Database Technologies (UE20CS343)
Project Report

Topic: Bitcoin Data Processing and Visualization

Team:

1. Chaitanya Madhav R PES1UG20CS634
2. Vallabha Kowshik PES1UG21CS839

Document Presented with Screenshots:

Name:	Chaitanya Madhav R
SRN:	PES1UG20CS634
Section:	K

Table of Contents

1. Introduction

This database technologies project leverages modern technologies such as web sockets, Kafka, and Spark to aggregate real-time price, best ask, and best bid values of Bitcoin. The objective of this project is to develop a robust and fast system capable of collecting and processing Bitcoin data with high precision.

To achieve this objective, we will utilize web sockets to establish a continuous connection with the Bitcoin data source, transmitting the incoming data to Kafka, a distributed streaming platform that will act as an intermediary between system components. Additionally, we will utilize Spark, a highly efficient data processing engine, to analyse and aggregate the Bitcoin data in real-time, providing our users with up-to-date and accurate information.

2. Installation of Software

A. Streaming software

To install Apache Zookeeper and Apache Kafka [3.4.0] we downloaded the latest release from this URL and extracted the TAR file in the manner described by the guide released by apache.org.

Apache Spark 3.4.0 can be installed via pip3 with PySpark or installation of the software can be done manually following steps provided here.

Furthermore, 'kafka-python,' 'pyspark.sql,' 'pyspark,' and 'websocket' need to be installed via pip3 using Python 3 and required node modules need to be installed via NPM or YARN.

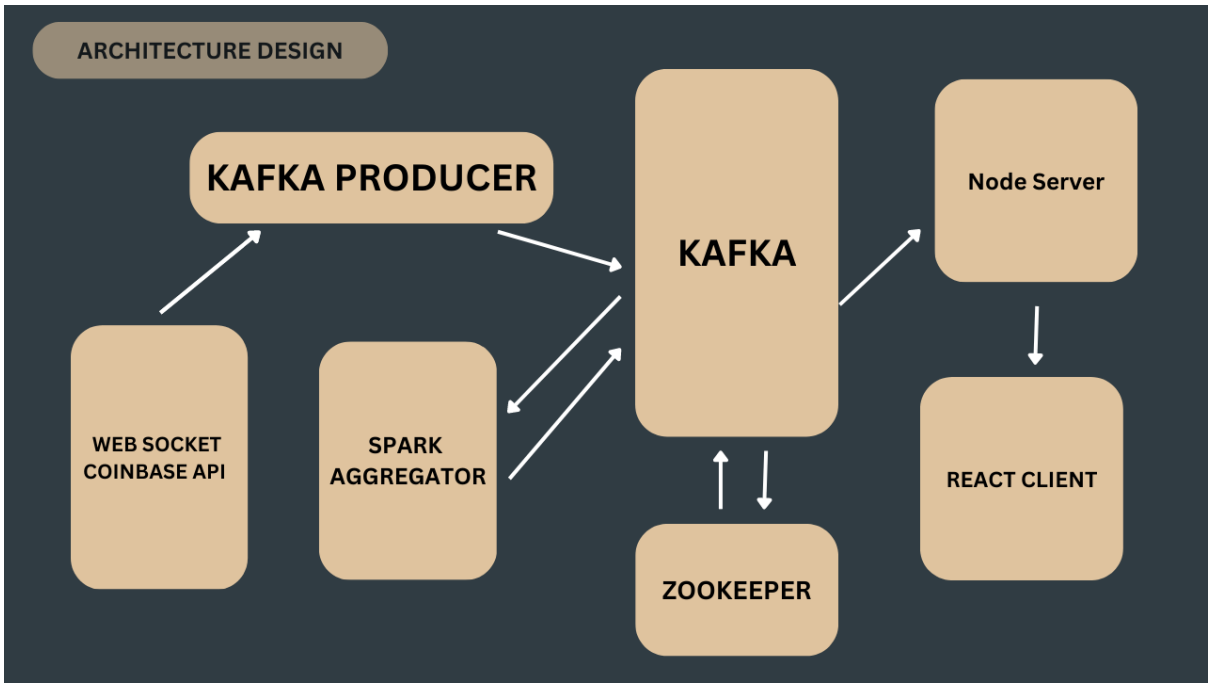
B. DBMS Software

The DBMS software used in this project is a relational database, namely MySQL (8+). It is required to create a schema within a database for the python scripts to execute successfully.

3. Problem Description

The lack of accessible and accurate systems for visualizing real-time Bitcoin data presents a significant challenge for businesses and individuals who require this information. Current solutions often lack accuracy and visualization capabilities, making it difficult for users to make informed decisions. To address this issue, we have developed a project that relies on an external API to stream and aggregate Bitcoin data over one-minute intervals. This data is stored in a database and streamed to a web interface, where it is displayed as a line graph.

4. Architecture Diagram



5. INPUT SOURCE

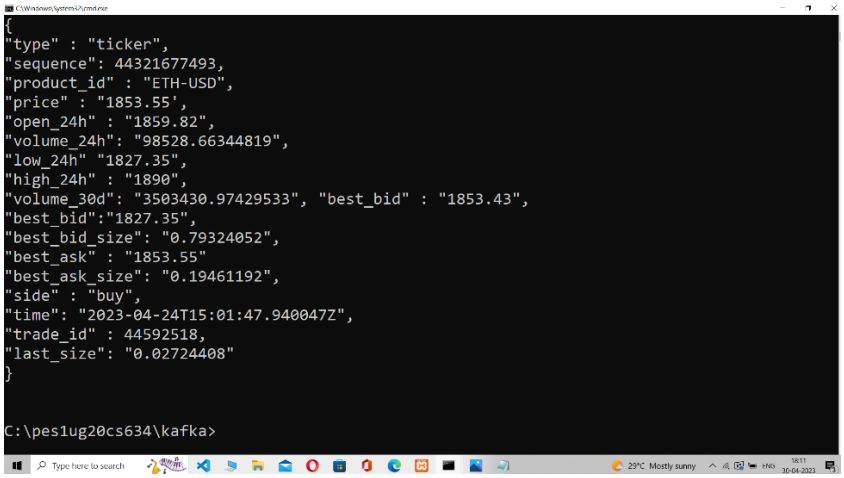
For the streaming data to be processed using Spark via PySpark, we are using the free CWE will be using the processed data from spark and send it to MySQL . Coinbase API to provide cryptocurrency data of Ethereum, Bitcoin, and Dogecoin assets. The API provides best-bid, best-ask and price values, of which we're taking an aggregation of the same and presenting it to a user of our system.

a. Source

We are using the websocket version of the API to avail continuous data which can be processed in our python scripts.
Data is received in JSON format, which is promptly sent as utf-8 strings into Kafka.

b. Description (**terminal contains SRN**)

Below is a sample of the JSON data received from Coinbase's API.



6. Streaming Mode Experiment

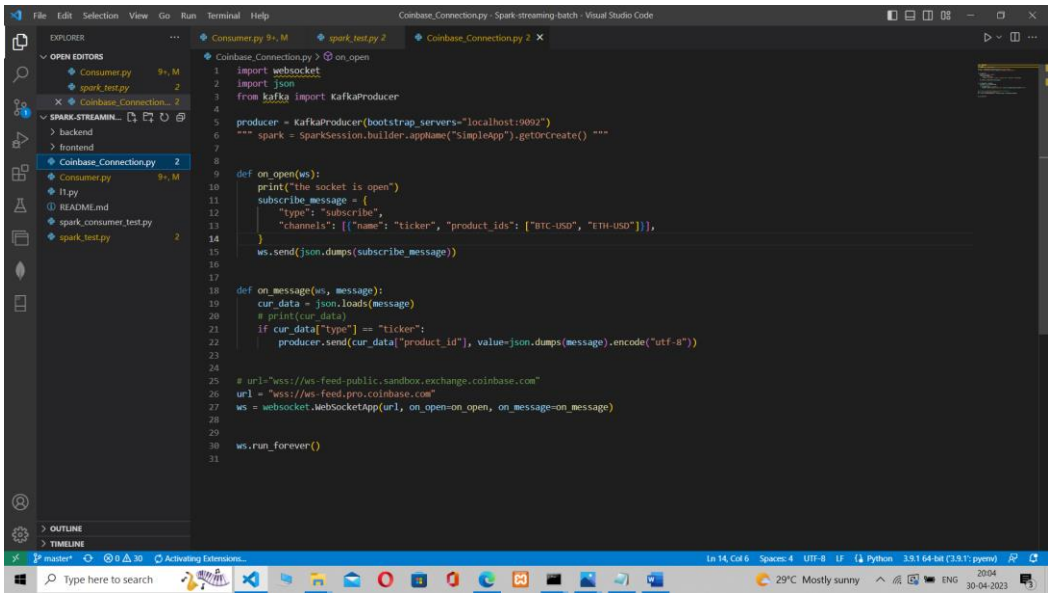
a. Windows

We are utilising a 1-minute tumbling window to get near real-time data aggregations and a 10-minute watermark value. We are pushing the 1-minute aggregations along with their corresponding timestamp values into a Kafka topic and into our MySQL database.

b. Workload

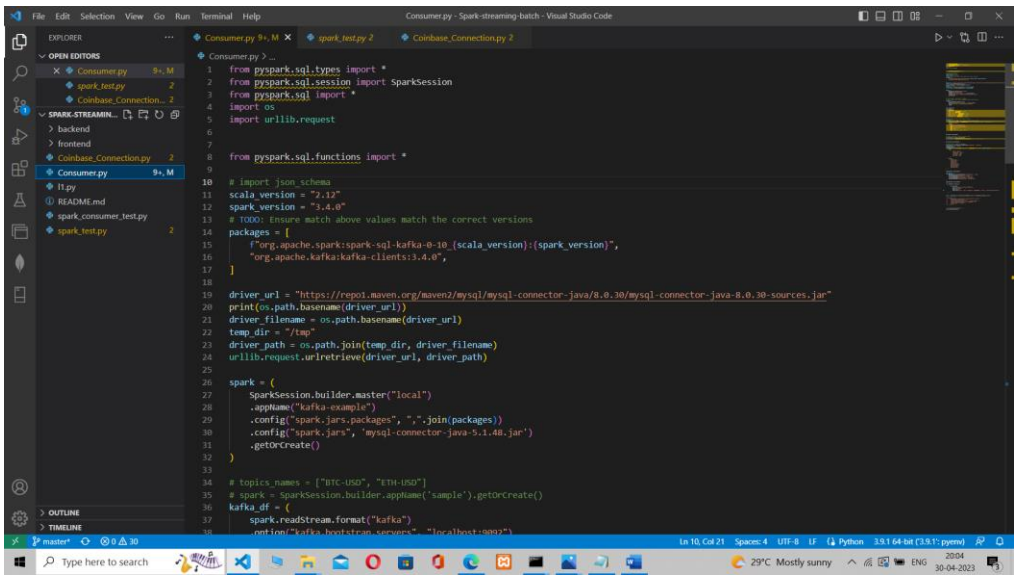
c. Scripts

Coinbase_Connection.py

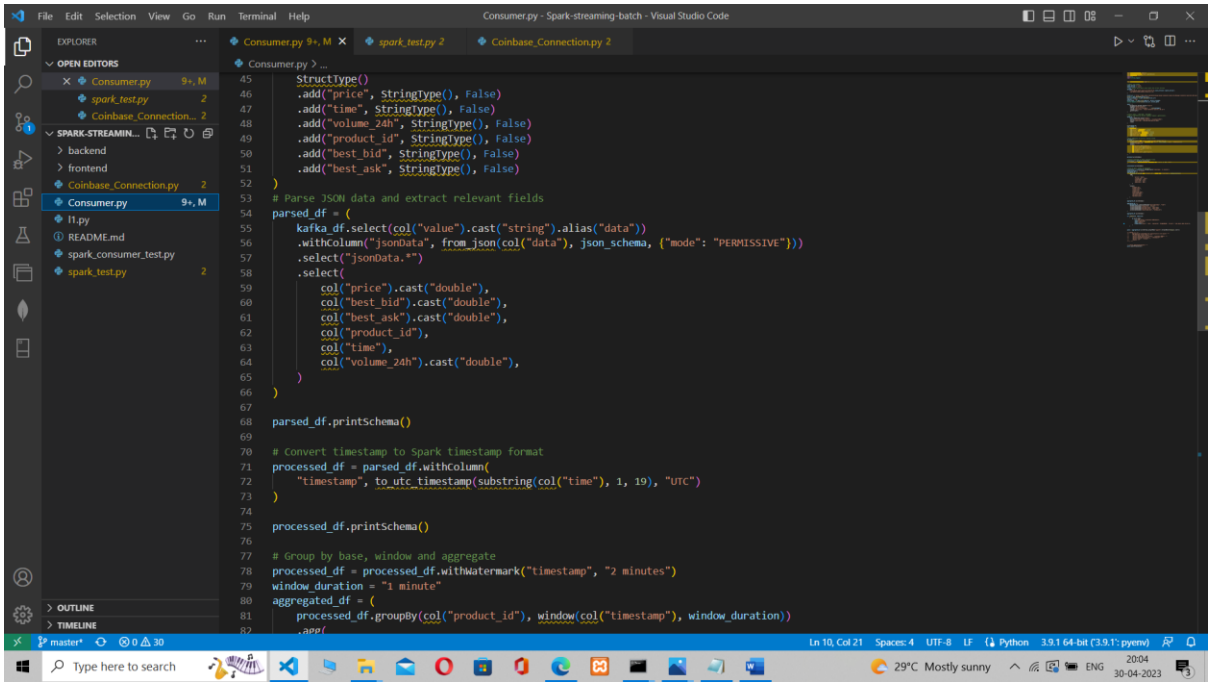


```
import websocket
import json
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers="localhost:9092")
""" spark = SparkSession.builder.appName("SimpleApp").getOrCreate() """


def on_open(ws):
    print("the socket is open")
    subscribe_message = {
        "type": "subscribe",
        "channels": [{"name": "ticker", "product_ids": ["BTC-USD", "ETH-USD"]}],
    }
    ws.send(json.dumps(subscribe_message))


def on_message(ws, message):
    cur_data = json.loads(message)
    # print(cur_data)
    if cur_data["type"] == "ticker":
        producer.send(cur_data["product_id"], value=json.dumps(message).encode("utf-8"))


# url="wss://ws-feed-public.sandbox.exchange.coinbase.com"
url = "wss://ws-feed.pro.coinbase.com"
ws = websocket.WebSocketApp(url, on_open=on_open, on_message=on_message)


ws.run_forever()
```

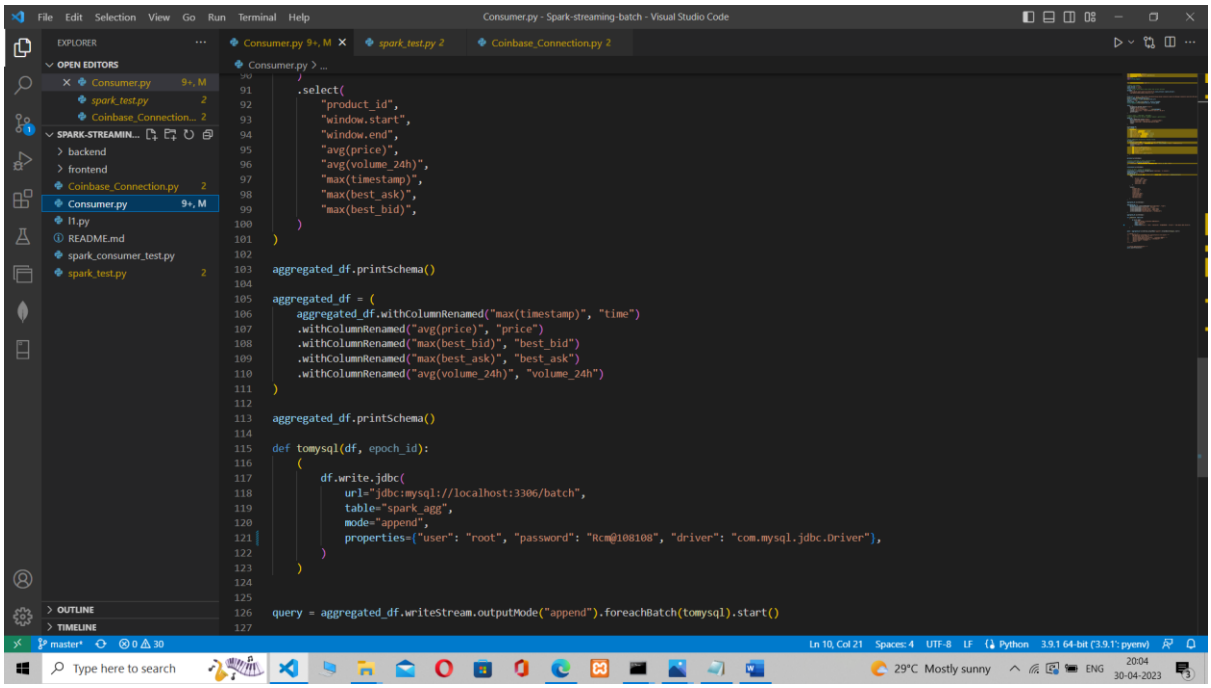
Consumer.py



```
from pyspark.sql.types import *
from pyspark.sql.session import SparkSession
from pyspark.sql import *
import os
import urllib.request


from pyspark.sql.functions import *

# import json_schema
scala_version = "2.12"
spark_version = "3.4.0"
# TODO: Ensure match above values match the correct versions
packages = [
    f"org.apache.spark:spark-sql-kafka-0-10_{scala_version}:{spark_version}",
    "org.apache.kafka:kafka-clients:3.4.0",
]

driver_url = "https://repo1.maven.org/maven2/mysql/mysql-connector-java/8.0.30/mysql-connector-java-8.0.30-sources.jar"
print(os.path.basename(driver_url))
driver_filename = os.path.basename(driver_url)
temp_dir = "/tmp"
driver_path = os.path.join(temp_dir, driver_filename)
urllib.request.urlretrieve(driver_url, driver_path)

spark = (
    SparkSession.builder.master("local")
    .appName("kafka-example")
    .config("spark.jars.packages", ",".join(packages))
    .config("spark.jars", 'mysql-connector-java-5.1.48.jar')
    .getOrCreate()
)

# topics_names = ["BTC-USD", "ETH-USD"]
# spark = SparkSession.builder.appName('sample').getOrCreate()
kafka_df = (
    spark.readStream.format("kafka")
```

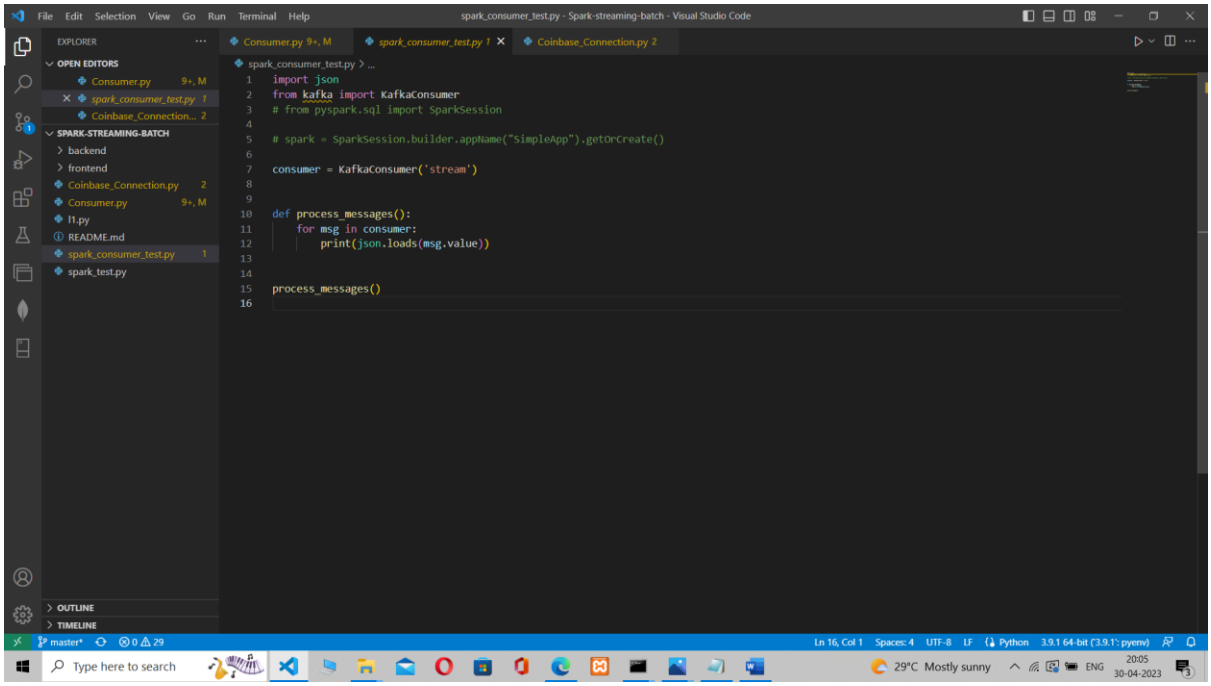


```
    StructType()
    .add("price", StringType(), False)
    .add("time", StringType(), False)
    .add("volume_24h", StringType(), False)
    .add("product_id", StringType(), False)
    .add("best_bid", StringType(), False)
    .add("best_ask", StringType(), False)
)
# Parse JSON data and extract relevant fields
parsed_df = (
    kafka_df.select(col("value").cast("string").alias("data"))
    .withColumn("jsonData", from_json(col("data"), json_schema, {"mode": "PERMISSIVE"}))
    .select("jsonData.*")
    .select(
        col("price").cast("double"),
        col("best_bid").cast("double"),
        col("best_ask").cast("double"),
        col("product_id"),
        col("time"),
        col("volume_24h").cast("double"),
    )
)

parsed_df.printSchema()

# Convert timestamp to Spark timestamp format
processed_df = parsed_df.withColumn(
    "timestamp", to_utc_timestamp(substring(col("time"), 1, 19), "UTC")
)

processed_df.printSchema()

# Group by base, window and aggregate
processed_df = processed_df.withWatermark("timestamp", "2 minutes")
window_duration = "1 minute"
aggregated_df = (
    processed_df.groupBy(col("product_id"), window(col("timestamp"), window_duration))
    .agg(
```

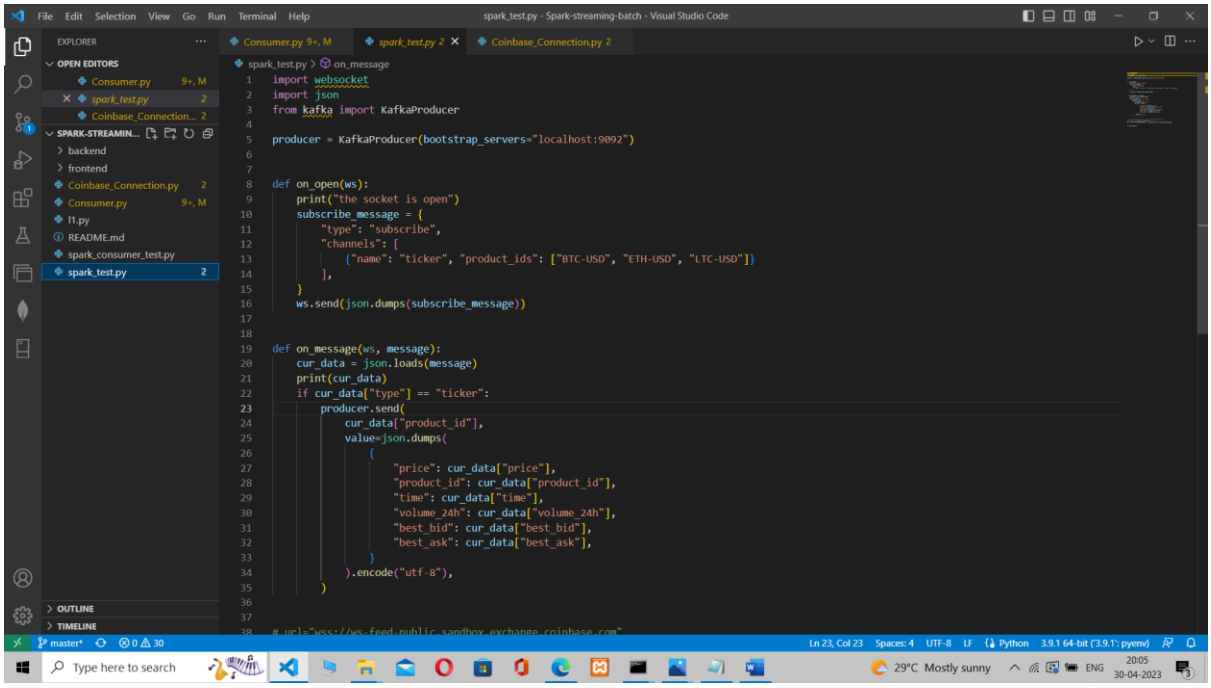


```
    .select(
        "product_id",
        "window.start",
        "window.end",
        "avg(price)",
        "avg(volume_24h)",
        "max(timestamp)",
        "max(best_ask)",
        "max(best_bid)",
    )
)

aggregated_df.printSchema()

aggregated_df = (
    aggregated_df.withColumnRenamed("max(timestamp)", "time")
    .withColumnRenamed("avg(price)", "price")
    .withColumnRenamed("max(best_bid)", "best_bid")
    .withColumnRenamed("max(best_ask)", "best_ask")
    .withColumnRenamed("avg(volume_24h)", "volume_24h")
)

aggregated_df.printSchema()

def tomysql(df, epoch_id):
    (
        df.write.jdbc(
            url="jdbc:mysql://localhost:3306/batch",
            table="spark_agg",
            mode="append",
            properties={"user": "root", "password": "Rcm@108108", "driver": "com.mysql.jdbc.Driver"},
        )
    )


query = aggregated_df.writeStream.outputMode("append").foreachBatch(tomysql).start()
```

Spark_consumer_test.py



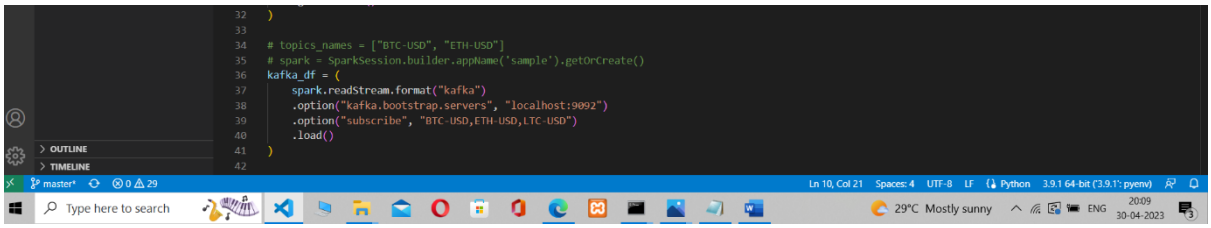
Spark_test.py



i) Getting the data from kafka and forming a dataframe so that we can analyze the data

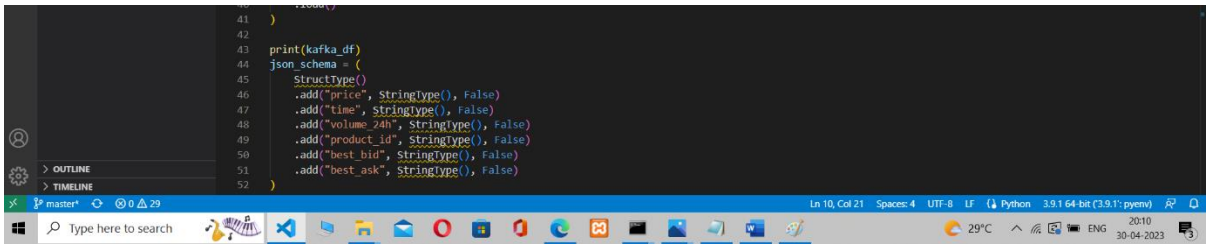
The three topics we are subscribing to are -

- BTC-USD
- ETH-USD
- LTC-USD



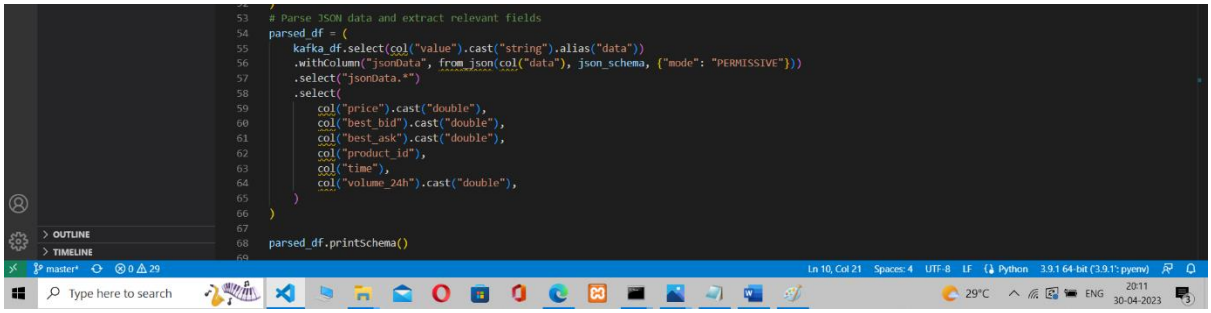
```
)

# topics_names = ["BTC-USD", "ETH-USD"]
# spark = SparkSession.builder.appName('sample').getOrCreate()
kafka_df = (
    spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("subscribe", "BTC-USD,ETH-USD,LTC-USD")
    .load()
)
```

ii) Defining the json schema for the dataframe which we are getting from kafka



```
)

print(kafka_df)
json_schema = (
    StructType()
    .add("price", StringType(), False)
    .add("time", StringType(), False)
    .add("volume_24h", StringType(), False)
    .add("product_id", StringType(), False)
    .add("best_bid", StringType(), False)
    .add("best_ask", StringType(), False)
)
```

iii) Parse the data from kafka which is provided in string which we need to change to numerical type(double) for numerical dat



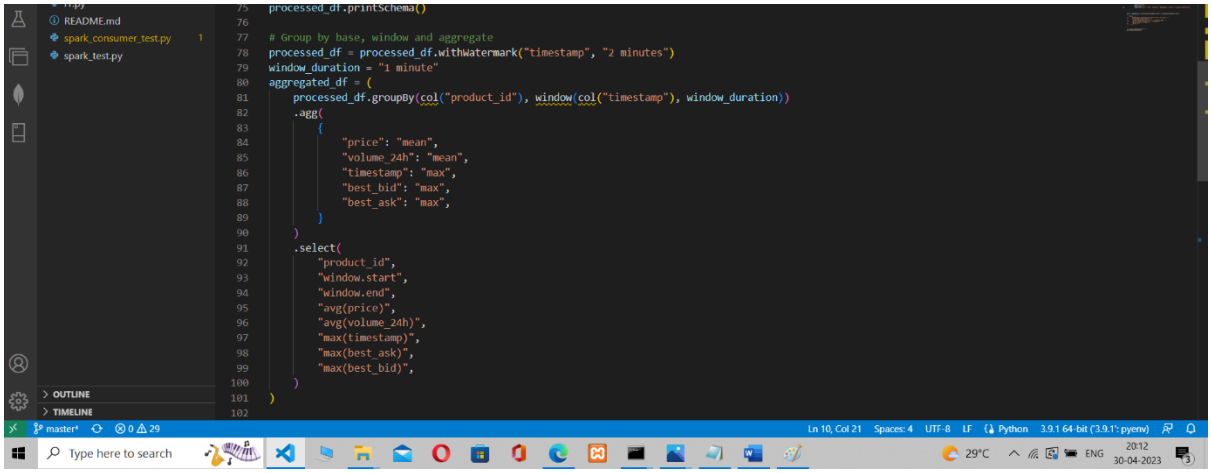
```
# Parse JSON data and extract relevant fields
parsed_df = (
    kafka_df.select(col("value").cast("string").alias("data"))
    .withColumn("jsonData", from_json(col("data"), json_schema, {"mode": "PERMISSIVE"}))
    .select("jsonData.*")
    .select(
        col("price").cast("double"),
        col("best_bid").cast("double"),
        col("best_ask").cast("double"),
        col("product_id"),
        col("time"),
        col("volume_24h").cast("double"),
    )
)

parsed_df.printSchema()
```

Data frame schema we are making use of-

```
root
 |-- product_id: string (nullable = true)
 |-- start: timestamp (nullable = true)
 |-- end: timestamp (nullable = true)
 |-- price: double (nullable = true)
 |-- volume_24h: double (nullable = true)
 |-- time: timestamp (nullable = true)
 |-- best_ask: double (nullable = true)
 |-- best_bid: double (nullable = true)
```

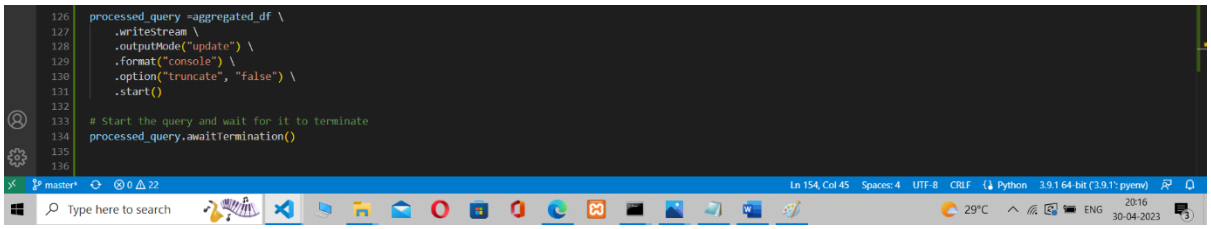
iv) Performing analysis on the streaming data frame

For every one minute we are making a window from the data frame and using it to find the mean of the price and mean of the volume_24h column and finding the best ask (max) and best bid (max) for every one-minute tumbling window.

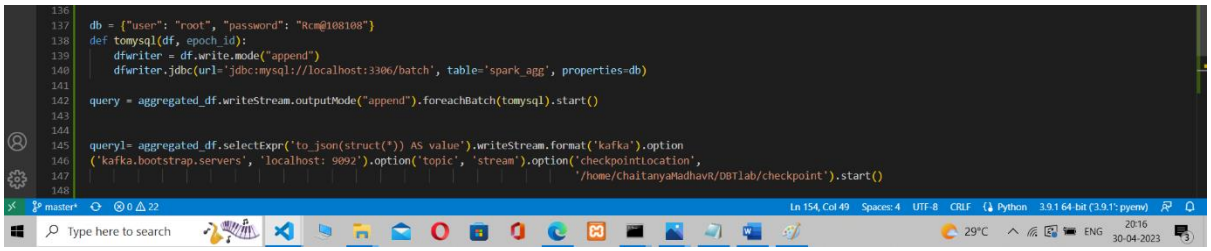


v) Then finally we write the aggregated data from above to Kafka, MySQL, or the console

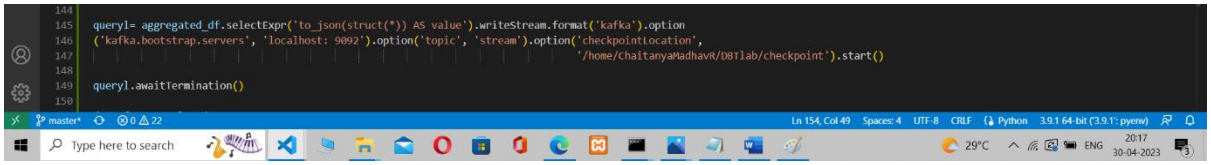
i) Write to the console



ii) Writing the data to MySQL so that it can be used later for batch processing



iii)Writing the data to Kafka so that it can be consumed by the node server



d. Corresponding Results

These are the processed window data which is being written to console

```
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\xampp\mysql\bin>mysql -u root
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 24
Server version: 10.4.24-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use batch;
Database changed
MariaDB [(batch)]>
```

```
-------------------------------------------
Batch: 2
-------------------------------------------
+----------+-------------------+-------------------+-------------------+--------------------+-------------------+--------+--------+
|product_id|              start|                end|              price|          volume_24h|               time|best_ask|best_bid|
+----------+-------------------+-------------------+-------------------+--------------------+-------------------+--------+--------+
|   ETH-USD|2023-04-24 17:08:00|2023-04-24 17:09:00|1817.65242187500005|124946.908058676 23|2023-04-24 17:08:19| 1818.34| 1818.04|
|   LTC-USD|2023-04-24 17:07:00|2023-04-24 17:08:00| 86.60909090909091|186326.17322922635|2023-04-24 17:07:59|   86.62|   86.59|
|   BTC-USD|2023-04-24 17:07:00|2023-04-24 17:08:00|27143.367972972967|11140.097995030676|2023-04-24 17:07:59|27152.84| 27150.2|
|   ETH-USD|2023-04-24 17:07:00|2023-04-24 17:08:00|1817.37026785714 47|124857.57319553176|2023-04-24 17:07:59| 1817.97| 1817.73|
|   LTC-USD|2023-04-24 17:08:00|2023-04-24 17:09:00|               86.63| 186349.3575235656|2023-04-24 17:08:11|   86.65|   86.61|
|   BTC-USD|2023-04-24 17:08:00|2023-04-24 17:09:00| 27153.43571428572|11141.201396046625|2023-04-24 17:08:19|27162.73|27158.58|
+----------+-------------------+-------------------+-------------------+--------------------+-------------------+--------+--------+
```

C:\Windows\System32\cmd.exe - mysql -u root

```
-------------------------------------------
Batch: 6
-------------------------------------------
+----------+-------------------+-------------------+-------------------+--------------------+-------------------+--------+--------+
|product_id|              start|                end|              price|          volume_24h|               time|best_ask|best_bid|
+----------+-------------------+-------------------+-------------------+--------------------+-------------------+--------+--------+
|   ETH-USD|2023-04-24 17:09:00|2023-04-24 17:10:00|1819.6158045977015|125329.54900703751|2023-04-24 17:09:28| 1820.77| 1820.62|
|   ETH-USD|2023-04-24 17:08:00|2023-04-24 17:09:00|1819.0950000000003|125068.89191696404|2023-04-24 17:08:59| 1821.57| 1821.28|
|   LTC-USD|2023-04-24 17:07:00|2023-04-24 17:08:00| 86.60909090909091|186326.17322922635|2023-04-24 17:07:59|   86.62|   86.59|
|   BTC-USD|2023-04-24 17:07:00|2023-04-24 17:08:00|27143.367972972967|11140.097995030676|2023-04-24 17:07:59|27152.84| 27150.2|
|   ETH-USD|2023-04-24 17:07:00|2023-04-24 17:08:00|1817.37026785714 47|124857.57319553176|2023-04-24 17:07:59| 1817.97| 1817.73|
|   LTC-USD|2023-04-24 17:08:00|2023-04-24 17:09:00| 86.74542857142855|186465.79301278145|2023-04-24 17:08:58|   86.84|   86.81|
|   BTC-USD|2023-04-24 17:08:00|2023-04-24 17:09:00|27181.335647058833|11146.709127292024|2023-04-24 17:08:59|27206.56|27206.54|
|   BTC-USD|2023-04-24 17:09:00|2023-04-24 17:10:00| 27186.89214876034|11159.003404502811|2023-04-24 17:09:28|27202.42| 27198.1|
|   LTC-USD|2023-04-24 17:09:00|2023-04-24 17:10:00| 86.75699999999999|  186550.194970313|2023-04-24 17:09:27|    86.8|   86.78|
+----------+-------------------+-------------------+-------------------+--------------------+-------------------+--------+--------+
```

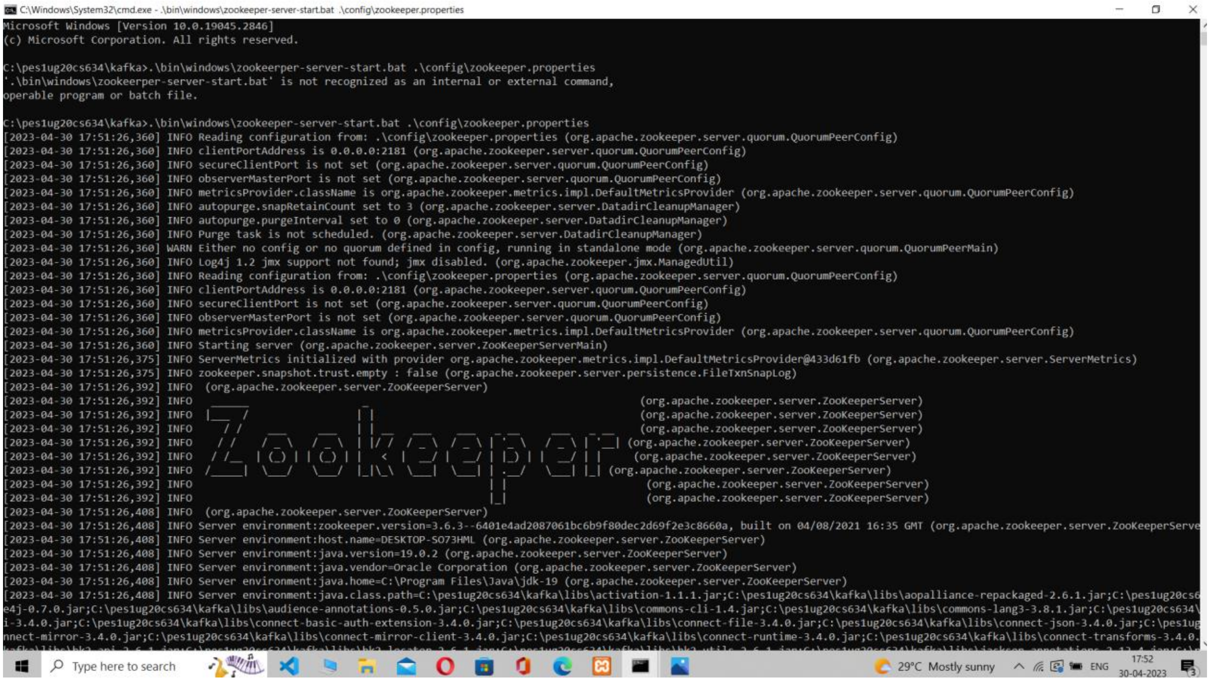
Checking the data on mysql

C:\Windows\System32\cmd.exe - mysql -u root

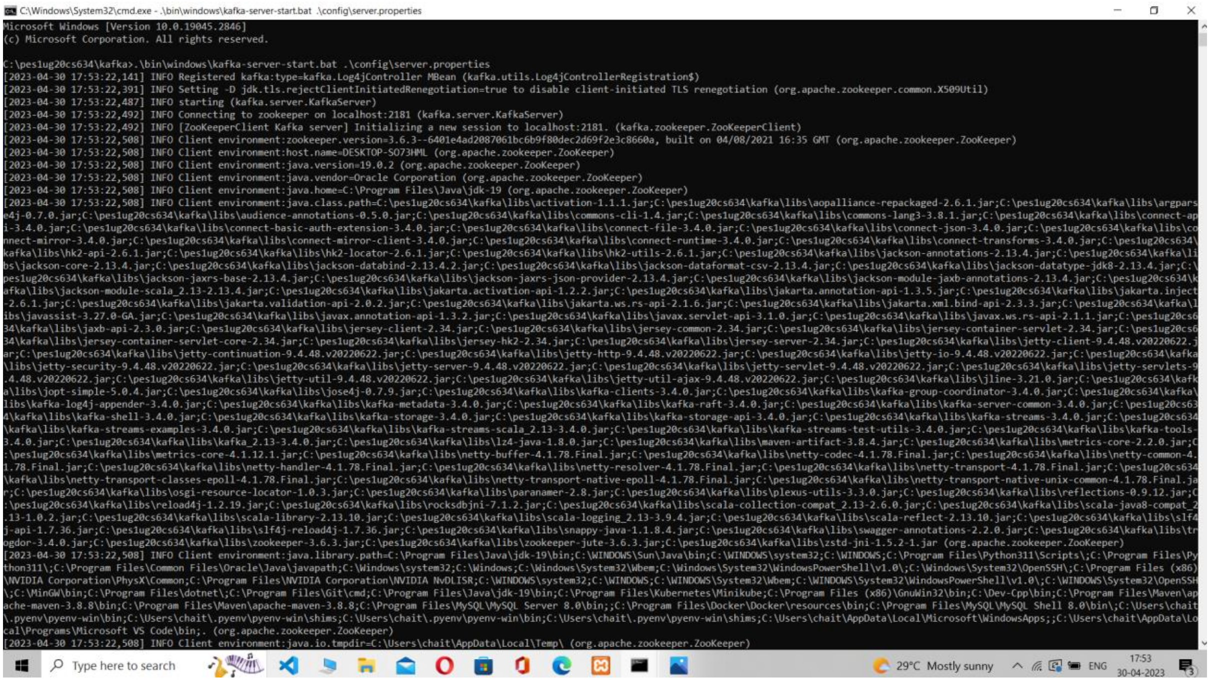
```
MariaDB [(batch)]>select * from spark_agg limit 5 offset 200;
+------------+---------------------+---------------------+-------------+--------------+---------------------+----------+----------+
| product_id | start               | end                 | price       | volume_24h   | time                | best_ask | best_bid |
+------------+---------------------+---------------------+-------------+--------------+---------------------+----------+----------+
| BTC-USD    | 2023-04-23 13:23:00 | 2023-04-23 13:24:00 | 27630.98197 |   7323.01279 | 2023-04-23 13:23:57 | 27632.68 | 27632.67 |
| ETH-USD    | 2023-04-24 17:23:00 | 2023-04-24 17:24:00 |  1827.88410 | 128617.30546 | 2023-04-24 17:23:59 |  1829.08 |  1828.99 |
| ETH-USD    | 2023-04-23 14:42:00 | 2023-04-23 14:43:00 |  1873.18718 |  52047.39715 | 2023-04-23 14:42:59 |  1873.74 |  1873.67 |
| LTC-USD    | 2023-04-24 17:20:00 | 2023-04-24 17:21:00 |    87.40538 | 189082.06881 | 2023-04-24 17:20:56 |    87.50 |    87.46 |
| ETH-USD    | 2023-04-24 16:30:00 | 2023-04-24 16:31:00 |  1837.09924 | 107094.57938 | 2023-04-24 16:30:58 |  1837.63 |  1837.53 |
+------------+---------------------+---------------------+-------------+--------------+---------------------+----------+----------+
5 rows in set (0.000 sec)
```

Results from the React client showing data points

(terminal contains SRN)



(terminal contains SRN)



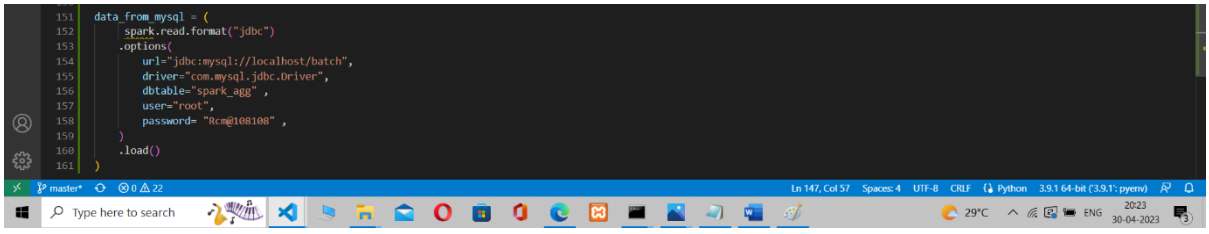
Time of execution 2023-04-30 (20:20) (mentioned in screenshot)

7. Batch Mode Experiment

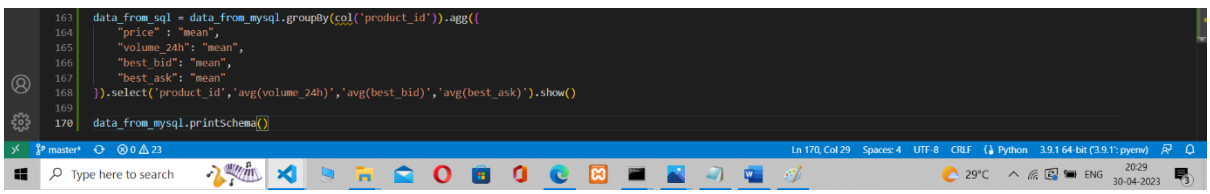
a. Description

Upon pushing streaming data to the MySQL database, we aggregate the data present in the table over a longer period of time. We write to the database using the method described above through python scripting. We aggregate the results from the database, by connecting spark to the database and calling an aggregating method, as shown below. Results obtained from aggregating through regular SQL query and through PySpark are identical as observed.

i) Getting data from mysql and forming a dataframe of it



ii)Running query on the dataframe



b. Data

Data schema-

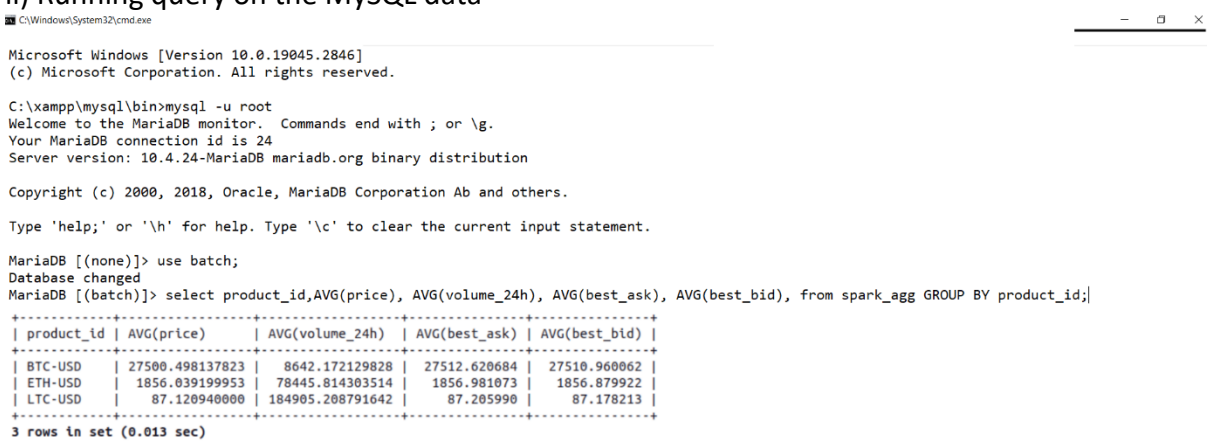
```
root
 |-- product_id: string (nullable = true)
 |-- start: timestamp (nullable = true)
 |-- end: timestamp (nullable = true)
 |-- price: double (nullable = true)
 |-- volume_24h: decimal(20,5) (nullable = true)
 |-- time: timestamp (nullable = true)
 |-- best_ask: double (nullable = true)
 |-- best_bid: double (nullable = true)
```

c. Results

i) Running the query through spark from the data which is stored from MySQL

```
+----------+------------------+----------------+------------------+------------------+
|product_id|        avg(price)| avg(volume_24h)|     avg(best_bid)|     avg(best_ask)|
+----------+------------------+----------------+------------------+------------------+
|   LTC-USD| 87.12093999999998|184905.208791643| 87.17821256038647|  87.2059903381642|
|   ETH-USD|1856.0391999533426| 78445.814303515|1856.8799222395023|1856.9810730948666|
|   BTC-USD| 27500.49813782272|  8642.172129829|27510.960062208393|27512.620684292375|
+----------+------------------+----------------+------------------+------------------+
```

ii) Running query on the MySQL data



8. Comparison of Streaming and Batch modes

- Data Processing: In batch mode, data is collected over a period of time, stored, and then processed as a batch. In streaming mode, data is processed in real-time, as soon as it is generated or received.

- Latency: Batch mode processing is optimized for high throughput, but may have high latency. Streaming mode processing is optimized for low latency, but may have lower throughput.
- Scale: Batch mode processing is better suited for processing large volumes of data in one go, while streaming mode processing is better suited for processing data as it arrives, even if the volume of data is huge.
- Data freshness: In batch mode processing, data may not be fresh as it could be delayed until the batch is processed. In streaming mode processing, data is processed as soon as it arrives, ensuring the freshness of the data.
- Data processing complexity: Batch processing can be simpler as it requires processing a fixed set of data. Streaming processing can be more complex as it requires processing data in real-time, which may involve handling out-of-order data, data loss, and duplicate data.
- Resources required: Batch processing typically requires more resources, such as memory and processing power, to process large batches of data. Streaming processing requires real-time processing and may require specialized hardware or software.
- In summary, batch processing is better suited for processing large volumes of data in one go, while streaming processing is better suited for processing data as it arrives in real-time. Both approaches have their own advantages and disadvantages, and the choice of approach depends on the specific requirements of the application.

s9. Conclusion

In conclusion, our project has successfully addressed the challenge of providing accessible and accurate real-time visualization of Bitcoin data. By relying on an external API to stream and aggregate data, we have created a solution that provides users with up-to-date information in an easily digestible format. Our system stores this data in a database and streams it to a web interface, where it is displayed as a line graph, allowing users to identify trends and patterns quickly.

10. References

https://codedamn.com/news/full-stack/how-to-build-a-websocket-in-node-js
https://recharts.org/en-US
https://www.rittmanmead.com/blog/2017/01/getting-started-with-spark-streaming-with-python-and-kafka/
https://sparkbyexamples.com/spark/spark-parse-json-from-text-file-string/
https://sparkbyexamples.com/spark/spark-streaming-with-kafka/
https://sparkbyexamples.com/spark/spark-convert-string-to-timestamp-format/
https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html
https://www.databricks.com/blog/2017/04/26/processing-data-in-apache-kafka-with-structured-streaming-in-apache-spark-2-2.html
https://kafka-python.readthedocs.io/en/master/
https://www.npmjs.com/package/kafka-node