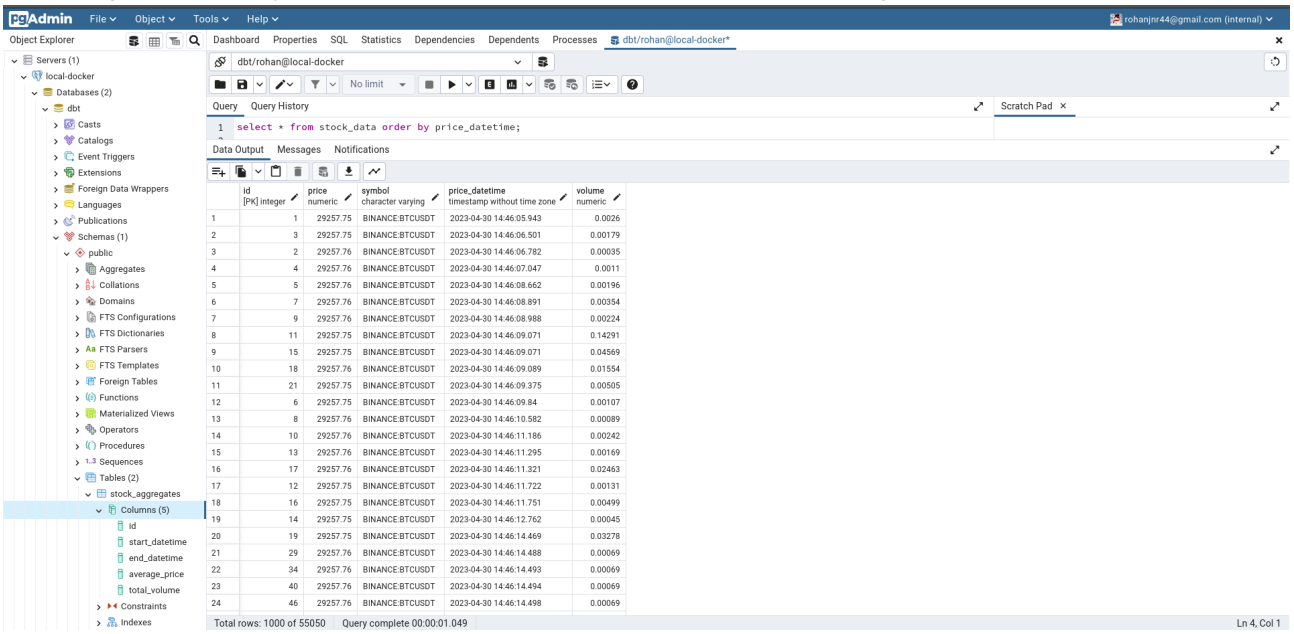


- The logs of interest are in “**logs/spark_streaming.log**”.

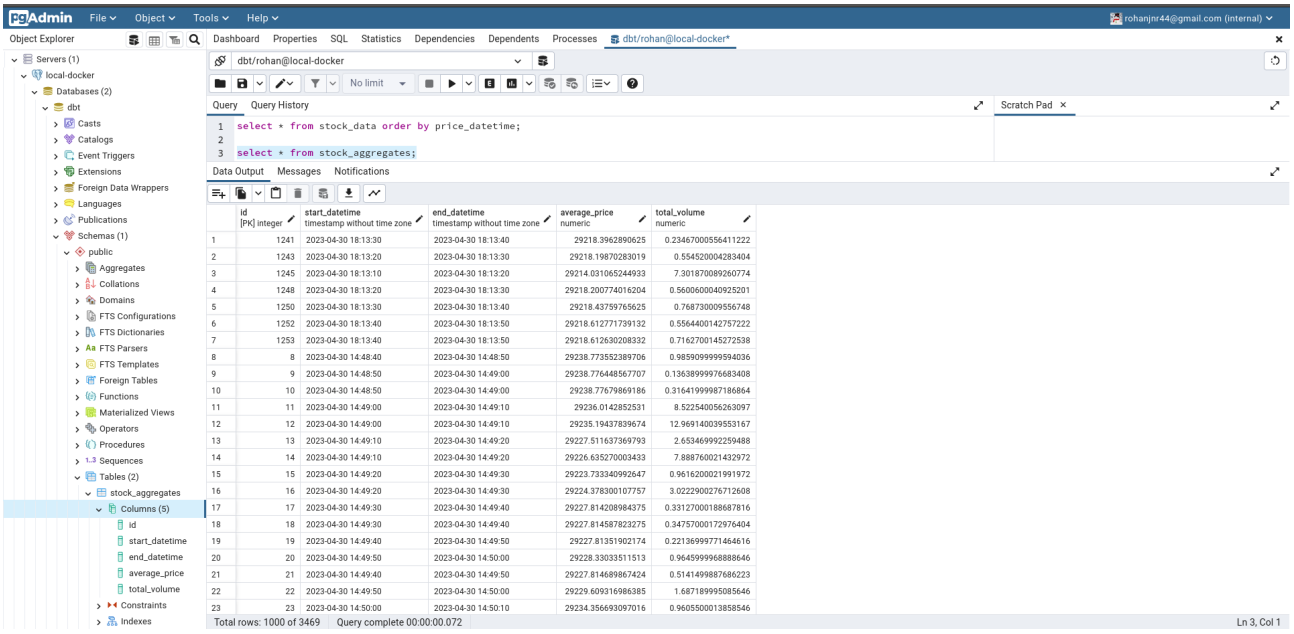


- From the screenshot, Spark is receiving data from kafka and processing them in batches with a sliding window set to a duration of 10 seconds.

- The results will be available in the database as well. We can see this by running the query: **select * from stock_data order by price_datetime;**



- Query to see results after sliding window aggregations: **select * from stock_aggregates;**



- Database is being populated as spark is outputting data after aggregations.

Relevant Code

- Code for sql tables

```
CREATE TABLE IF NOT EXISTS stock_data (
    id SERIAL PRIMARY KEY,
    price DECIMAL,
    symbol VARCHAR(128),
    price_datetime TIMESTAMP WITHOUT TIME ZONE,
    volume DECIMAL
);

CREATE TABLE IF NOT EXISTS stock_aggregates (
    id SERIAL PRIMARY KEY,
    start_datetime TIMESTAMP WITHOUT TIME ZONE,
    end_datetime TIMESTAMP WITHOUT TIME ZONE,
    average_price DECIMAL,
    total_volume DECIMAL
)
```

- Code for spark aggregations

```
# Read data from Kafka as a DataFrame
df =
spark.readStream.format("kafka").options(**kafka_params).load()

kafka_df = df.selectExpr("CAST(value AS STRING)", "CAST(timestamp
AS TIMESTAMP)").withWatermark(
    "timestamp", "20 seconds")
```

```
parsed_df = kafka_df.select(from_json("value",
schema).alias("data"))

windowed_stream = parsed_df \
    .groupBy(window(col("data.datetime"), "10 seconds", "10
seconds")) \
    .agg(
        avg(col("data.p")),
        _sum(col("data.v"))
    )

# Write to console
console_write = ( windowed_stream.writeStream \
    .outputMode("update") \
    .format("console") \
    .option("truncate", False) \
    .trigger(processingTime="10 seconds") \
    .start()
)

# Write to kafka topic
kafka_write = windowed_stream \
    .selectExpr("to_json(struct(*)) AS value") \
    .writeStream \
    .outputMode("update") \
    .trigger(processingTime="10 seconds") \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "SPARK_RESULTS") \
    .start()
```

```
kafka_write.awaitTermination()
console_write.awaitTermination()
```

7. Batch mode experiment

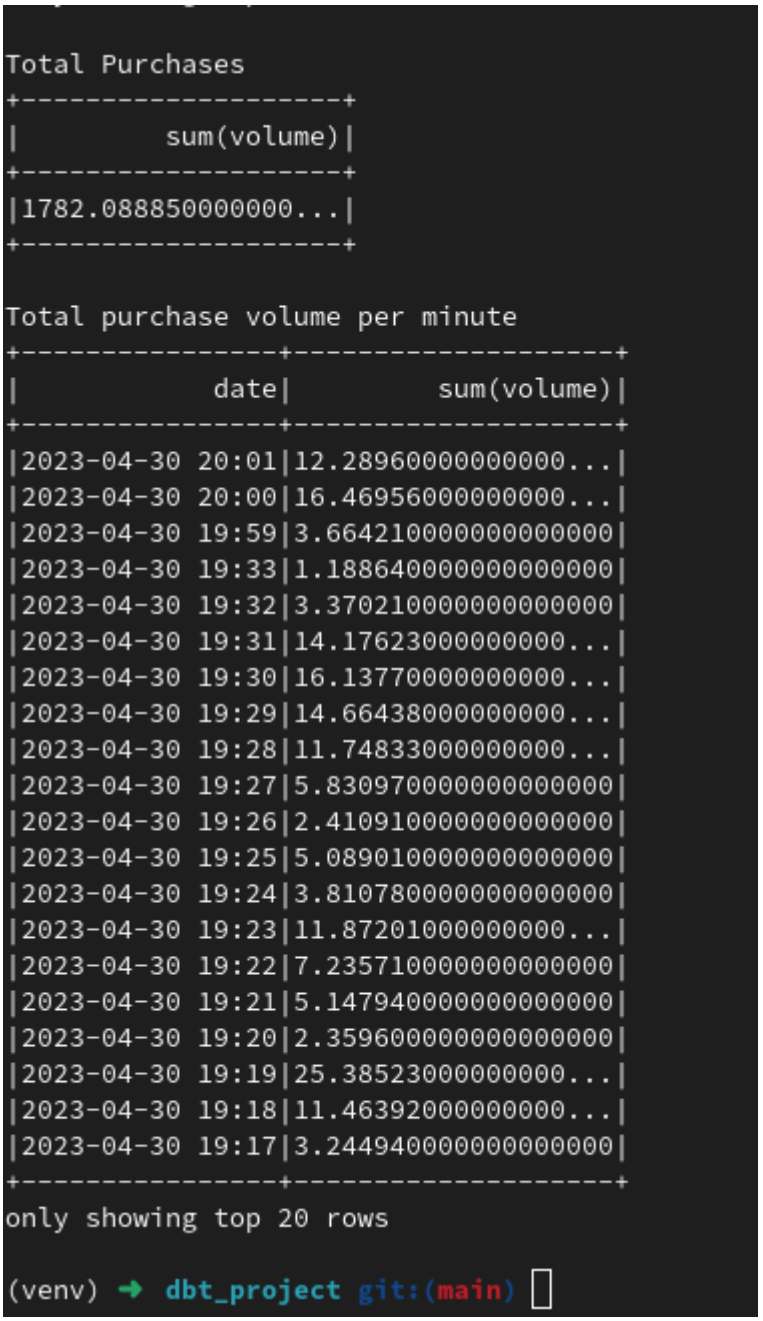
- Install all dependencies by creating a python and virtualenv and install packages using requirements.txt as shown previously
- Start all docker containers as shown previously
- We will now run **“spark/spark_batch.py”** which will put data from the database and perform analysis on the data in spark.

```
(venv) ➜ dbt_project git:(main) python spark/spark_batch.py
23/04/30 20:20:01 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/04/30 20:20:01 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Time intervals for the following calculations.
Start Timestamp
+--------------------+
| min(price_datetime)|
+--------------------+
|2023-04-30 14:46:...|
+--------------------+

End Timestamp
+--------------------+
| max(price_datetime)|
+--------------------+
|2023-04-30 20:01:...|
+--------------------+

Overall average price
+--------------------+
|          avg(price)|
+--------------------+
|29239.60886630336...|
+--------------------+

Calculating average price per minute
+----------------+--------------------+
|            date|          avg(price)|
+----------------+--------------------+
|2023-04-30 18:39|29263.97522598870...|
|2023-04-30 18:35|29242.12655303030...|
|2023-04-30 18:00|29197.10410326086...|
|2023-04-30 18:09|29205.90596551724...|
|2023-04-30 17:12|29186.83859296482...|
|2023-04-30 14:46|29257.75787500000...|
|2023-04-30 17:31|29197.68438709677...|
|2023-04-30 18:36|29251.87102564102...|
|2023-04-30 19:09|29272.21846153846...|
|2023-04-30 17:20|29194.13819767441...|
|2023-04-30 19:29|29291.88873239436...|
|2023-04-30 18:37|29265.36465181058...|
|2023-04-30 17:46|29208.35416083916...|
|2023-04-30 18:07|29190.48142857142...|
|2023-04-30 18:11|29205.31230158730...|
|2023-04-30 19:19|29291.17620183486...|
|2023-04-30 18:46|29266.35148401826...|
|2023-04-30 17:18|29186.52013752455...|
|2023-04-30 18:42|29263.08375000000...|
|2023-04-30 17:27|29199.53582568807...|
+----------------+--------------------+
only showing top 20 rows
```



```
Total Purchases
+------------------+
|       sum(volume)|
+------------------+
|1782.088850000000...|
+------------------+

Total purchase volume per minute
+----------------+------------------+
|            date|       sum(volume)|
+----------------+------------------+
|2023-04-30 20:01|12.28960000000000...|
|2023-04-30 20:00|16.46956000000000...|
|2023-04-30 19:59|3.664210000000000000|
|2023-04-30 19:33|1.188640000000000000|
|2023-04-30 19:32|3.370210000000000000|
|2023-04-30 19:31|14.17623000000000...|
|2023-04-30 19:30|16.13770000000000...|
|2023-04-30 19:29|14.66438000000000...|
|2023-04-30 19:28|11.74833000000000...|
|2023-04-30 19:27|5.830970000000000000|
|2023-04-30 19:26|2.410910000000000000|
|2023-04-30 19:25|5.089010000000000000|
|2023-04-30 19:24|3.810780000000000000|
|2023-04-30 19:23|11.87201000000000...|
|2023-04-30 19:22|7.235710000000000000|
|2023-04-30 19:21|5.147940000000000000|
|2023-04-30 19:20|2.359600000000000000|
|2023-04-30 19:19|25.38523000000000...|
|2023-04-30 19:18|11.46392000000000...|
|2023-04-30 19:17|3.244940000000000000|
+----------------+------------------+
only showing top 20 rows

(venv) ➜ dbt_project git:(main)
```

- The above gives insights on average prices, number of purchases and prices and purchases of bitcoin per minute.

Relevant Code

```
# Start timestamp
print("Start Timestamp")
start_timestamp = df.agg({"price_datetime": "min"})
start_timestamp.show()

# End timestamp
print("End Timestamp")
end_timestamp = df.agg({"price_datetime": "max"})
end_timestamp.show()

# Overall average price
print("Overall average price")
average = df.agg({"price": "avg"})
average.show()

# Calculating average price per minute
print("Calculating average price per minute")
df = df.withColumn("date", date_format(col("price_datetime"), "yyyy-MM-dd
HH:mm"))
average_per_min = df.groupBy("date").mean("price")
average_per_min.show()


# Total purchase volume
print("Total Purchases")
volume_sum = df.agg({"volume": "sum"})
volume_sum.show()

# Total purchase volume per minute
print("Total purchase volume per minute")
sum_volume_per_min = df.groupBy("date").sum("volume").orderBy(desc("date"))
sum_volume_per_min.show()
```

8. Comparison of streaming and batch mode

In Spark, batch mode and streaming mode are two different processing models that are used to process data.

Batch processing refers to processing a fixed amount of data at a time. In Spark, batch processing involves reading a set of data, processing it, and then outputting the results. Batch processing is typically used for non-real-time applications where processing time is not critical. In this application, batch mode was done on already collected data from the bitcoin api.

Streaming processing, on the other hand, refers to processing data in real-time as it is generated. In Spark, streaming processing involves reading data in small batches, processing it, and then outputting the results. Streaming processing is typically used for real-time applications where processing time is critical. In this application, streaming processing was done on real time, high velocity data of bitcoin information.

9. Conclusion

The real time analysis of bitcoin prices and purchase volume done using Python, Spark, Kafka and postgres showed interesting insights on the trend of bitcoin prices and purchase volume. Value of bitcoin is extremely unpredictable and hence this analysis helps us understand the market and how the price fluctuates. The purchase volume is quite random and the price fluctuates heavily due to this high purchase by 100s of dollars.

Streaming analysis was helpful in understanding the sub-minute fluctuation in bitcoin and this can be very useful for day traders to instantly monitor and sell/buy bitcoin.

Batch mode processing has been demonstrated on a per minute basis which can be easily extended to hourly or daily batch processing whose insights can be useful for long term buyers of bitcoin.

10. References

- https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html
- https://developer.confluent.io/quickstart/kafka-docker/
- https://www.psycopg.org/docs/
- https://spark.apache.org/docs/latest/api/python/
- https://kafka-python.readthedocs.io/en/master/