College: Engineering and Information Technology
Department: Information Technology
Program: Data Analytics

Big Data Technologie course Project

BTC Trades Data Pipeline

Prepared by:
Amnah Khaled – 202210779
Hala Renan – 202211790
Lubna Aslam Sher – 202120102
Al-Anood Tawfeeq Mohsen Al-Sowaidi – 202211607
Huda Mohammed Bilal - 202211270

Supervised by:
Salam Fraihat

Academic Year 2024- 2025 – Spring

Introduction

With today's high-frequency trading and volatile financial markets, real-time analysis of data is the secret to informed decision-making and risk management. The rapid emergence of cryptocurrency exchanges, of which Binance is one of the largest, has created a vast amount of real-time trade data with valuable insights into market trends, anomalies, and investor sentiment. But with such high-velocity, high-volume data comes the issue of deriving valuable intelligence, let alone scalability, fault tolerance, and real-time processing.

This project aims at resolving the challenge of handling live Binance WebSocket trade data efficiently in a horizontally scalable big data design. With streaming data, processing systems proved not to be adequate, especially for applications such as predictive pricing, where latency translates into excessive economic loss or opportunity loss.

To resolve this problem, we have created and implemented an end-to-end big data pipeline with open-source technologies, including Kafka (KRaft mode), Spark Structured Streaming, Hadoop HDFS and MapReduce, InfluxDB, Grafana - all containerized through Docker for reproducibility and portability. The project enables continuous ingestion, transformation, and machine learning over real-time trade data, alongside historical batch processing and interactive visualizations through Grafana.

Objectives

- To design a pipeline for real-time data ingestion and processing from Binance WebSocket trade data.
- To extract useful features from trade data for future machine learning tasks.
- To store useful features from streaming data in a scalable way using InfluxDB (Time – Series Data)
- To store historical data in a scalable way using distributed storage systems (HDFS)
- To perform Aggregations on the historical data using MapReduce
- To apply machine learning models such as Random Forest, Gradient Boosted Trees, and Linear Regression for price prediction.
- To visualise real-time metrics and model outputs using InfluxDB and Grafana.

In order to ensure the pipeline is scalable, fault-tolerant, and in line with modern data engineering practices.

Task and data

What is Binance?

Binance is a cryptocurrency exchange and one of the largest crypto currencies in the world that lists more than 350 cryptocurrencies globally. It offers several services that enhance the experience for users and blockchain developers. It provides real-time data on crypto asset prices, trading activity, and market movements through its public API’s and WebSocket endpoints.

How are we using Binance in our project?

In our real-time streaming project for big data, we are utilizing Binance's WebSocket API to extract live cryptocurrency trade data. Our project focuses on building a real-time streaming pipeline that simulates a data ingestion and pre-processing system using the following technologies:

- WebSocket (Producer): Connects to Binance and streams live trade data.
- Apache Kafka: Acts as the message broker (communicator) between the producer and consumer.
- Apache Spark (PySpark (part of Spark)): Consumes the data from Kafka and processes it in real time.

Currently, our focus is getting streaming data information related to trading pair BTCUSDT- Bitcoin/ Tether where Bitcoin is traded against Tether (a stable coin).

Where are we getting the Data from?

The data we are getting from is from Binance's public WebSocket endpoint:

wss://stream.binance.com:9443/ws/btcusdt@trade

This endpoint (connects physical devices and exchanges information with a computer network) provides real-time trade events for BTCUSDT trading pair. Each trade message is transmitted in JSON format, containing detailed information about the trade.

What do the Binance Trade Fields mean in our Project?

Here is a table containing breakdown of each field we are using in our Binance WebSocket stream:

Field	Key	Type	Meaning
e	String	Event Type	Always 'Trade' for trade stream
E	Long	Event Time	Timestamp when the event occurred (in ms)
s	String	Symbol	Trading pair symbol
t	Long	Trade ID	Unique Trader ID
p	Long	Price	Trade price
q	String	Quantity	Quantity traded
b	String	Buyer Order ID	Order ID of the buyer
a	Long	Seller Order ID	Order ID of the seller
T	Long	Trade Time	Timestamp when the trade was executed
m	Boolean	Is Buyer the Market Maker?	'True' if the buyer is the market maker
M	Boolean	Ignore	Always 'true'; can be ignored.

Are we using the streaming Data to train our model as well?

No, for training our model's, we use Binance historical trading:

https://data.binance.vision/?prefix=data/spot/daily/trades/BTCUSDT/

In the historical data, we have 7 features:

Trade_ID (t), Price (p), Quantity (q), Quote_Quantity (p * q), Trade_Time (T), Market Maker (m), Flag (M)

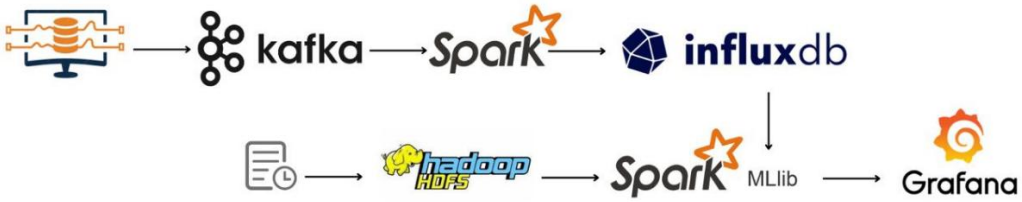
	A	B	C	D	E	F	G
1	4.78E+09	78430	0.00064	50.1952	1.74E+15	FALSE	TRUE
2	4.78E+09	78430	0.00026	20.3918	1.74E+15	FALSE	TRUE
3	4.78E+09	78429.99	0.00021	16.4703	1.74E+15	TRUE	TRUE
4	4.78E+09	78429.99	0.00054	42.35219	1.74E+15	TRUE	TRUE
5	4.78E+09	78430	0.00007	5.4901	1.74E+15	FALSE	TRUE
6	4.78E+09	78430	0.00007	5.4901	1.74E+15	FALSE	TRUE
7	4.78E+09	78430	0.00007	5.4901	1.74E+15	FALSE	TRUE
8	4.78E+09	78430	0.00191	149.8013	1.74E+15	FALSE	TRUE
9	4.78E+09	78430	0.00039	30.5877	1.74E+15	FALSE	TRUE
10	4.78E+09	78430	0.00017	13.3331	1.74E+15	FALSE	TRUE
11	4.78E+09	78430	0.00201	157.6443	1.74E+15	FALSE	TRUE
12	4.78E+09	78430	0.00007	5.4901	1.74E+15	FALSE	TRUE
13	4.78E+09	78430	0.02226	1745.852	1.74E+15	FALSE	TRUE

Can you describe the data in terms of volume, veracity, variety, and velocity aspect?

- Veracity: Our data is high quality, and we can trust its integrity because it is coming credible source like Binance.
- Variety: Our raw data is unstructured, which is then converted to Json String (semi – structured), and then to spark's RDD (structured)
- Velocity: Our data is continuously streaming. Our data has very high velocity. On average, around 50 – 300 trades take place per second.
- Volume: Due to the data's high velocity, we have huge volumes of data. A csv file of trades of one day is around 500 MB in size, whereas for one month, it's around (10 GB)

Methodology

Our Big data Architecture



Historical Data Architecture:

- Historical trades data (csv) stored in Hadoop's HDFS
- Aggregations performed (1 minute average's) on the data using mapReduce (Output stored in HDFS)
- Spark extracts the output and processes it for training our model
- Pyspark.ml used for creating a pipeline for our models (Once models are trained, we store the models)

Streaming Data Architecture:

- Apache Kafka ingests data from binance WebSocket
- The data ingested by kafka is consumed by apache spark
- Spark performs preprocessing on the Data
- After pre-processing, the data is stored in Influxdb
- We load the pre- trained models stored in hdfs, we load data from influx, and then test our model on this data.
- We store results back in influxdb, and then perform visualizations on it using Grafana)

What is Kafka?

Kafka is an open-source distributed event streaming platform, used for high throughput, fault tolerant and real time data pipelines.

Why are we using Kafka?

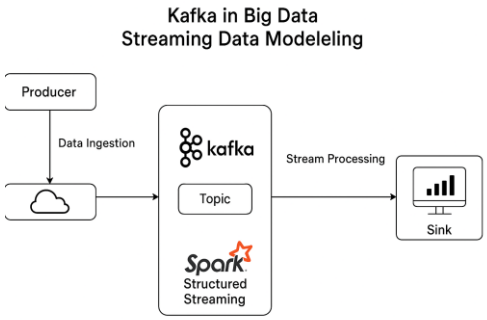
We used Kafka to ingest real-time data from Binance WebSocket streams. It was the best fit in terms of Scalability (horizontally Scalable) which means we can add brokers whenever the load is increased, Decoupled architecture (producers and consumers are independent (better for maintenance and flexibility)), and it integrates natively with spark.

Major Kafka Components:

- Producer: Publishes data on Kafka topics.
- Topic: A category or feed name to which data are published.
- Broker: Kafka server where data are kept.
- Consumer: Subscribes to topics and reads incoming data.
- Zookeeper: Manages Kafka's metadata. (we didn't use because newer version of Kafka can work without it by using KRaft Mode)

Pipeline of Kafka

- Data Ingestion: Kafka ingests real-time data from sources like IoT devices, sensors, live streaming websites, logs, user activity, etc.
- Stream Processing with Spark: Spark reads the stream data in micro-batches from Kafka and consumes from Kafka. And transformation is possible at this stage like (joining, aggregating, windowing, filtering, etc.).
- Sink: Output can be written back to Kafka, or databases, or dashboards (We are writing to InfluxDB)



Spark

What is Spark/pySpark?

PySpark is Apache Spark's Python interface. PySpark assists Python developers in writing Spark applications using Python APIs and running distributed data processing operations without any issues. Spark is coded in Scala, yet PySpark has bindings to enable its powerful distributed data processing capabilities in Python. PySpark allows interacting with Resilient Distributed Datasets (RDDs), DataFrames, and Structured Streaming, etc.

Modelling: use pre-trained models in Spark to make real-time predictions. Or update models incrementally with libraries like MLlib or custom logic.

Why are we using Pyspark?

Because it is giving us two main things

1. Scalability for bigdata and
2. Friendly python syntax

It is making our task easier by helping us in:

- Preprocessing large data sets, from Kafka (real-time Binance WebSocket) and from HDFS (historical data about Binance WebSocket).
- Train our machine learning models.
- It integrates seamlessly with Kafka, influxDB, HDFS.

InfluxDB

What is InfluxDB?

Open-source time-series data base that is widely used to store, query, and analyse time-series data. It also handles high volumes of time-stamped data and provide powerful querying capabilities and real-time analytics. We started by using Cassandra, but failed in connecting the nodes of Cassandra efficiently, which resulted in the shutting down of node due to limited CPU resources well. Which is why, we then planned to use InfluxDB.

Why are we using InfluxDB?

Influx-DB is a powerful and scalable time-series database that's specifically designed for handling high-volume, time-stamped data. In our project, it enabled efficient storage of real-time Binance

WebSocket trade data it also provided low-latency performance for fast data retrieval. Its seamless integration with Grafana allowed us to visualize streaming data through dynamic dashboards, making real-time monitoring and analytics both intuitive and high-performing.

Hadoop

What is Hadoop – HDFS/MapReduce?

Hadoop is an open-source software platform used to store and process bigdata sets on distributed clusters of computers. HDFS is a distributed file system that's fault tolerant and stores big data files on multiple nodes. MapReduce is a programming model that's used to process the bigdata parallelly, it maps tasks into mappers and reducers for efficient and distributed processing .

Why are we using it?

It helped us to aggregate averages on minutes using the MapReduce, in a scalable, fault tolerant and seamless integration with spark.

Grafana

What is Grafana?

Grafana is an open-source visualization and analytics platform that enables monitoring and analysing time-series data.

Why are we using Grafana

Grafana supports influx dB so it helped us visualize our Binance trade data, create a dashboard.

Docker

Docker is an open-source containerization platform that enables developers to package applications along with all their dependencies into isolated, lightweight units called containers. These containers can run consistently across any environment — from development to production — without compatibility issues.

We used docker to run multiple services (Kafka, Spark, HDFS, InfluxDB, Grafana) isolated into containers, avoid software conflicts and ensure each component runs within a predictable environment, simplify data pipeline scalability and deployment using Docker Compose, and reduce the overhead of manual tool setup or creating various environments.

Machine Learning Model - Price Prediction

1. Linear Regression

Linear regression is a supervised machine learning algorithm that predicts continuous values (e.g., avg_price). It assumes a linear relationship between input features (like quantity) and the target variable.

- Goal: Predict the average future price of BTCUDST using the *avg_price*.
- Input Features:
  - ⇒ Scaled numerical features: avg_quantity, avg_quote_quantity, Market_Maker%
  - ⇒ Time-based features: dayofweek, hour, minutes.
- Target Variable: *avg_price.* (continuous values)

Key Parameters

Parameters	**Value**	**Purpose**
featuresCol	"features"	Combined feature vector after scaling and assembly
labelCol	"avg_price"	Target variable to predict
regParam	Default (0)	L2 regularization (not explicitly set here)
elasticNetParam	Default (0)	Mix of L1/L2 regularization (not set)

Implementation Steps:

- Feature Engineering: Scaled numerical feature (StandardScaler) to normalize.
- Combined scaled and time features to a single vector using VectorAssembler.

Evaluation Metrics:

- RMSE (Root Mean Squared Error): measures prediction error magnitude.
- R^2 (R-squared): Explains variance in the target variable.

Random Forest Regressor

An assemble learning method that builds multiple decision trees and averages their predictions to improve accuracy and control overfitting.

Input Features:

- All Features: *dayofweek, hour, minutes, avg_quantity, avg_quote_quantity, Market_Maker%*

Target Variable: *avg_price*.

Key Parameters

Parameter	Value	Purpose
numTrees	150	Number of decision trees (higher = better accuracy but slower)
maxDepth	7	Maximum depth of each tree (deeper = more complex patterns)
maxBins	128	Number of bins for splitting continuous features
featuresCol	"features"	Combined feature vector
labelCol	"avg_price"	Target variable

Implementation Steps:

- Feature Assembly: Directly combine all features without scaling using VectorAssembler.

Evaluation Metrics:

- Evaluate the performance using RMSE and R^2

Gradient-Boosted Trees (GBT) Regressor

An assemble model that builds trees sequentially where each tree corrects error from the previous ones often more accurate than Random Forest but slower.

- Input Features: *dayofweek, hour, minutes, avg_quantity, avg_quote_quantity, Market_Maker%*
- Target Variable: *avg_price*.

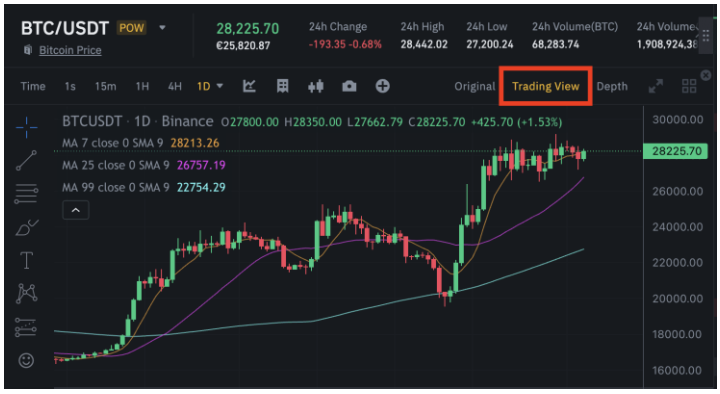
If Gradient-Boosted Trees are sequential then why are we using it in spark?
Yes, GBT is sequential because it creates a new tree by correcting the error of the previous tree. But, when building each tree, within each tree the job is parallelized (distributing data partitions, etc).

Key Parameters

Parameter	Value	Purpose
maxIter	150	Number of boosting iterations (similar to numTrees in RF)
maxDepth	7	Depth of each tree
maxBins	128	Bins for splitting continuous features
featuresCol	"features"	Feature vector
labelCol	"avg_price"	Target variable

Implementation Steps:

- Feature Assembly: Combine features using VectorAssembler.
- Evaluation Metrics: Evaluate the model's performance using RMSE and R^2



The Docker-compose File

```
networks:
  sparkkafkapipeline:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: "172.18.0.0/18"

services:

  kafka:
    image: 'bitnami/kafka:latest'
    container_name: kafka
    hostname: kafka
    ports:
      - '9092:9092'
      - '29092:29092'
      - '9093:9093'
    environment:
      - KAFKA_CFG_NODE_ID=0
      - KAFKA_CFG_PROCESS_ROLES=controller,broker
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=0@kafka:9093
      - KAFKA_CFG_LISTENERS=PLAINTEXT://0.0.0.0:9092,EXTERNAL://0.0.0.0:29092,CONTROLLER://:9093
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092,EXTERNAL://localhost:29092
      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,EXTERNAL:PLAINTEXT
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
      - KAFKA_CFG_INTER_BROKER_LISTENER_NAME=PLAINTEXT
    volumes:
      - kafka_data:/bitnami/kafka/data
    networks:
      sparkkafkapipeline:
        ipv4_address: 172.18.0.4
```

custom bridge network so that containers can talk to each other directly

all containers will get an address in this subnet

cluster-internal listener
external listener
Kraft controller port

Single-node KRaft-mode Kafka (no ZooKeeper)

one for inter-container traffic (PLAINTEXT://kafka:9092),

one bound to localhost:29092 so our local apps can produce/consume.

- Kafka is receiving and holding real-time trade data from Binance, which will be consumed by Spark.
- Producers our Python WebSocket script will send data to Kafka and then Spark Structured Streaming jobs read from Kafka's internal listener at kafka:9092 and it is
- It's accessible externally on localhost:29092 for debugging or monitoring from the machine.
- Kafka is acting as the central message broker decoupling data ingestion from processing and storage.

```yaml
spark-master:
  image: bitnami/spark:latest
  container_name: spark-master
  ports:
    - "8080:8080"
    - "7077:7077"
  volumes:
    - ./spark-apps:/opt/spark-apps
    - ./output:/opt/spark/output
    - ./extra-jars:/opt/bitnami/spark/extra-jars
  environment:
    - SPARK_MODE=master
    - SPARK_LOCAL_IP=172.18.0.2
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.2

spark-worker-a:
  image: bitnami/spark:latest
  container_name: spark-worker-a
  depends_on:
    - spark-master
    - kafka
  ports:
    - "8081:8080"
  environment:
    - SPARK_MODE=worker
    - SPARK_MASTER_URL=spark://172.18.0.2:7077
    - SPARK_WORKER_MEMORY=1G
    - SPARK_WORKER_CORES=1
    - SPARK_LOCAL_IP=172.18.0.3
  volumes:
    - ./spark-apps:/opt/spark-apps
    - ./output:/opt/spark/output
    - ./extra-jars:/opt/bitnami/spark/extra-jars
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.3

spark-worker-b:
  image: bitnami/spark:latest
  container_name: spark-worker-b
  depends_on:
    - spark-master
    - kafka
  ports:
    - "8082:8080"
  environment:
    - SPARK_MODE=worker
    - SPARK_MASTER_URL=spark://172.18.0.2:7077
    - SPARK_WORKER_MEMORY=1G
    - SPARK_WORKER_CORES=1
    - SPARK_LOCAL_IP=172.18.0.6
  volumes:
    - ./spark-apps:/opt/spark-apps
    - ./output:/opt/spark/output
    - ./extra-jars:/opt/bitnami/spark/extra-jars
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.6

spark-worker-c:
  image: bitnami/spark:latest
  container_name: spark-worker-c
  depends_on:
    - spark-master
    - kafka
  ports:
    - "8083:8080"
  environment:
    - SPARK_MODE=worker
    - SPARK_MASTER_URL=spark://172.18.0.2:7077
    - SPARK_WORKER_MEMORY=1G
    - SPARK_WORKER_CORES=1
    - SPARK_LOCAL_IP=172.18.0.5
  volumes:
    - ./spark-apps:/opt/spark-apps
    - ./output:/opt/spark/output
    - ./extra-jars:/opt/bitnami/spark/extra-jars
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.5
```

Spark UI

Spark master port-worker registration & job submission

Each worker is connected to the master and has its own unique IP.

1 CPU core and 1 GB RAM each for testing.

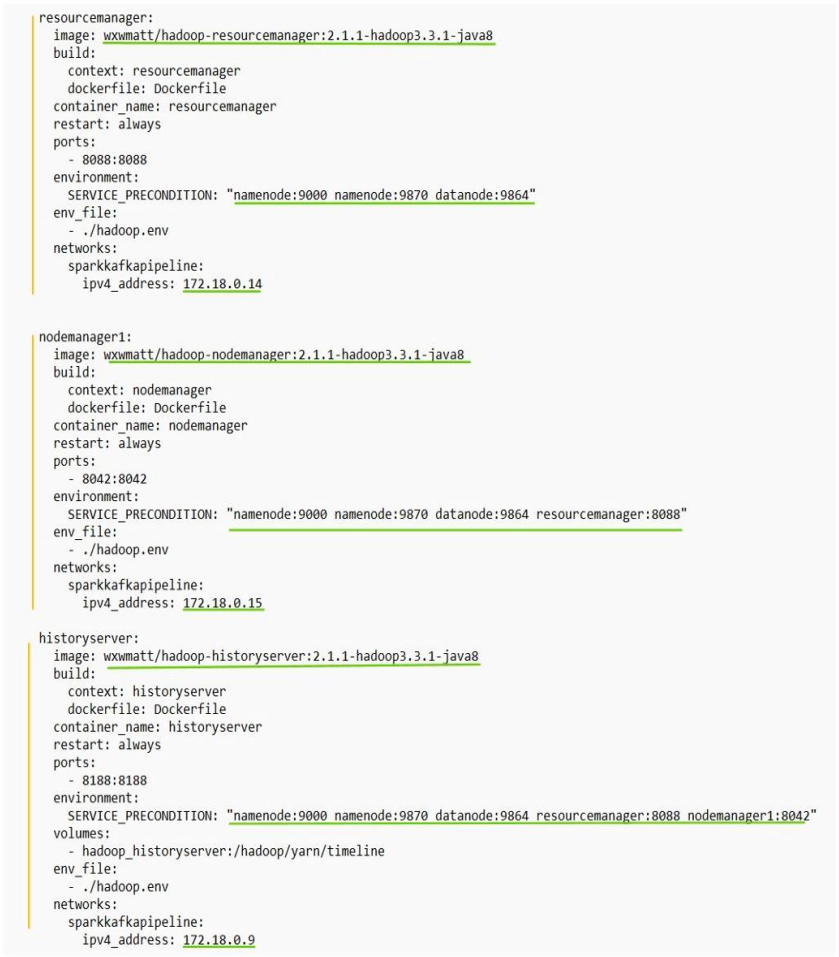
- Spark is consuming data from kafka and is monitoring the three Spark Workers (worker-a, b, c) that do the heavy lifting.
- Spark Workers connect to it at 172.18.0.2:7077.
- Spark Master acts as the brain of the distributed processing. It schedules and manages all Spark jobs and executors.
- Spark worker(a,b,c) are slave nodes in the Spark cluster. They execute tasks assigned by the master, using specified allocated memory and CPU. Spark Workers connect to the Spark Master and waits for jobs. When a job (like a data processing pipeline) is submitted, each worker run a part of the computation (like one stage or partition of a dataset).
- spark-apps: the job Python code
- output: job results will be saved here
- extra-jars: additional Spark or Kafka libraries
- The workers are part of the scalable processing layer that lets Spark distribute and parallelize machine learning, feature extraction, and stream/batch jobs.

```yaml
namenode:
  image: wxwmatt/hadoop-namenode:2.1.1-hadoop3.3.1-java8
  build:
    context: namenode
    dockerfile: Dockerfile
  container_name: namenode
  hostname: namenode  # 🔑 Important for HDFS resolution
  restart: always
  ports:
    - 9870:9870
    - 9000:9000
  volumes:
    - hadoop_namenode:/hadoop/dfs/name
    - ./python_mapreduce:/app
  working_dir: /app
  environment:
    - CLUSTER_NAME=test
    - HOME=/app
    - SPARK_HOME=/usr/local/lib/python3.7/dist-packages/pyspark
    - HIVE_HOME=/hive/apache-hive-3.1.3-bin
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.7

datanode:
  image: wxwmatt/hadoop-datanode:2.1.1-hadoop3.3.1-java8
  build:
    context: datanode
    dockerfile: Dockerfile
  container_name: datanode
  restart: always
  ports:
    - 9864:9864
  volumes:
    - hadoop_datanode:/hadoop/dfs/data
  environment:
    SERVICE_PRECONDITION: "namenode:9000"
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.8

datanode2:
  image: wxwmatt/hadoop-datanode:2.1.1-hadoop3.3.1-java8
  build:
    context: datanode
    dockerfile: Dockerfile
  container_name: datanode2
  restart: always
  ports:
    - 9865:9864  # New port if you want to expose it
  volumes:
    - hadoop_datanode2:/hadoop/dfs/data
  environment:
    SERVICE_PRECONDITION: "namenode:9000"
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.12


datanode3:
  image: wxwmatt/hadoop-datanode:2.1.1-hadoop3.3.1-java8
  build:
    context: datanode
    dockerfile: Dockerfile
  container_name: datanode3
  restart: always
  ports:
    - 9866:9864  # New port if you want to expose it
  volumes:
    - hadoop_datanode3:/hadoop/dfs/data
  environment:
    SERVICE_PRECONDITION: "namenode:9000"
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.13
```

NameNode UI

HDFS RPC

- Hadoop Name Node is responsible for controlling and managing all file access in HDFS, accepting requests from Spark jobs to read/write files, and oversees Data Nodes that are actually storing data blocks .
- The Data Nodes, receive file chunks Binance data, Stores them as blocks on disk and they Regularly update the Name Node about which blocks each is holding. All Data Nodes store blocks of files in a distributed and replicated manner.



```
resourcemanager:
  image: wxwmatt/hadoop-resourcemanager:2.1.1-hadoop3.3.1-java8
  build:
    context: resourcemanager
    dockerfile: Dockerfile
  container_name: resourcemanager
  restart: always
  ports:
    - 8088:8088
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864"
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.14

nodemanager1:
  image: wxwmatt/hadoop-nodemanager:2.1.1-hadoop3.3.1-java8
  build:
    context: nodemanager
    dockerfile: Dockerfile
  container_name: nodemanager
  restart: always
  ports:
    - 8042:8042
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864 resourcemanager:8088"
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.15

historyserver:
  image: wxwmatt/hadoop-historyserver:2.1.1-hadoop3.3.1-java8
  build:
    context: historyserver
    dockerfile: Dockerfile
  container_name: historyserver
  restart: always
  ports:
    - 8188:8188
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864 resourcemanager:8088 nodemanager1:8042"
  volumes:
    - hadoop_historyserver:/hadoop/yarn/timeline
  env_file:
    - ./hadoop.env
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.9
```

- ResourceManager accepts job submissions like MapReduce scripts or Spark-on-YARN jobs, schedules tasks to run on Node Managers , tracks application lifecycle (start, fail, finish) and monitors overall cluster usage through the Web UI.
- It communicates with the Node Manager to launch and monitor containers,
- with History Server to archive completed job data and Name Node
- to verify input/output locations in HDFS .

- Node Manager is a worker node in YARN. It launches and monitors containers (tasks) assigned by the Resource Manager. It Runs MapReduce or Hadoop Streaming jobs in YARN containers and reports container status (running, failed, finished) back to the Resource Manager. It communicates with the Resource Manager 172.18.0.14→ to receive and run tasks , Name Node 172.18.0.7 + Data Nodes→ to access HDFS files.
- Node Manager is the execution layer in YARN—it physically runs the containerized jobs (Python scripts, mappers, reducers) across the cluster. This is where real work happens!
- History Server is a Hadoop YARN service that provides a UI to browse completed MapReduce job logs (task attempts, counters, output paths, and errors).
- It helps in Collecting and displaying historical job information from HDFS, debug or verify job execution after its finished and supports reproducibility and post-analysis of jobs. It is connected by Reading job logs from HDFS (so depends on NameNode + DataNodes) and it must wait for ResourceManager and NodeManager to be ready since they write job history.
- HistoryServer is our diagnostics and monitoring dashboard for finished jobs. After running MapReduce scripts (like aggregating Binance data), we can view their input, logs, counters, and results from this service.

```yaml
influxdb2:
  image: influxdb:2
  container_name: influxdb2
  ports:
    - "8086:8086"
  environment:
    DOCKER_INFLUXDB_INIT_MODE: setup
    DOCKER_INFLUXDB_INIT_USERNAME: admin
    DOCKER_INFLUXDB_INIT_PASSWORD: adminpassword
    DOCKER_INFLUXDB_INIT_ADMIN_TOKEN: supersecrettoken
    DOCKER_INFLUXDB_INIT_ORG: myorg
    DOCKER_INFLUXDB_INIT_BUCKET: home
  volumes:
    - influxdb2-data:/var/lib/influxdb2
    - influxdb2-config:/etc/influxdb2
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.10

grafana:
  image: grafana/grafana:latest
  container_name: grafana
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin
  depends_on:
    - influxdb2
  networks:
    sparkkafkapipeline:
      ipv4_address: 172.18.0.11
```

```yaml
volumes:
  kafka_data:
    driver: local
  influxdb2-data:
  influxdb2-config:
  hadoop_namenode:
  hadoop_datanode:
  hadoop_datanode2:
  hadoop_datanode3:
  hadoop_historyserver:
```

Kafka Logs

InfluxDB storage

InfluxDB config

HDFS NameNode metadata

HDFS DataNode blocks: node1, node2, node3

Yarn history data

- InfluxDB stores real-time Binance trade data, allows us to query and analyse the data using flux, it also Acts as a data source for Grafana dashboards. So, we are pushing high-frequency, recent data to Influx DB for fast querying and dashboards.
- Grafana connects to InfluxDB to visualize Binance trade metrics or Spark job outputs and enable us to create time-series dashboards with charts, gauges, heatmaps **and alerts** it also Helps in monitoring, debugging, and presentation of insights.

(To run the above docker-compose file refer to the environment configuration setup pdf.)

But in general, a docker-compose file is run using the following command.

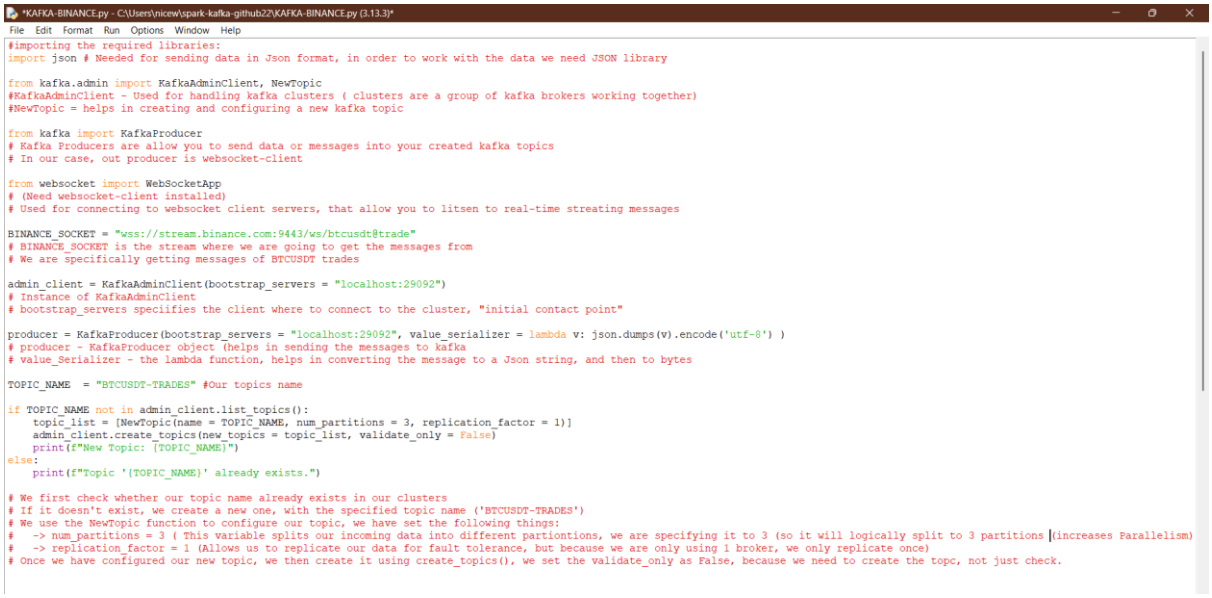
```
docker-compose up -d
```

Experiments

(In the configuration environment, we have described in detail how to set up our docker, yml file and the respective python scripts)

Experiment 1: Ingesting Data using Kafka

- **Aim**: Successfully extract streaming data from Binance WebSocket using Kafka.



```
#importing the required libraries:
import json # Needed for sending data in Json format, in order to work with the data we need JSON library

from kafka.admin import KafkaAdminClient, NewTopic
#KafkaAdminClient - Used for handling kafka clusters ( clusters are a group of kafka brokers working together)
#NewTopic = helps in creating and configuring a new kafka topic

from kafka import KafkaProducer
# Kafka Producers are allow you to send data or messages into your created kafka topics
# In our case, out producer is websocket-client

from websocket import WebSocketApp
# (Need websocket-client installed)
# Used for connecting to websocket client servers, that allow you to litsen to real-time streating messages

BINANCE_SOCKET = "wss://stream.binance.com:9443/ws/btcusdt@trade"
# BINANCE_SOCKET is the stream where we are going to get the messages from
# We are specifically getting messages of BTCUSDT trades

admin_client = KafkaAdminClient(bootstrap_servers = "localhost:29092")
# Instance of KafkaAdminClient
# bootstrap_servers speciifies the client where to connect to the cluster, "initial contact point"

producer = KafkaProducer(bootstrap_servers = "localhost:29092", value_serializer = lambda v: json.dumps(v).encode('utf-8') )
# producer - KafkaProducer object (helps in sending the messages to kafka
# value_Serializer - the lambda function, helps in converting the message to a Json string, and then to bytes

TOPIC_NAME  = "BTCUSDT-TRADES" #Our topics name

if TOPIC_NAME not in admin_client.list_topics():
    topic_list = [NewTopic(name = TOPIC_NAME, num_partitions = 3, replication_factor = 1)]
    admin_client.create_topics(new_topics = topic_list, validate_only = False)
    print(f"New Topic: {TOPIC_NAME}")
else:
    print(f"Topic '{TOPIC_NAME}' already exists.")

# We first check whether our topic name already exists in our clusters
# If it doesn't exist, we create a new one, with the specified topic name ('BTCUSDT-TRADES')
# We use the NewTopic function to configure our topic, we have set the following things:
#   -> num_partitions = 3 ( This variable splits our incoming data into different partiontions, we are specifying it to 3 (so it will logically split to 3 partitions (increases Parallelism)
#   -> replication_factor = 1 (Allows us to replicate our data for fault tolerance, but because we are only using 1 broker, we only replicate once)
# Once we have configured our new topic, we then create it using create_topics(), we set the validate_only as False, because we need to create the topc, not just check.
```

```
def on_message(WebSocket_client_Obj, message):
    data = json.loads(message)
    print("Sending data to Kafka:", data)
    producer.send(TOPIC_NAME, value = data)
# The on_message() function, takes as input the message from the stream, and a Web_socket client
# It converts the message to Json string and then sends it to kafka using the producer client

def on_error(WebSocket_client_Obj, error):
    print("WebSocket error:", error)
# The on_error() fucntion is called whenever there is an error in the conenction

def on_close(WebSocket_client_Obj):
    print("WebSocket closed")
# The on_close() fucntion is called when the connection is closed

def on_open(WebSocket_client_Obj):
    print("WebSocket connection opened")
# The on_open() fucntion is called when the connection is opened

WebSocket_client_Obj = WebSocketApp(
    BINANCE_SOCKET,
    on_message=on_message,
    on_error=on_error,
    on_close=on_close,
    on_open=on_open
)

# WebSocket_client_Obj is as the name states a client of the webSocket App
# We specify to it out URL (BINANCE_SOCKET) and our callbacks (which tells what to do in case of a certain event)
# So WebSocket_client_Obj is creating a connection to our binance trades stream.


WebSocket_client_Obj.run_forever() #Specifies it to continuously run and stream data, unless the connection is closed
```

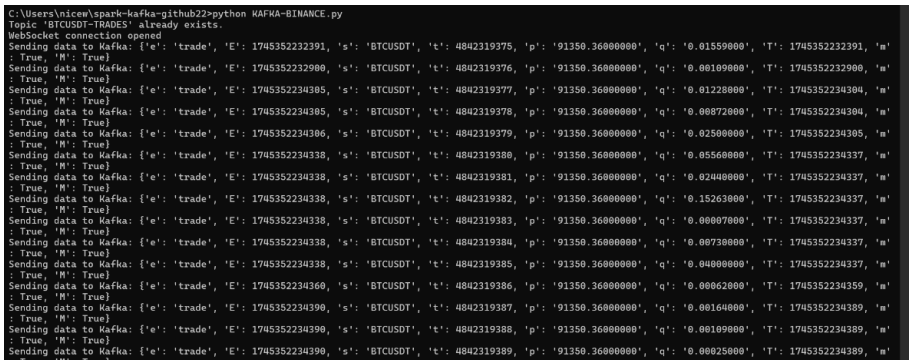
Explaining the process:

- We start by creating an admin_client that handles the Kafka clusters. We use this client to create a Kafka topic 'BTCUSDT-TRADES'. While creating the topic we have set the replication factor as 1 (since we are only using one broker), we have also set the number of partitions as 3, which creates parallelism while streaming messages. (In order to achieve true parallelism and replication, it is encouraged to use more than one broker. But due to our issues with connecting to the specific ports, we decided to stick with one broker)
- We then subscribe to the Binance WebSocket stream btcusdt@trade, which publishes real-time BTC/USDT trade messages. We configure and create a KafkaProducer client that forwards each trade message to the kafka topic BTCUSDT-TRADES.

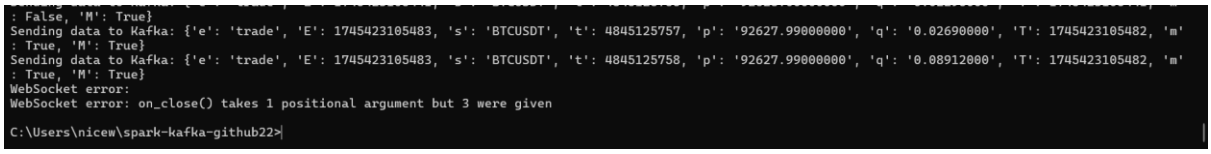
- Kafka is using port 29092 as the bootstrap_server (used by other tools to connect to kafka)
- A WebSocket client (WebSocketApp) handles real-time receiving of data, invoking on_message() for all trade messages.
- Each message was serialized to JSON and sent over the Kafka topic using producer.send().

We run the above Kafka script on cmd terminal by entering the following command in our projects folder:

```
Python KAFKA-BINANCE.py
```



We can also close the stream using CTRL + C



Results:

- Our websocket_client can successfully open the connection
- We can successfully read messages from the websocket stream using kafka.
- Our websocket_client can successfully close the connection

Experiment 2: Processing the streaming data using pyspark

- **Aim**:
    - Connect pyspark with kafka
    - Process the streaming data in spark
    - Store the processed data in influxDB

Once we have successfully connected the WebSocket client with Kafka, and we have made sure that our data is streaming continuously, we then run the spark script on a different terminal. This spark script makes connection with Kafka at the port 9092, and reads its incoming messages, process it, and then stores it in InfluxDB. So, pyspark is our Kafka's consumer.

```
*Processing_Storing_Streaming_Spark.py - C:\Users\nicew\spark-kafka-github22\spark-apps\Processing_Storing_Streaming_Spark.py (3.13.3)*
File  Edit  Format  Run  Options  Window  Help
from pyspark.sql import SparkSession #Required for creating a Spark Session
from pyspark.sql.functions import when, from_unixtime, col, date_format, minute, hour, dayofweek, count, sum, avg, to_date #Data Manipulation or Processing Functions
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StringType, BooleanType #Schema for the incoming kafka messages
import requests #Needed for sending HTTP requests to influxDB

# We need a SparkSession to use Spark. SparkSession gives us access to its clusters and features
spark = SparkSession.builder \
    .appName("StreamKlineToInfluxDB") \
    .master("spark://spark-master:7077") \
    .config("spark.jars", "/opt/bitnami/spark/extra-jars/spark-sql-kafka-0-10_2.12-3.5.0.jar,/opt/bitnami/spark/extra-jars/spark-token-provider-kafka-0-10_2.12-3.5.0.jar, \
            /opt/bitnami/spark/extra-jars/kafka-clients-3.5.0.jar,/opt/bitnami/spark/extra-jars/commons-pool2-2.11.1.jar") \
    .getOrCreate()
# Unique name for each sessions for logs
# Creates the spark session
# .master() Informs spark about the cluster location, which is where our spark-master is located at port 7077
# In order to connect spark with kafka, we need to mention specific JAR files in config(), which have been downloaded and stored in folder extra-jars

# Since our data is in Json format from kafka, we need to specify the structure of out data for spark
# Although, the data contains more features like TradeID etc, since we do not require it, we can omit it out
schema = StructType() \
    .add("s", StringType()) \
    .add("p", StringType()) \
    .add("q", StringType()) \
    .add("T", StringType()) \
    .add("m", BooleanType())

# readStream is used to read the messsages from kafka
mssg = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "BTCUSDT-TRADES") \
    .option("startingOffsets", "latest") \
    .load()
# Read streaming data from 'kafka' topic
# Specifies which port to connect at Kafka
# 'Subscribes' to our kafka topic
# Doesn't read old history, only the latest data (Helps in managing storage)
# Since we receive data from kafka as bytes, we need to convert it to Json string so we can apply schema on it

# Applying the above defined schema on our messages
parsed_mssg = mssg.selectExpr("CAST(value AS STRING) as json_mssg_str") \
    .select(from_json(col("json_mssg_str"), schema).alias("data")) \
    .select("data.*") #Flattens all the features in the data column
```

```
# Once our data is structured based on the schema, we can apply some transformations on it
# wihtColumn() is used to create a new columns in spark dataframe, we specify the new column name and its values
mydata = parsed_mssg.withColumn("price", col("p").cast("double")) \
                    .withColumn("quantity", col("q").cast("double")) \
                    .withColumn("quote_quantity", (col("p").cast("double"))*(col("q").cast("double"))) \
                    .withColumn("market_MT", col("m").cast("int")) \
                    .withColumn("trade_time", (from_unixtime(col("T").cast("long") / 1000)).cast("timestamp")) \
                    .withColumn("dayofweek", dayofweek(col("trade_time"))) \
                    .withColumn("hour", hour(col("trade_time"))) \
                    .withColumn("minutes", minute(col("trade_time")))
# Transformations Applied:
#    --> Casted price, quantity into double data type
#    --> quote_quantiy is calculated using quantity or volume of the trade and the price of one BTC
#    --> Market_MT has boolean values True/False, we cast them as int 1/0 (for feeding it to our model)
#    --> trade_time is converted to a readable timestamp
#    --> dayofweek, hour and minutes are extracted from the trade_time


# Now our data is processed and ready to be stored!
# write_to_influx() function is called for on each micro-batch of data for storing
```

```
def write_to_influx(batch_df, epoch_id): # batch_df --> a batch of streaming data, epoch_id --> unique ID of the batch

    for row in batch_df.collect(): # Accessing each row inside our batch

        try:
            #Accessing each attributes inside the row and storing it as a variable
            price = row.price
            quantity = row.quantity
            quote_quantity = row.quote_quantity
            market_MT = row.market_MT
            trade_time = int(row.trade_time.timestamp()) #Converted to Unix time for InfluxDB
            dayofweek = row.dayofweek
            hour = row.hour
            minutes = row.minutes

            line_protocol = (
                f"binance_trades,symbol=BTCUSDT " #measurement, Symbol
                f"dayofweek={dayofweek},hour={hour},minutes={minutes},price={price},quantity={quantity},quote_quantity={quote_quantity},market_MT={market_MT} {trade_time}")
                # line is a protocol that tells how the row is stored

            headers = {
                "Authorization": "Token supersecrettoken", # InfluxDB token
                "Content-Type": "text/plain"} # For Line protocol

            # Sending the data to InfluxDB(accessed using the URL)
            response = requests.post(
                url="http://influxdb2:8086/api/v2/write?org=myorg&bucket=home&precision=s",
                headers = headers,
                data = line_protocol
            )

            # Checks if influxDB stored the data, if it didn't then prints error
            if response.status_code >= 300:
                print(f"InfluxDB error [{response.status_code}]:", response.text)
            else:
                print("Succesfully wrote to InfluxDB")

        except Exception as e: #Catching any error
            print("Error writing to InfluxDB: ", str(e))

query = mydata.writeStream \
    .foreachBatch(write_to_influx) \
    .outputMode("append") \
    .start()
# writeStream() Specifies to spark to write the streaming data
# the write_to_influx() is called on each micro-batch of data (foreachBatch())
# Append - to add more date (not change previous data)
query.awaitTermination() # Terminates due to an error or when kafka stops stremaing data
```

The aim of this experiment was to stream, process, and store real-time trade data into a time-series database (InfluxDB) using Apache Spark Structured Streaming. This builds on top of the earlier experiment by adding real-time transformation and long-term storage. The experiment imposes concepts like enforcing schema, stream processing, time-based feature engineering, and interaction with external APIs.

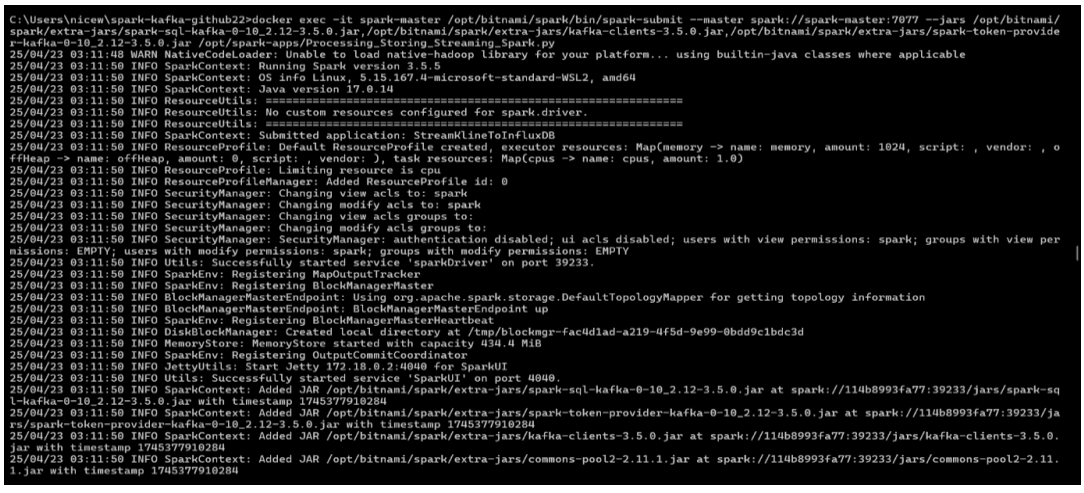
Explaining the process:

- We used SparkSession to create a real-time streaming application that consumes messages from a Kafka topic (BTCUSDT-TRADES). Messages are JSON and follow a predefined schema of price, quantity, trade timestamp, and a market maker flag.
- Kafka is integrated and connected with Spark using required JAR dependencies located in extra-jars. We create our spark session using master () that helps connect to the spark-master at port 7077. It tells that our spark runs on a distributed spark cluster, which uses resources from the workers (a, b and c). We specify the schema of our data to structure it based on the features that needs to be stored.
- We use spark.readStream() that connects or litsens to kafka at port 9092, subscribes to the topic name "BTCUSDT-TRADES", and starts reading only the latest data. Once the data is structured based on the schema, we then apply some transformations on it. We rename and cast the datatype of certain features. Create new feature like quote_quantity, convert the trade_time to a readable time stamp, and extract features like dayofweek, hour and minutes from it.
- After our data is processed, we need to store it in influxDB. we store it in batches of data. trade_time is an essential feature that is needed for storing the data in influxDB. we specify the how each data message should be stored. We use requests() that sends the requests to store the message at influxdb2:8086. writeStream() used for appending the processed streaming data to influx.

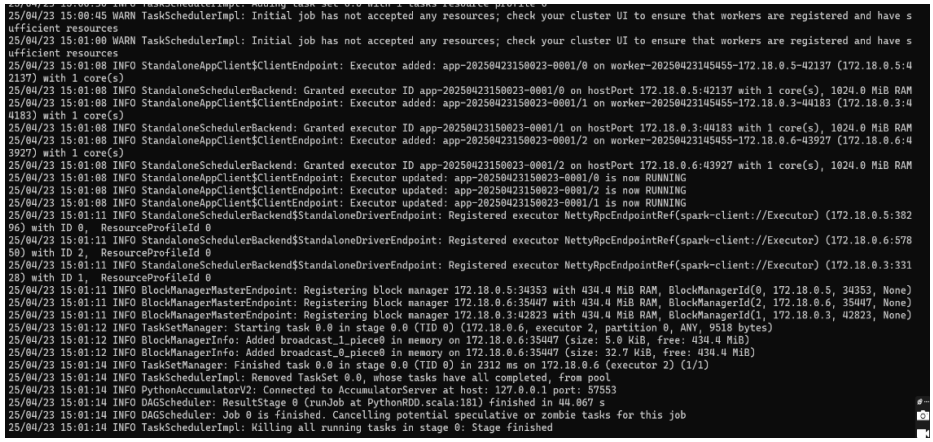
To run a spark script, we use the following command (We store all spark scripts in spark-apps folder):

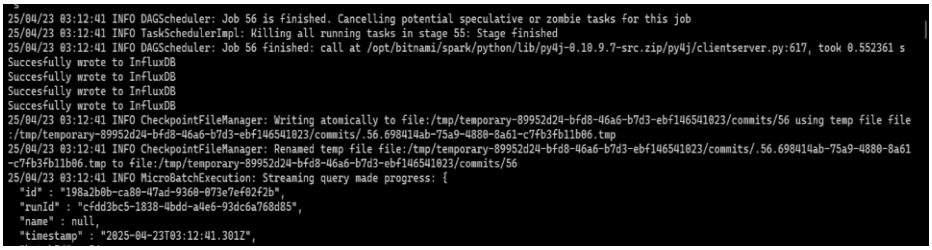
```
docker exec -it spark-master /opt/bitnami/spark/bin/spark-submit --master spark://spark-master:7077 --jars /opt/bitnami/spark/extra-jars/spark-sql-kafka-0-10_2.12-3.5.0.jar,/opt/bitnami/spark/extra-jars/kafka-clients-3.5.0.jar,/opt/bitnami/spark/extra-jars/spark-token-provider-kafka-0-10_2.12-3.5.0.jar /opt/spark-apps/Processing_Storing_Streaming_Spark.py
```

In the above commands, we enter into the spark-master container, we specify the jars (these are extra jars needed to connect to Kafka) and the location of our python script, we submit the spark job to the cluster at spark-master. Keep in mind, that the kafka script to stream data should be running on one cmd terminal. The above script is run on a separate terminal.



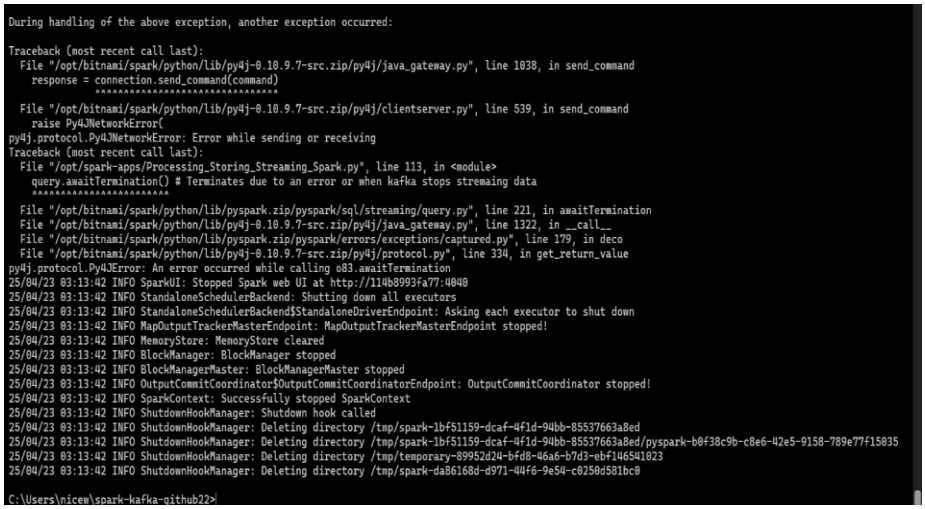
From the screenshot, we can understand that our spark session was created successfully, and the respective JAR files were also added.





Spark's worker nodes are running and have distributed the tasks, and are completing it. Spark is successfully writing the data to InfluxDB

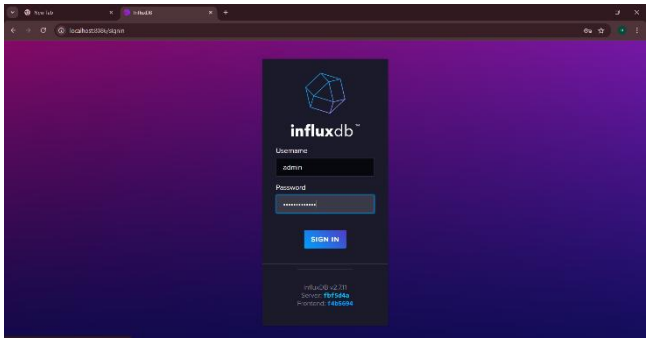
Spark logs data for each batch when using foreachBatch() function, it logs information such as how many rows were ingested using kafka, how many rows to append to influx based on the starOffset and latestOffset.



We can enter CTRL + C, to stop our spark job.

To ensure that your data is stored in influx:

- Enter localhost:8086 on the web-browser.
- Enter your username (admin) and password (adminpassword)
- Go to buckets, and then go to home



As we can see, our data is stored in influxdb as time series data, we can easily look at our visualizations of data stored in influxdb

Results:

- Spark successfully connects and litsens to kafka
- Spark successfully adds the JAR files
- Partitions added for parallelism
- Successfully wrote to InfluxDB
- When the kafka's websocket connection is closed, our Spark waits for the data (stays idle), we can then enter CTRL + C, to successfully stop the job.

Experiment 3: Storing the historical data in HDFS and performing Aggregations using MapReduce

- **Aim**:
  - Store the Historical trade data csv files on HDFS
  - Use MapReduce to extract the averages of each minute

- **Important Steps:**

  ➢ Open cmd, go to your project folder and enter namenode container using:

  ```
  docker-exec -it namenode bash
  ```

  ➢ Create an input folder in hdfs

  ```
  hdfs dfs -mkdir -p /user/root/input
  ```

- Copy our csv files to namenode:/apps

```
docker cp input namenode/apps
```

- Put all the csv files from namenode/apps/input to /user/root/input/ in hdfs

```
hdfs dfs -put input/*
```

- Copy the mapper and reducer in python's namenode

```
docker cp "C:\Users\nicew\spark-kafka-github22\python_mapreduce\mapper.py" namenode:/apps

docker cp "C:\Users\nicew\spark-kafka-github22\python_mapreduce\reducer.py" namenode:/apps
```

```
C:\Users\nicew\spark-kafka-github22>docker exec -it namenode bash
root@namenode:~# hdfs dfs -ls /user/root/input
Found 13 items
-rw-r--r--   3 root supergroup  253770715 2025-04-21 19:31 /user/root/input/BTCUSDT-trades-2025-04-01.csv
-rw-r--r--   3 root supergroup  407142245 2025-04-21 19:31 /user/root/input/BTCUSDT-trades-2025-04-02.csv
-rw-r--r--   3 root supergroup  313659581 2025-04-21 19:32 /user/root/input/BTCUSDT-trades-2025-04-03.csv
-rw-r--r--   3 root supergroup  380410582 2025-04-21 19:32 /user/root/input/BTCUSDT-trades-2025-04-04.csv
-rw-r--r--   3 root supergroup  120204475 2025-04-21 19:32 /user/root/input/BTCUSDT-trades-2025-04-05.csv
-rw-r--r--   3 root supergroup  301897511 2025-04-21 19:32 /user/root/input/BTCUSDT-trades-2025-04-06.csv
-rw-r--r--   3 root supergroup  742153884 2025-04-21 19:34 /user/root/input/BTCUSDT-trades-2025-04-07.csv
-rw-r--r--   3 root supergroup  419889979 2025-04-21 19:34 /user/root/input/BTCUSDT-trades-2025-04-08.csv
-rw-r--r--   3 root supergroup  590615905 2025-04-21 19:35 /user/root/input/BTCUSDT-trades-2025-04-09.csv
-rw-r--r--   3 root supergroup  345239116 2025-04-21 19:36 /user/root/input/BTCUSDT-trades-2025-04-10.csv
-rw-r--r--   3 root supergroup  336996640 2025-04-21 19:36 /user/root/input/BTCUSDT-trades-2025-04-11.csv
-rw-r--r--   3 root supergroup  190681600 2025-04-21 19:36 /user/root/input/BTCUSDT-trades-2025-04-12.csv
-rw-r--r--   3 root supergroup  249656476 2025-04-21 19:37 /user/root/input/BTCUSDT-trades-2025-04-13.csv
root@namenode:~#
```

Once our data is stored in HDFS, we can perform our mapreduce job on it.

```
mapred streaming  -files mapper.py,reducer.py  -input /user/root/input  -output /user/root/output1212  -
mapper "/usr/bin/python3 mapper.py"  -reducer "/usr/bin/python3 reducer.py"
```

In the above command, we are writing a mapreduce program using python. -files distributes our mapper and reducer to the Hadoop data nodes where the data is stored. We specify the shebang "/usr/bin/python3" to tell Hadoop that we are using python.

mapper.py - C:\Users\nicew\spark-kafka-github22\python_mapreduce\mapper.py (3.13.3)

File Edit Format Run Options Window Help

```python
#!/usr/bin/python3
"""mapper.py"""

# The first line is a shebang line, it tells our system to run our mapper using python3
import sys # Read input from stdin
from datetime import datetime # Timestamps to readable date-time formats

for line in sys.stdin: # Accessing each line from standard input
    rows = line.strip().split(',') #Stripping the line of any trailing spaces and then splitting it
    if len(rows) == 7: # Making sure that i hav all the 7 features

        try:  #Trying to access each attribute using index, and then casting to suitable data type
            price = float(rows[1])
            quantity = float(rows[2])
            quote_quantity = float(rows[3])
            trade_time = datetime.fromtimestamp(int(rows[4]) / 1_000_000) # Diving by 1_000_000,  cuz time is in Microseconds
            market_MT = 1 if rows[5].strip().lower() == "true" else 0 # Converting to 1/0
            day_hour = trade_time.strftime("%Y-%m-%d %H:%M") # Getting the date_hour_minute
            print(f"EMITTING: {day_hour}", file = sys.stderr) # Keeps track of what is happening when running the Job
            print(f"{day_hour}#{price}#{quantity}#{quote_quantity}#{market_MT}", flush = True) #
            # Sending the Data to reducer, where each attribute is seperated by '#'
            # flush = True (Makes sure that the output is immediately written)

        except ValueError: #In case of any error
            continue
```

The mapper.py script processes each line from the historical trades data from standard input, in structured CSV format with exactly seven attributes. It extracts key attributes such as price, quantity, quote quantity, trade time, market maker or taker. It converts the timestamp from microseconds to human-readable timestamp at the minute level. It also converts the market maker flag into binary form

(1 for true, 0 for false). It prints a key value pair foreach record, where my key is the converted timestamp and the values are the rest of the features. It used '#' as a delimiter to separate the features.

The goal of our reducer.py script is to aggregate our trade values by getting averages for each attribute for each minute, each hour and each Day. In total we have 13 csv files, where each file contains trade data for that day. (So, in one day we have 1440 minutes, for 13 days we will have 18, 720 minutes, which means my reducers output should contaim only 18,720 rows. The reducer takes as input the partitioned data emitted from mapper.

reducer.py - C:\Users\nicew\spark-kafka-github22\python_mapreduce\reducer.py (3.13.3)

File Edit Format Run Options Window Help

```
#!/usr/bin/python3
"""reducer.py"""

import sys

# In Reducer, our goal is to reduce our mapped values based on certain functions:
# curr (Date_Hour) is the key which our data is sorted on

#Initializing the variables
curr = None
sum_prices = 0
sum_quantity = 0
sum_quote_quantity = 0
No_of_trades = 0
Market_MT = []

#Accessing each row in standard input
for rows in sys.stdin:

    # Do you remember that our mapper prints the output like: {day_hour}#{price}#{quantity}#{quote_quantity}#{market_MT}
    # In order to extract values for each attribute, we split them on our delimitter '#' (We unpack our values)
    Date_Hour, price, quantity, quote_quantity, mm = rows.strip().split("#")

    # If our curr is already equal to day_hour, we can just add our the values for each attribute, to get the sum, and increase count by 1
    if curr == Date_Hour:
        sum_prices += float(price)
        sum_quantity += float(quantity)
        sum_quote_quantity += float(quote_quantity)
        Market_MT.append(int(mm))
        No_of_trades += 1

    else:
        if curr is not None:  # When the curr changes, we then calculate the averages for the previous curr and print it
            avg_price_per_day_hour = sum_prices/No_of_trades
            avg_quantity_per_day_hour = sum_quantity/No_of_trades
            avg_quote_quantity_per_day_hour = sum_quote_quantity/No_of_trades
            MM = sum(Market_MT) / len(Market_MT)
            Market_MT = []
            print(f"{curr}#{avg price per day hour}#{avg quantity per day hour}#{avg quote quantity per day hour}#{MM}")
        # If curr is None, we then assign our day_hour to it and the values to the attributes
        curr = Date_Hour
        sum_prices = float(price)
        sum_quantity = float(quantity)
        sum_quote_quantity = float(quote_quantity)
        No_of_trades = 1
        Market_MT.append(int(mm))

#Once all the rows from stdin are parsed through, we still need to calculate the averages and print it for the final value of curr
if curr is not None:
    avg_price_per_day_hour = sum_prices/No_of_trades
    avg_quantity_per_day_hour = sum_quantity/No_of_trades
    avg_quote_quantity_per_day_hour = sum_quote_quantity/No_of_trades
    MM = sum(Market_MT) / len(Market_MT)
    print(f"{curr}#{avg_price_per_day_hour}#{avg_quantity_per_day_hour}#{avg_quote_quantity_per_day_hour}#{MM}")
```

It reads each line of mapped input, splits it on the # delimiter to obtain price, quantity, quote quantity, and market maker flag. If the current key (curr) matches the emitted timestamp, it adds the values and increments the trade count. When the timestamp is incremented, it calculates the last timestamp's average price, quantity, quote quantity, and market maker ratio and prints the same. This ensures that there will be a single summarized line printed for every different minute. The script executes the last group when the loop terminates and uses try-except constructs implicitly by writing the data flow robustly.

```
root@namenode:~# mapred streaming   -files mapper.py,reducer.py   -input /user/root/input   -output /user/root/outputnew   -mapper "/usr/bin/python3 mapper.
py"   -reducer "/usr/bin/python3 reducer.py"
packageJobJar: [] [/opt/hadoop-3.3.1/share/hadoop/tools/lib/hadoop-streaming-3.3.1.jar] /tmp/streamjob6089359837916499801.jar tmpDir=null
2025-04-23 15:07:14,348 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at resourcemanager/172.18.0.14:8032
2025-04-23 15:07:14,532 INFO client.AHSProxy: Connecting to Application History server at historyserver/172.18.0.9:10200
2025-04-23 15:07:14,572 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at resourcemanager/172.18.0.14:8032
2025-04-23 15:07:14,573 INFO client.AHSProxy: Connecting to Application History server at historyserver/172.18.0.9:10200
2025-04-23 15:07:14,963 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1745420114369_0001
2025-04-23 15:07:15,502 INFO mapred.FileInputFormat: Total input files to process : 13
2025-04-23 15:07:15,513 INFO net.NetworkTopology: Adding a new node: /default-rack/172.18.0.13:9866
2025-04-23 15:07:15,514 INFO net.NetworkTopology: Adding a new node: /default-rack/172.18.0.12:9866
2025-04-23 15:07:15,514 INFO net.NetworkTopology: Adding a new node: /default-rack/172.18.0.8:9866
2025-04-23 15:07:15,612 INFO mapreduce.JobSubmitter: number of splits:40
2025-04-23 15:07:15,835 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1745420114369_0001
2025-04-23 15:07:15,835 INFO mapreduce.JobSubmitter: Executing with tokens: []
2025-04-23 15:07:16,081 INFO conf.Configuration: resource-types.xml not found
2025-04-23 15:07:16,082 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2025-04-23 15:07:17,033 INFO impl.YarnClientImpl: Submitted application application_1745420114369_0001
2025-04-23 15:07:17,075 INFO mapreduce.Job: The url to track the job: http://resourcemanager:8088/proxy/application_1745420114369_0001/
2025-04-23 15:07:17,076 INFO mapreduce.Job: Running job: job_1745420114369_0001
2025-04-23 15:07:25,217 INFO mapreduce.Job: Job job_1745420114369_0001 running in uber mode : false
2025-04-23 15:07:25,218 INFO mapreduce.Job:  map 0% reduce 0%
2025-04-23 15:07:43,591 INFO mapreduce.Job:  map 1% reduce 0%
2025-04-23 15:07:58,695 INFO mapreduce.Job:  map 2% reduce 0%
2025-04-23 15:08:15,018 INFO mapreduce.Job:  map 3% reduce 0%
2025-04-23 15:08:33,128 INFO mapreduce.Job:  map 4% reduce 0%
2025-04-23 15:08:49,367 INFO mapreduce.Job:  map 5% reduce 0%
2025-04-23 15:08:56,406 INFO mapreduce.Job:  map 6% reduce 0%
2025-04-23 15:08:57,513 INFO mapreduce.Job:  map 7% reduce 0%
```

Starting our map reduce job on three data nodes.

Number of splits = 40, means we have 40 mappers.

```
2025-04-23 15:23:00,863 INFO mapreduce.Job:  map 90% reduce 0%
2025-04-23 15:23:02,872 INFO mapreduce.Job:  map 91% reduce 0%
2025-04-23 15:23:08,971 INFO mapreduce.Job:  map 92% reduce 0%
2025-04-23 15:23:15,989 INFO mapreduce.Job:  map 93% reduce 0%
2025-04-23 15:23:16,992 INFO mapreduce.Job:  map 94% reduce 0%
2025-04-23 15:23:20,000 INFO mapreduce.Job:  map 95% reduce 0%
2025-04-23 15:23:22,006 INFO mapreduce.Job:  map 96% reduce 0%
2025-04-23 15:23:24,012 INFO mapreduce.Job:  map 97% reduce 0%
2025-04-23 15:23:25,014 INFO mapreduce.Job:  map 98% reduce 0%
2025-04-23 15:23:34,037 INFO mapreduce.Job:  map 100% reduce 0%
2025-04-23 15:23:38,004 INFO mapreduce.Job:  map 100% reduce 33%
2025-04-23 15:24:40,306 INFO mapreduce.Job:  map 100% reduce 57%
2025-04-23 15:24:46,333 INFO mapreduce.Job:  map 100% reduce 68%
2025-04-23 15:24:52,375 INFO mapreduce.Job:  map 100% reduce 69%
2025-04-23 15:24:57,395 INFO mapreduce.Job:  map 100% reduce 70%
2025-04-23 15:25:01,498 INFO mapreduce.Job:  map 100% reduce 71%
2025-04-23 15:25:07,528 INFO mapreduce.Job:  map 100% reduce 72%
2025-04-23 15:25:13,551 INFO mapreduce.Job:  map 100% reduce 73%
2025-04-23 15:25:19,573 INFO mapreduce.Job:  map 100% reduce 75%
2025-04-23 15:25:25,595 INFO mapreduce.Job:  map 100% reduce 76%
2025-04-23 15:25:29,522 INFO mapreduce.Job:  map 100% reduce 77%
2025-04-23 15:25:35,570 INFO mapreduce.Job:  map 100% reduce 78%
2025-04-23 15:25:41,642 INFO mapreduce.Job:  map 100% reduce 79%
2025-04-23 15:25:47,692 INFO mapreduce.Job:  map 100% reduce 81%
2025-04-23 15:25:53,744 INFO mapreduce.Job:  map 100% reduce 82%
2025-04-23 15:25:57,809 INFO mapreduce.Job:  map 100% reduce 83%
2025-04-23 15:26:03,837 INFO mapreduce.Job:  map 100% reduce 84%
2025-04-23 15:26:09,861 INFO mapreduce.Job:  map 100% reduce 86%
2025-04-23 15:26:15,886 INFO mapreduce.Job:  map 100% reduce 87%
2025-04-23 15:26:21,937 INFO mapreduce.Job:  map 100% reduce 88%
2025-04-23 15:26:25,914 INFO mapreduce.Job:  map 100% reduce 89%
2025-04-23 15:26:31,949 INFO mapreduce.Job:  map 100% reduce 91%
2025-04-23 15:26:37,976 INFO mapreduce.Job:  map 100% reduce 92%
2025-04-23 15:26:44,000 INFO mapreduce.Job:  map 100% reduce 93%
2025-04-23 15:26:50,023 INFO mapreduce.Job:  map 100% reduce 94%
2025-04-23 15:26:54,074 INFO mapreduce.Job:  map 100% reduce 96%
2025-04-23 15:27:00,100 INFO mapreduce.Job:  map 100% reduce 97%
2025-04-23 15:27:06,129 INFO mapreduce.Job:  map 100% reduce 98%
2025-04-23 15:27:12,152 INFO mapreduce.Job:  map 100% reduce 99%
2025-04-23 15:27:15,162 INFO mapreduce.Job:  map 100% reduce 100%
```

Our MapReduce took almost 20 minutes to complete.

```
        Total megabyte-milliseconds taken by all reduce tasks=1876672512
    Map-Reduce Framework
        Map input records=61021622
        Map output records=61021622
        Map output bytes=2790437535
        Map output materialized bytes=349877482
        Input split bytes=4720
        Combine input records=0
        Combine output records=0
        Reduce input groups=35741413
        Reduce shuffle bytes=349877482
        Reduce input records=61021622
        Reduce output records=18720
        Spilled Records=177533946
        Shuffled Maps =40
        Failed Shuffles=0
        Merged Map outputs=40
        GC time elapsed (ms)=5096
        CPU time spent (ms)=4051730
        Physical memory (bytes) snapshot=20785954816
        Virtual memory (bytes) snapshot=210045247488
        Total committed heap usage (bytes)=20601372672
        Peak Map Physical memory (bytes)=541274112
        Peak Map Virtual memory (bytes)=5060931584
        Peak Reduce Physical memory (bytes)=3487019008
        Peak Reduce Virtual memory (bytes)=8404529152
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=4651629301
    File Output Format Counters
```

Shows logs about how many input records mapped, how many input splits made, etc.

```
root@namenode:~#
root@namenode:~# hdfs dfs -cat /user/root/output22/part-00000 | wc -l
18720
root@namenode:~#
```

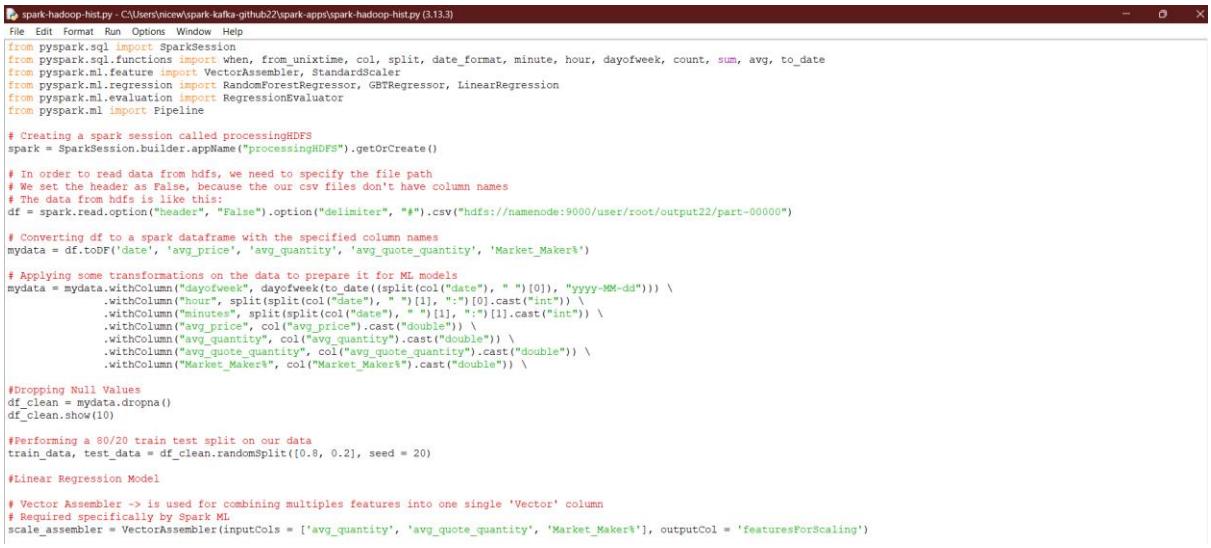
Our Reducer's Output contains 18729 rows.

Results:

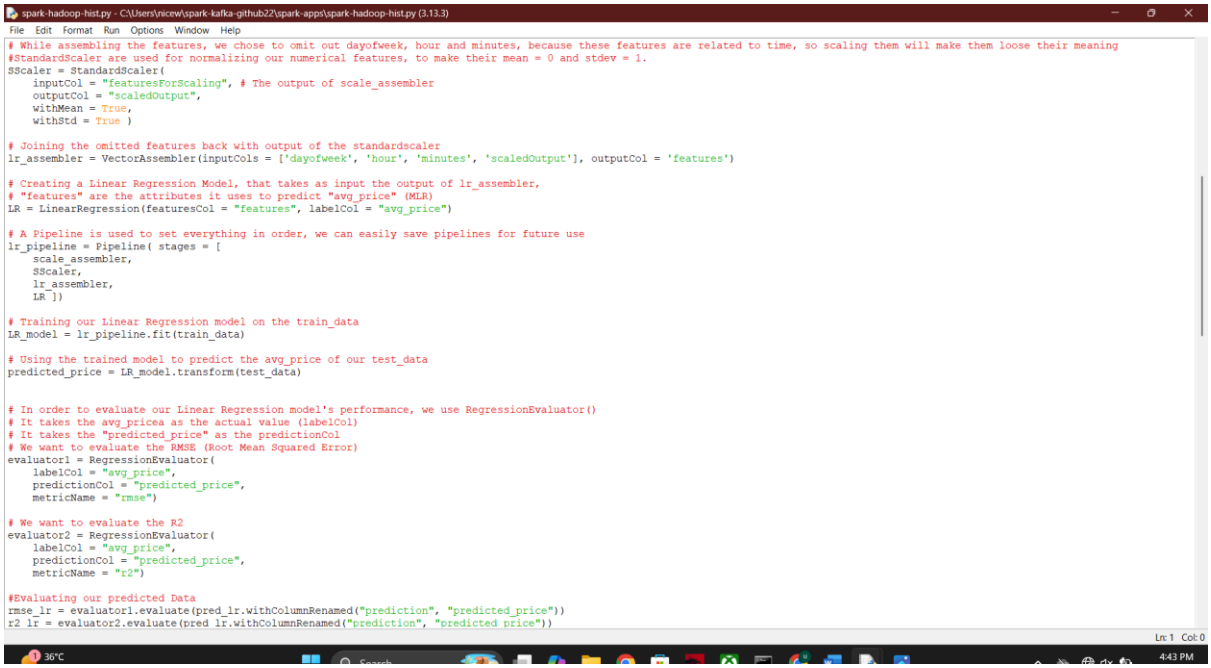
- Data is stored successfully in the 3 datanodes, all the information about the datanodes location is stored in the namenode.
- MapReduce is successfully performing its Job. It is distributing the mapper and reducer over the datanodes (Data Locality), which helps in saving time by removing the time spent to access the data.

Experiment 4: Extracting Aggregated Data from HDFS and Processing it using Hadoop for training our ML models using pyspark.ml .

- **Aim**: Create a pipeline for our Machine Learning Models and save them



```
spark-hadoop-hist.py - C:\Users\nicew\spark-kafka-github22\spark-apps\spark-hadoop-hist.py (3.13.3)
File  Edit  Format  Run  Options  Window  Help
from pyspark.sql import SparkSession
from pyspark.sql.functions import when, from_unixtime, col, split, date_format, minute, hour, dayofweek, count, sum, avg, to_date
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.regression import RandomForestRegressor, GBTRegressor, LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml import Pipeline

# Creating a spark session called processingHDFS
spark = SparkSession.builder.appName("processingHDFS").getOrCreate()

# In order to read data from hdfs, we need to specify the file path
# We set the header as False, because the our csv files don't have column names
# The data from hdfs is like this:
df = spark.read.option("header", "False").option("delimiter", "#").csv("hdfs://namenode:9000/user/root/output22/part-00000")

# Converting df to a spark dataframe with the specified column names
mydata = df.toDF('date', 'avg_price', 'avg_quantity', 'avg_quote_quantity', 'Market_Maker%')

# Applying some transformations on the data to prepare it for ML models
mydata = mydata.withColumn("dayofweek", dayofweek(to_date((split(col("date"), " ")[0]), "yyyy-MM-dd"))) \
               .withColumn("hour", split(split(col("date"), " ")[1], ":")[0].cast("int")) \
               .withColumn("minutes", split(split(col("date"), " ")[1], ":")[1].cast("int")) \
               .withColumn("avg_price", col("avg_price").cast("double")) \
               .withColumn("avg_quantity", col("avg_quantity").cast("double")) \
               .withColumn("avg_quote_quantity", col("avg_quote_quantity").cast("double")) \
               .withColumn("Market_Maker%", col("Market_Maker%").cast("double")) \

#Dropping Null Values
df_clean = mydata.dropna()
df_clean.show(10)

#Performing a 80/20 train test split on our data
train_data, test_data = df_clean.randomSplit([0.8, 0.2], seed = 20)

#Linear Regression Model

# Vector Assembler -> is used for combining multiples features into one single 'Vector' column
# Required specifically by Spark ML
scale_assembler = VectorAssembler(inputCols = ['avg_quantity', 'avg_quote_quantity', 'Market_Maker%'], outputCol = 'featuresForScaling')
```



```
spark-hadoop-hist.py - C:\Users\nicew\spark-kafka-github22\spark-apps\spark-hadoop-hist.py (3.13.3)
File  Edit  Format  Run  Options  Window  Help
# While assembling the features, we chose to omit out dayofweek, hour and minutes, because these features are related to time, so scaling them will make them loose their meaning
#StandardScaler are used for normalizing our numerical features, to make their mean = 0 and stdev = 1.
SScaler = StandardScaler(
    inputCol = "featuresForScaling", # The output of scale_assembler
    outputCol = "scaledOutput",
    withMean = True,
    withStd = True )

# Joining the omitted features back with output of the standardscaler
lr_assembler = VectorAssembler(inputCols = ['dayofweek', 'hour', 'minutes', 'scaledOutput'], outputCol = 'features')

# Creating a Linear Regression Model, that takes as input the output of lr_assembler,
# "features" are the attributes it uses to predict "avg_price" (MLR)
LR = LinearRegression(featuresCol = "features", labelCol = "avg_price")

# A Pipeline is used to set everything in order, we can easily save pipelines for future use
lr_pipeline = Pipeline( stages = [
    scale_assembler,
    SScaler,
    lr_assembler,
    LR ])

# Training our Linear Regression model on the train_data
LR_model = lr_pipeline.fit(train_data)

# Using the trained model to predict the avg_price of our test_data
predicted_price = LR_model.transform(test_data)


# In order to evaluate our Linear Regression model's performance, we use RegressionEvaluator()
# It takes the avg_pricea as the actual value (labelCol)
# It takes the "predicted_price" as the predictionCol
# We want to evaluate the RMSE (Root Mean Squared Error)
evaluator1 = RegressionEvaluator(
    labelCol = "avg_price",
    predictionCol = "predicted_price",
    metricName = "rmse")

# We want to evaluate the R2
evaluator2 = RegressionEvaluator(
    labelCol = "avg_price",
    predictionCol = "predicted_price",
    metricName = "r2")

#Evaluating our predicted Data
rmse_lr = evaluator1.evaluate(pred_lr.withColumnRenamed("prediction", "predicted_price"))
r2_lr = evaluator2.evaluate(pred_lr.withColumnRenamed("prediction", "predicted_price"))
Ln: 1  Col: 0
36°C
4:43 PM
```

```
#Saving our Linear Regression model as pred_lr_price_model22 in HDFS
LR_model.write().overwrite().save("hdfs://namenode:9000/user/hadoop/root/models/pred_lr_price_model22")
print("Model Saved!!")

#Random Forest

# In Random Forest, there is no need for scaling data, so we only have one assembler
assembler = VectorAssembler(inputCols = ["dayofweek", "hour", "minutes", "avg_quantity", "avg_quote_quantity", "Market_Maker%"], outputCol = "features")

# Defining our RandomForestRegressor Model, featuresCol are the "features" to train on, labelCol is the target to predict, predictionCol is the predicted value
# We have confifured our model to define:
#     --> numTrees = 150 (How many decision trees should your model have)
#     --> maxBins = 64 (How many groups? can be of the data)
#     --> maxDepth = 7 (How deep can each tree go)
rf = RandomForestRegressor(
    featuresCol = "features",
    labelCol = "avg_price",
    predictionCol = "predicted_price",
    numTrees = 150, maxBins = 128, maxDepth = 7)

# pl_rf Pipeline for our Random Forest Model
pl_rf = Pipeline( stages = [assembler, rf] )

#Training the model on train_data
model = pl_rf.fit(train_data)

#Testing it on test_data
pred = model.transform(test_data)

# Evaluating our model on the predicted price
rmse_rf = evaluator1.evaluate(pred)
r2_rf = evaluator2.evaluate(pred)

#Saving our Random Forest Model
model.write().overwrite().save("hdfs://namenode:9000/user/hadoop/root/models/pred_rf_price_model22")
print("Model Saved!!")

#GBT Regressor

# Creating our GBTRegressor model (configurations same as random_forest)
gbt = GBTRegressor(labelCol = "avg_price", featuresCol = "features", predictionCol = "predicted_price", maxIter = 150, maxBins = 128, maxDepth = 7)

# pl_gbt Pipeline for our GBT Regressor Model
pl_gbt = Pipeline( stages = [assembler, gbt] )
```

```
# pl_gbt Pipeline for our GBT Regressor Model
pl_gbt = Pipeline( stages = [assembler, gbt] )

#Training the model on train_data
model_gbt = pl_gbt.fit(train_data)

#Testing it on test_data
pred_gbt = model_gbt.transform(test_data)

# Evaluating our model on the predicted price
rmse_gbt = evaluator1.evaluate(pred_gbt)
r2_gbt = evaluator2.evaluate(pred_gbt)

#Saving our GBTRegressor Model
model_gbt.write().overwrite().save("hdfs://namenode:9000/user/hadoop/root/models/pred_gbt_price_model22")
print("Model Saved!!")


#Printing the spark DF showing the predicted and actual prize for the GBTRegressor and RandomForesr model
pred.select("avg_price", "predicted_price").show(20)
print("RndomForest Model")
print("\n")
print(f"RMSE: {rmse_rf:2f}")
print("\n")
print(f"r2: {r2_rf:2f}")
print("\n")

pred_gbt.select("avg_price", "predicted_price").show(20)
print("GBT Model")
print("\n")
print(f"RMSE: {rmse_gbt:2f}")
print("\n")
print(f"r2: {r2_gbt:2f}")
print("\n")

print("Linear Regression Model")
print("\n")
print(f"RMSE: {rmse_lr:2f}")
print("\n")
print(f"r2: {r2_lr:2f}")
print("\n")
```

The above pyspark script does the following tasks:

- Create a spark session, and reads data from the output file output1212 stored in the HDFS.
- Converts the data frame to a spark dataframe and apply transformations on it, like casting certain features, extracting time features.

- Once our data is all cleaned and processed, we then split it into 80% training and 20% testing. We specify the seed as 20, for reproducibility.
- We then start creating a pipeline for each ML model:

1. Linear Regression Model:
    - We Assemble the relevant features together using VectorAssembler, to get one output feature vector.
    - The output of the above assembler is fed to the Standard Scaler to scale and standardize our data.
    - Once the data is standardized, we once again use the VectorAssembler to compine the standardized features with the time features like dayofweek.
    - We then create the LinearRegression, which takes as input the assembled features, and takes the avg_price, as the labelCol (y) which is the column our features are trained to predict.
    - We create a Pipeline to set every step in each stage, so we can save our model.
    - After our pipeline is created we train our model on our train_data, and then test it on the test_data and evaluate its predicted_price based on RMSE and R2.
    - Finally, we save our model in HDFS, for predicting the streaming data's avg_prices.

2. Random Forest Regressor:
    - The pipeline is very similar like the above one, the only difference is that we don't need to scale our data, so need for standardscaler(), we just combine our relevant features and create a vector.
    - We then create a Random Forest regression model with number of trees as 150, number of bins as 128 and maxmimum depth is 7.
    - We then train our model on the train_data, and save the models pipeline and evaluate the performance.

3. Gradient Boosting Trees Regressor:
    - We create the exact same pipeline like the Random Forest Regressor, but we configure the model, to have 150 maximum iterations, 128 bins and 7 as the maximum depth of the model.

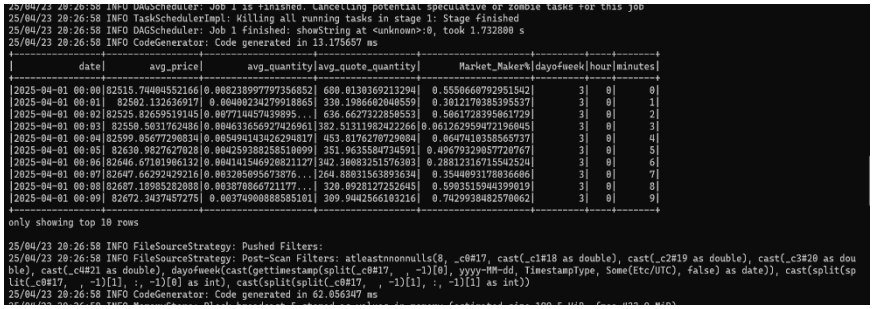
The above spark script is run on the terminal using the following command:

```
docker exec -it spark-master /opt/bitnami/spark/bin/spark-submit --master spark://spark-master:7077
/opt/spark-apps/spark-hadoop-hist.py
```

```
C:\Users\nicew\spark-kafka-github22>docker exec -it spark-master /opt/bitnami/spark/bin/spark-submit --master spark://spark-master:7077 /opt/spark-apps/spar
k-hadoop-hist.py
25/04/23 20:26:47 INFO SparkContext: Running Spark version 3.5.5
25/04/23 20:26:47 INFO SparkContext: OS info Linux, 5.15.167.4-microsoft-standard-WSL2, amd64
25/04/23 20:26:47 INFO SparkContext: Java version 17.0.14
25/04/23 20:26:47 INFO ResourceUtils: ==============================================================
25/04/23 20:26:47 INFO ResourceUtils: No custom resources configured for spark.driver.
25/04/23 20:26:47 INFO ResourceUtils: ==============================================================
25/04/23 20:26:47 INFO SparkContext: Submitted application: processingHDFS
25/04/23 20:26:47 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(memory -> name: memory, amount: 1024, script: , vendor: , o
ffHeap -> name: offHeap, amount: 0, script: , vendor: ), task resources: Map(cpus -> name: cpus, amount: 1.0)
25/04/23 20:26:47 INFO ResourceProfile: Limiting resource is cpu
25/04/23 20:26:47 INFO ResourceProfileManager: Added ResourceProfile id: 0
25/04/23 20:26:47 INFO SecurityManager: Changing view acls to: spark
25/04/23 20:26:47 INFO SecurityManager: Changing modify acls to: spark
25/04/23 20:26:47 INFO SecurityManager: Changing view acls groups to:
25/04/23 20:26:47 INFO SecurityManager: Changing modify acls groups to:
25/04/23 20:26:47 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: spark; groups with view per
missions: EMPTY; users with modify permissions: spark; groups with modify permissions: EMPTY
25/04/23 20:26:47 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
25/04/23 20:26:48 INFO Utils: Successfully started service 'sparkDriver' on port 41839.
25/04/23 20:26:48 INFO SparkEnv: Registering MapOutputTracker
25/04/23 20:26:48 INFO SparkEnv: Registering BlockManagerMaster
25/04/23 20:26:48 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper for getting topology information
25/04/23 20:26:48 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
25/04/23 20:26:48 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
25/04/23 20:26:48 INFO DiskBlockManager: Created local directory at /tmp/blockmgr-f469aaca-ad96-4a26-b5f4-ab4bd8d32057
25/04/23 20:26:48 INFO MemoryStore: MemoryStore started with capacity 434.4 MiB
25/04/23 20:26:48 INFO SparkEnv: Registering OutputCommitCoordinator
25/04/23 20:26:48 INFO JettyUtils: Start Jetty 172.18.0.2:4040 for SparkUI
25/04/23 20:26:48 INFO Utils: Successfully started service 'SparkUI' on port 4040.
```

Successfully submitted our Spark script

Displaying the processed data which is extracted from HDFS.



```
25/04/23 20:26:58 INFO TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished
25/04/23 20:26:58 INFO DAGScheduler: Job 1 finished: showString at <unknown>:0, took 1.732800 s
25/04/23 20:26:58 INFO CodeGenerator: Code generated in 13.175657 ms
+----------+---------+------------+--------------------+-------------------+--------+----+-------+
|      date|avg_price|avg_quantity|   avg_quote_quantity|       Market_Maker%|dayofweek|hour|minutes|
+----------+---------+------------+--------------------+-------------------+--------+----+-------+
|2025-04-01| 00:00|82515.744045552166|0.008238997797356852|  680.8130369213294|  0.5559660792951542|       3|   0|      0|
|2025-04-01| 00:01|  82502.132636917|  0.00480234279018865|  330.1986602040559|  0.3012170385395537|       3|   0|      1|
|2025-04-01| 00:02|82525.826595191451|0.007711457439895...|  636.6627322850553|  0.5061728395061729|       3|   0|      2|
|2025-04-01| 00:03|  82550.5031762486|0.004633656927426961|382.51311982422266|0.061262959472196045|       3|   0|      3|
|2025-04-01| 00:04|82599.856772908341|0.008549414342629481|7| 453.8176270729084|  0.0647410358565737|       3|   0|      4|
|2025-04-01| 00:05|  82630.9827627028|0.004259388258510099|  351.9635584734591|  0.4967932905772076|       3|   0|      5|
|2025-04-01| 00:06|82646.671019861321|0.004141546920821127|342.30083251576383|  0.2881231671554252|       3|   0|      6|
|2025-04-01| 00:07|82647.662292429216|0.003285895673876...|264.88031563893634|  0.3544893178836686|       3|   0|      7|
|2025-04-01| 00:08|82687.189852282088|0.003870866721177...|  320.0928127252645|  0.5903515944399019|       3|   0|      8|
|2025-04-01| 00:09|  82672.3437457275|0.003749008885851014|1| 309.9442566103216|  0.7429938482570062|       3|   0|      9|
+----------+---------+------------+--------------------+-------------------+--------+----+-------+
only showing top 10 rows

25/04/23 20:26:58 INFO FileSourceStrategy: Pushed Filters:
25/04/23 20:26:58 INFO FileSourceStrategy: Post-Scan Filters: atleastnnonnulls(8, _c0#17, cast(_c1#18 as double), cast(_c2#19 as double), cast(_c3#20 as double), cast(_c4#21 as double), dayofweek(cast(gettimestamp(split(_c0#17,  , -1)[0], yyyy-MM-dd, TimestampType, Some(Etc/UTC), false) as date)), cast(split(split(_c0#17,  , -1)[1], :, -1)[0] as int), cast(split(split(_c0#17,  , -1)[1], :, -1)[1] as int))
25/04/23 20:26:58 INFO CodeGenerator: Code generated in 62.056347 ms
```

```
25/04/23 20:27:28 INFO BlockManagerInfo: Removed broadcast_50_piece0 on 172.18.0.5:45015 in memory (size: 77.7 KiB, free: 422.5
25/04/23 20:27:28 INFO RandomForest: Internal timing for DecisionTree:
25/04/23 20:27:28 INFO RandomForest:   init: 0.002888789
  total: 9.303141634
  findBestSplits: 9.22551338
  chooseSplits: 9.1554204044
25/04/23 20:27:28 INFO MapPartitionsRDD: Removing RDD 98 from persistence list
25/04/23 20:27:28 INFO BlockManager: Removing RDD 98
25/04/23 20:27:28 INFO TorrentBroadcast: Destroying Broadcast(31) (from destroy at RandomForest.scala:305)
```

Logging the performance metrics for Decision Tree training:

- **init**: Time (s) to initialize the decision tree.
- **total**: Total time taken for building the Decision Tree.
- **findBestSplits**: Time spent for finding the best split at each node
- **chooseSplits**: Time taken to choose and apply the best split.

```
+------------------+------------------+
|         avg_price|   predicted_price|
+------------------+------------------+
|82525.82659519145|81245.04290849579|
|82622.40882903936|81593.67550351322|
|82453.97577535587|81733.37922717987|
|82454.76837651111|81031.54548123873|
|82660.90400932427|82276.70526102695|
|82677.05011557988|80984.62841645675|
|82684.857056524444|81464.12031384325|
| 82659.1242602941|81971.10497165819|
|82731.57566140206|81242.38251232091|
|82759.26724096917| 81318.7074939037|
|82581.37340375582|81035.19377517384|
|82574.43513844494|80981.62662590774|
|82661.18350221319|81473.89149092605|
| 82687.5655129828|81270.41550123865|
|82681.18214798618|81585.68066361618|
|82630.56391712712| 81457.1196835552|
|82566.48446685898| 80983.8745453395|
| 82582.3123794215|80970.11213487416|
|82686.60515915125|81196.72260006804|
|82727.98371044334|81288.61297325087|
+------------------+------------------+
only showing top 20 rows

RndomForest Model

RMSE: 2095.063696

r2: 0.386811
```

```
+------------------+------------------+
|         avg_price|   predicted_price|
+------------------+------------------+
|82525.82659519145|82971.37800195829|
|82622.40882903936|82816.07324190524|
|82453.97577535587|83080.01839891759|
|82454.76837651111|81999.73568999043|
|82660.90400932427|83078.80030829314|
|82677.05011557988|82955.11905770398|
|82684.857056524444| 80716.8941566132|
| 82659.1242602941|82856.27865294015|
|82731.57566140206|82875.35829741128|
|82759.26724096917|82569.40126293217|
|82581.37340375582|82053.67443258362|
|82574.43513844494|82183.80591166607|
|82661.18350221319|80147.32465457759|
| 82687.5655129828|82206.71903084702|
|82681.18214798618| 82798.8836126179|
|82630.56391712712|82363.82947882045|
|82566.48446685898|82654.60729368868|
| 82582.3123794215|82610.81297711865|
|82686.60515915125|81907.76994622164|
|82727.98371044334|82163.70835202289|
+------------------+------------------+
only showing top 20 rows

GBT Model

RMSE: 758.677565

r2: 0.919589

Linear Regression Model

RMSE: 1388.962132

r2: 0.730486
```

Displaying the Average and Predicted Values in a table.

Displaying the RMSE and R2 of all our Machine Learning Models

```
25/04/23 20:29:51 INFO GradientBoostedTrees: Internal timing for DecisionTree:
25/04/23 20:29:51 INFO GradientBoostedTrees:   building tree 92: 0.889732613
  building tree 105: 0.950113285
  building tree 126: 1.094529777
  building tree 50: 0.8074828441
  building tree 71: 0.850333901
  building tree 111: 0.968562103
  building tree 65: 0.8075464421
  building tree 59: 0.900161017
  building tree 4: 0.782625196
  building tree 86: 0.851312161
  building tree 108: 1.007690474
  building tree 53: 0.688812877
  building tree 68: 0.811668446
  buildMetadata: 1.004281097
  building tree 74: 1.404477103
  building tree 80: 0.8088466889
  building tree 89: 0.839633306
  building tree 32: 0.639866898
  building tree 47: 0.671336917
  building tree 41: 0.74690719
  building tree 35: 0.7130574521
  building tree 143: 1.030480763
  building tree 62: 0.8108477151
  init: 0.00319793
  building tree 29: 0.7164187621
  building tree 20: 0.7246299151
  building tree 137: 1.191474543
  building tree 56: 0.7511878011
  building tree 14: 1.002124871
  building tree 131: 1.188944258
  building tree 38: 0.7471215091
  building tree 146: 1.062307308
  building tree 119: 1.066763267
  building tree 23: 0.6826334651
  building tree 17: 1.143026104
  building tree 125: 1.115134487
```

Displays the time taken to build each tree for GBT Regressor.

Analysing our model's accuracy:

- The Random Forest model got 2095.063 as RMSE, which means that my predicted values are 2095 units way off my actual values on average. The R^2 is around 0.386811, means my model can explain 38% of the variation in the target variable (avg_price).
- The GBT Regressor model got 758.677 as RMSE, which means that my predicted values are 759 units way off my actual values on average. The R^2 is around 0.919589, means my model can explain 91.9% of the variation in the avg_price.
- The Linear Regression model got 1388.962132 as RMSE, which means that my predicted values are 1389 units way off my actual values on average. The R^2 is around 0.730486, means my model can explain 73% of the variation in the avg_price.

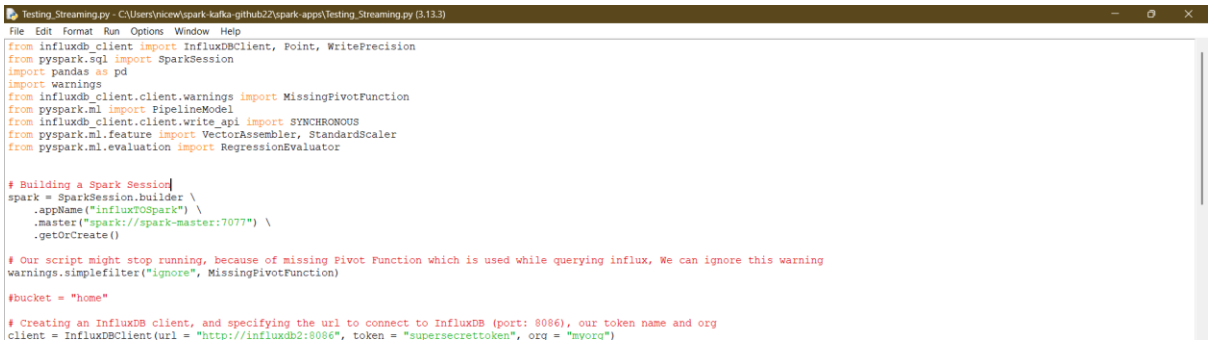
Comparing the results, we will consider the GBT Regressor model as our best model to predict the average price because it got the least error and was able to explain 91% of variation in avg_price.

Results:

- Spark has successfully loaded the output from Hadoop and processed it.
- We have utilized the ML models in pyspark.ml, and successfully trained and evaluated them on out historical data.
- We have saved our ML models for future testing.

Experiment 5: Extracting data from InfluxDB and predicting their avg_price

- **Aim**:
  - Load the stored data from influxdb using flux query.
  - Load our trained models from HDFS.
  - Process the data and test our ML models using this processed data.



```
Testing_Streaming.py - C:\Users\nicew\spark-kafka-github22\spark-apps\Testing_Streaming.py (3.13.3)
File  Edit  Format  Run  Options  Window  Help
from influxdb_client import InfluxDBClient, Point, WritePrecision
from pyspark.sql import SparkSession
import pandas as pd
import warnings
from influxdb_client.client.warnings import MissingPivotFunction
from pyspark.ml import PipelineModel
from influxdb_client.client.write_api import SYNCHRONOUS
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.evaluation import RegressionEvaluator


# Building a Spark Session
spark = SparkSession.builder \
    .appName("influxTOSpark") \
    .master("spark://spark-master:7077") \
    .getOrCreate()

# Our script might stop running, because of missing Pivot Function which is used while querying influx, We can ignore this warning
warnings.simplefilter("ignore", MissingPivotFunction)

#bucket = "home"

# Creating an InfluxDB client, and specifying the url to connect to InfluxDB (port: 8086), our token name and org
client = InfluxDBClient(url = "http://influxdb2:8086", token = "supersecrettoken", org = "myorg")
```

```
# Flux Query to extract data from InfluxDB
# The following query, extracts specific attributes from influx, performs some agg. on them like mean, and then store them in a data stream
# We need to perform 1-m interval aggregation (Getting the average's for each minute in each hour, in each day )
query = '''

import "math"
import "date"
priceStream = from(bucket: "home")
  |> range(start: -1d)
  |> filter(fn: (r) => r["_measurement"] == "binance_trades")
  |> filter(fn: (r) => r["_field"] == "price")
  |> aggregateWindow(every: 1m, fn: mean)
  |> rename(columns: {_value: "avg_price"})

quantityStream = from(bucket: "home")
  |> range(start: -1d)
  |> filter(fn: (r) => r["_measurement"] == "binance_trades")
  |> filter(fn: (r) => r["_field"] == "quantity")
  |> aggregateWindow(every: 1m, fn: mean)
    |> rename(columns: {_value: "avg_quantity"})


quoteQuantityStream = from(bucket: "home")
  |> range(start: -1d)
  |> filter(fn: (r) => r["_measurement"] == "binance_trades")
  |> filter(fn: (r) => r["_field"] == "quote_quantity")
  |> aggregateWindow(every: 1m, fn: mean)
    |> rename(columns: {_value: "avg_quote_quantity"})


marketMakerPct = from(bucket: "home")
  |> range(start: -1d)
  |> filter(fn: (r) => r["_measurement"] == "binance_trades")
  |> filter(fn: (r) => r["_field"] == "market_MT")
  |> aggregateWindow(every: 1m, fn: mean)
  |> rename(columns: {_value: "Market_Maker%"})

join1 = join(tables: {p: priceStream, q: quantityStream}, on: ["_time"])
join2 = join(tables: {pq: join1, qq: quoteQuantityStream}, on: ["_time"])
finalJoin = join(tables: {pqq: join2, mm: marketMakerPct}, on: ["_time"])


finalJoin
  |> map(fn: (r) => ({
    r with
    hour: date.hour(t: r._time),
    minutes: date.minute(t: r._time),
    dayofweek: date.weekDay(t: r._time)
  }))
  |> keep(columns: ["_time", "dayofweek", "hour", "minutes", "avg_quantity", "avg_quote_quantity", "Market_Maker%", "avg_price"])
  |> yield(name: "result_df")

'''
```

```
# --> Extract each attributes values, open an aggregation window of 1 minute, and then calculae the averages
# --> Rename and store them in their respective streams
# --> Join each data stream together one by one
# --> After the data streams are joined together, we can extract the dayofweek, hour, minutes from the _time and add it in the final resulted Join
# --> The output is result_df

df = client.query_api().query_data_frame(query)  # The query_data_frame() converts the result_df to a pandas dataframe
df = df.dropna() # Dropping any missing values
df = df.drop(columns = ["result", "table"]) # Dropping some extra columns resulted from the query
df = df[["_time", "dayofweek", "hour", "minutes", "avg_price", "avg_quantity", "avg_quote_quantity", "Market_Maker%"]] # Keeping only relevant features

# Creating a Spark DataFrame (df_spark) from  df (pandas dataframe)
df_spark = spark.createDataFrame(df)


#RandomForestRegressor Model
model = PipelineModel.load("hdfs://namenode:9000/user/hadoop/root/models/pred_rf_price_model22") # Loading the saved model from  /user/hadoop/root/models/pred_rf_price_model22 (stored in HI
pred = model.transform(df_spark) # Applying (testing) our model on df_spark

# Evaluating our results
evaluator1 = RegressionEvaluator(
    labelCol = "avg_price",
    predictionCol = "predicted_price",
    metricName = "rmse")

evaluator2 = RegressionEvaluator(
    labelCol = "avg_price",
    predictionCol = "predicted_price",
    metricName = "r2")

rmse_rf = evaluator1.evaluate(pred)
r2_rf = evaluator2.evaluate(pred)

# Writing the results back in Influx

write_api = client.write_api(write_options=SYNCHRONOUS)
for row in pred.select("avg_price", "predicted_price", "_time").collect():
    point = Point("price_predictions_rf") \
        .field("actual_price", row["avg_price"]) \
        .field("predicted_price", row["predicted_price"]) \
        .time(row["_time"], WritePrecision.NS)

    write_api.write(bucket = "home", org = "myorg", record = point)
```

```
#GBT Regressor

#Loading our GBT Regressor Model
model_gbt = PipelineModel.load("hdfs://namenode:9000/user/hadoop/root/models/pred_gbt_price_model22")
pred_gbt = model_gbt.transform(df_spark)

# Evaluating the model
rmse_gbt = evaluator1.evaluate(pred_gbt)
r2_gbt = evaluator2.evaluate(pred_gbt)

# Storing the results back in influx
for row in pred_gbt.select("avg_price", "predicted_price", "_time").collect():
    point = Point("price_predictions_gbt") \
        .field("actual_price", row["avg_price"]) \
        .field("predicted_price", row["predicted_price"]) \
        .time(row["_time"], WritePrecision.NS)

    write_api.write(bucket="home", org="myorg", record=point)

#Linear Regression

#Loading our Regression Model
model_lr = PipelineModel.load("hdfs://namenode:9000/user/hadoop/root/models/pred_lr_price_model22")
pred_lr = model_lr.transform(df_spark)

# Evaluating the model
rmse_lr = evaluator1.evaluate(pred_lr.withColumnRenamed("prediction", "predicted_price"))
r2_lr = evaluator2.evaluate(pred_lr.withColumnRenamed("prediction", "predicted_price"))

# Storing the results back in influx
for row in pred_lr.select("avg_price", "prediction", "_time").collect():
    point = Point("price_predictions_lr") \
        .field("actual_price", row["avg_price"]) \
        .field("predicted_price", row["prediction"]) \
        .time(row["_time"], WritePrecision.NS)

    write_api.write(bucket="home", org="myorg", record=point)

# Printing our results
print("RandomForest Model")
print("\n")
print(f"RMSE: {rmse_rf:2f}")
print("\n")
print(f"r2: {r2_rf:2f}")
print("\n")

print("GBT Model")
print("\n")
print(f"RMSE: {rmse_gbt:2f}")
print("\n")
print(f"r2: {r2_gbt:2f}")
print("\n")

print("Linear Regression Model")
print(f"RMSE: {rmse_lr:2f}")
print("\n")
print(f"r2: {r2_lr:2f}")
print("\n")
```

Explaing the process:

- Loading data from InfluxDB:

We create an influxDBClient to connect to influxdb at port 8086, to be able to query and retrieve the data. We run a flux query, that retrieves 1-minute averages of each feature for each hour and day. It calculates minute-level averages of price, quantity, quote quantity, and market maker % and combines them into an aggregated stream with time-based features like hour, minute, and day of the week.

- Processing the Data:

The results from the flux query is stored as a pandas dataframe. We clean the dataframe by dropping null values and removing irrelevant features. Once our pandas dataframe is cleaned, we convert it to a Sparks RDD.

- Testing our pre-trained models:

Pre-trained models (Random Forest, GBT, Linear Regression) are loaded from HDFS. Each model is executed on predicting avg_price based on new InfluxDB data. Predictions are evaluated using RMSE and R^2.

- Writing the result back into InfluxDB:

Predicted and actual prices of all three models are written back to InfluxDB as separate measurements to be visualized or analysed in the future.

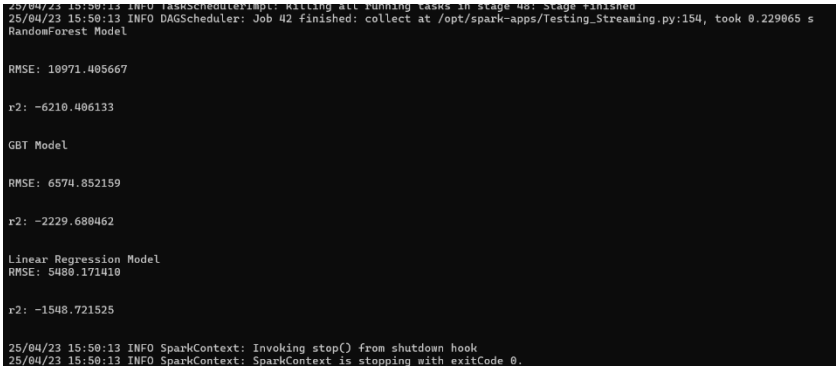
Submitting the above spark script:

```
docker exec -it spark-master /opt/bitnami/spark/bin/spark-submit --master spark://spark-master:7077
/opt/spark-apps/Testing_Streaming.py
```

Results:

- The connection to influxdb was successful and we retrieved data from it using flux query
- Spark was able to load the models successfully and test them on our processed real time data.

Interpreting the above results:



	Historical Data		Streaming Data	
Model/Evaluator	RMSE	R2	RMSE	R2
RandomForestClassifier	2102.508466	0.382445	10971.405667	-6210.406133
GBTRegressor	758.677565	0.919589	6574.852159	-229.680462
Linear Regression	1388.962132	0.730486	5480.171410	-1548.721525

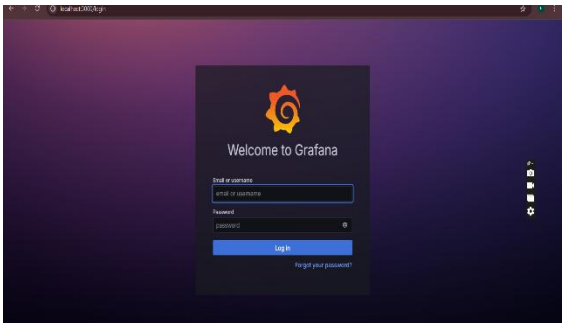
- The Random Forest model got 10, 971.405667 as RMSE, which means that my predicted values are 10, 971 units way off my actual values on average.
- The GBT Regressor model got 6574.852159 as RMSE, which means that my predicted values are 6575 units way off my actual values on average.
- The Linear Regression model got 5480.171410 as RMSE, which means that my predicted values are 5480 units way off my actual values on average.
- All of our models got negative R^2, which tells us that our models is doing a very bad job than just predicting the avg_price.

Comparing the results, we actually consider all models to be very poor, but we can say that the linear regression model gave the lowest error.

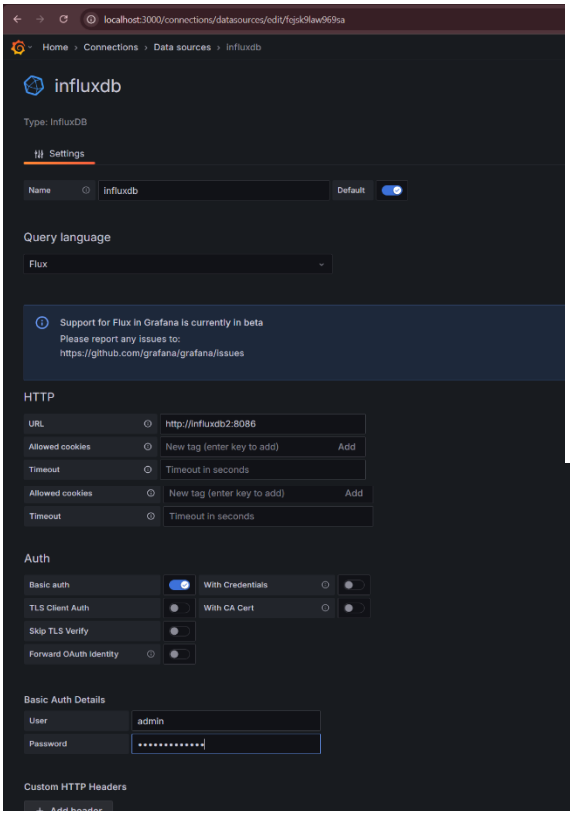
Experiment 6: Creating Dashboards Using Grafana

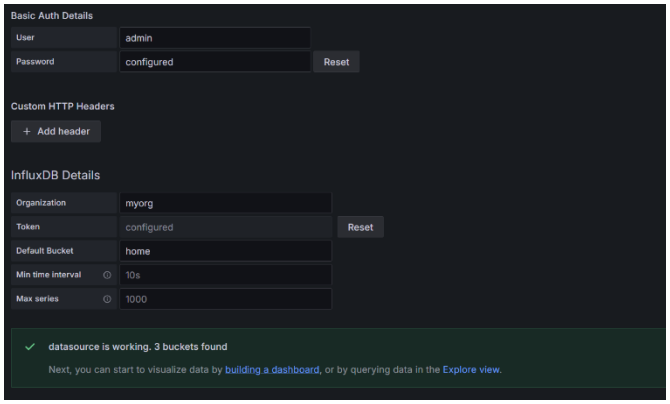
- **Aim**:
  - Connect to influxdb port on Grafana
  - Build charts and plots to plot our results

Explaining the process:



- Enter localhost:3000 on any web browser
- Enter username (admin) and password (admin)
- Click on Add your first data source
- Select InfluxDB
- Select Query language as flux
- Enter the HTTP url: http://influxdb2:8086
- Enter your influxdb's username and password
- Enter InfluxDB details:
  - Organization: myorg
  - Token: supersecrettoken
  - Default Bucket: home
- Press Save and test

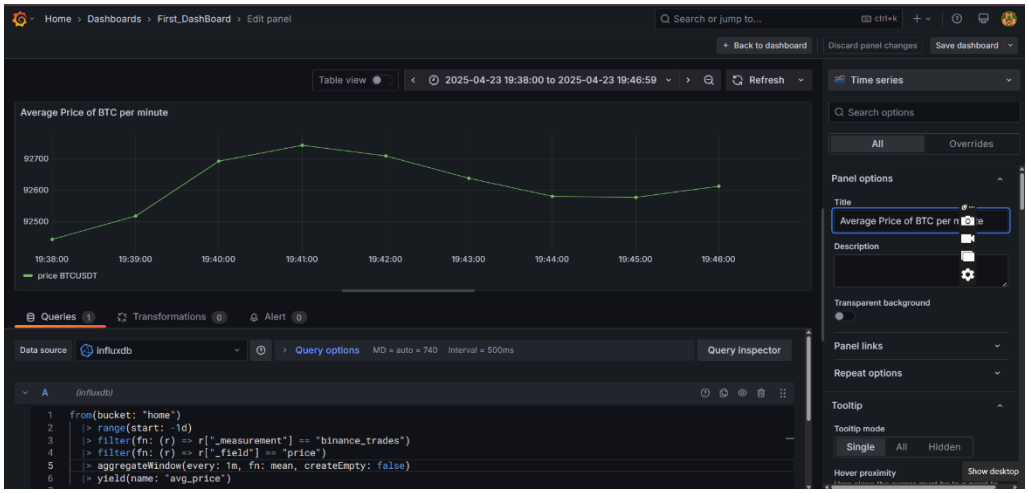




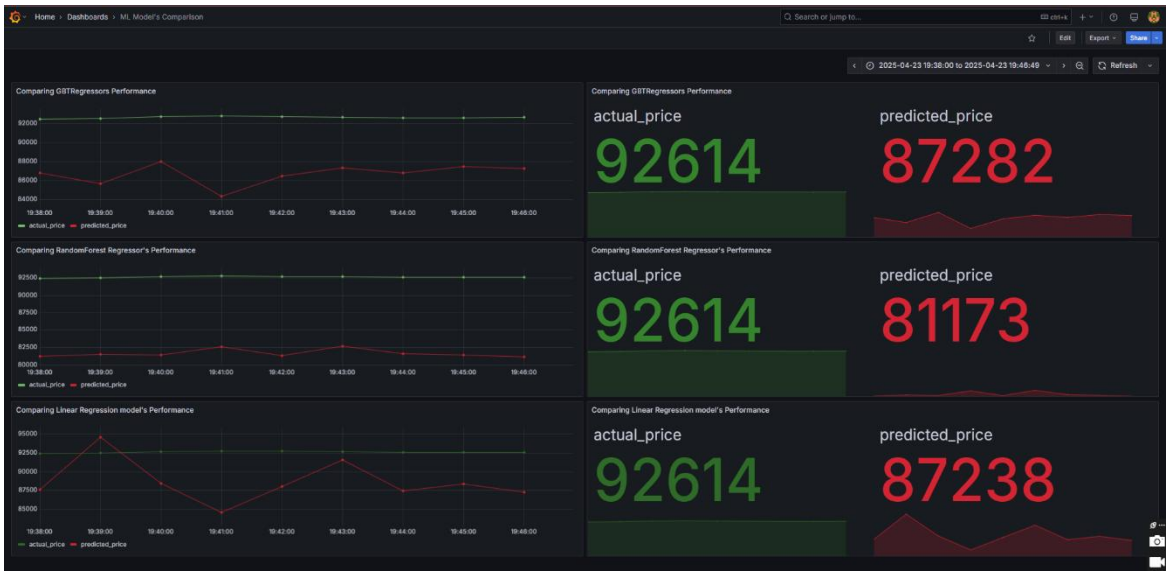
Now you can create dashboards using the data stored in influxdb, we use flux query to query the data from influx and create charts or time series on that data.

As we can see, we gave our influxdb a flux query to extract the average price per minute for 1 day data.

Here are the 2 dashboards we created.



Dashboard displaying the averages of our features of our streamed data



Dashboard displaying time series of our actual vs predicted price values for each model.

Conclusion and Discussion

Why did our model perform so poorly on real time data?

Here are some points to consider:

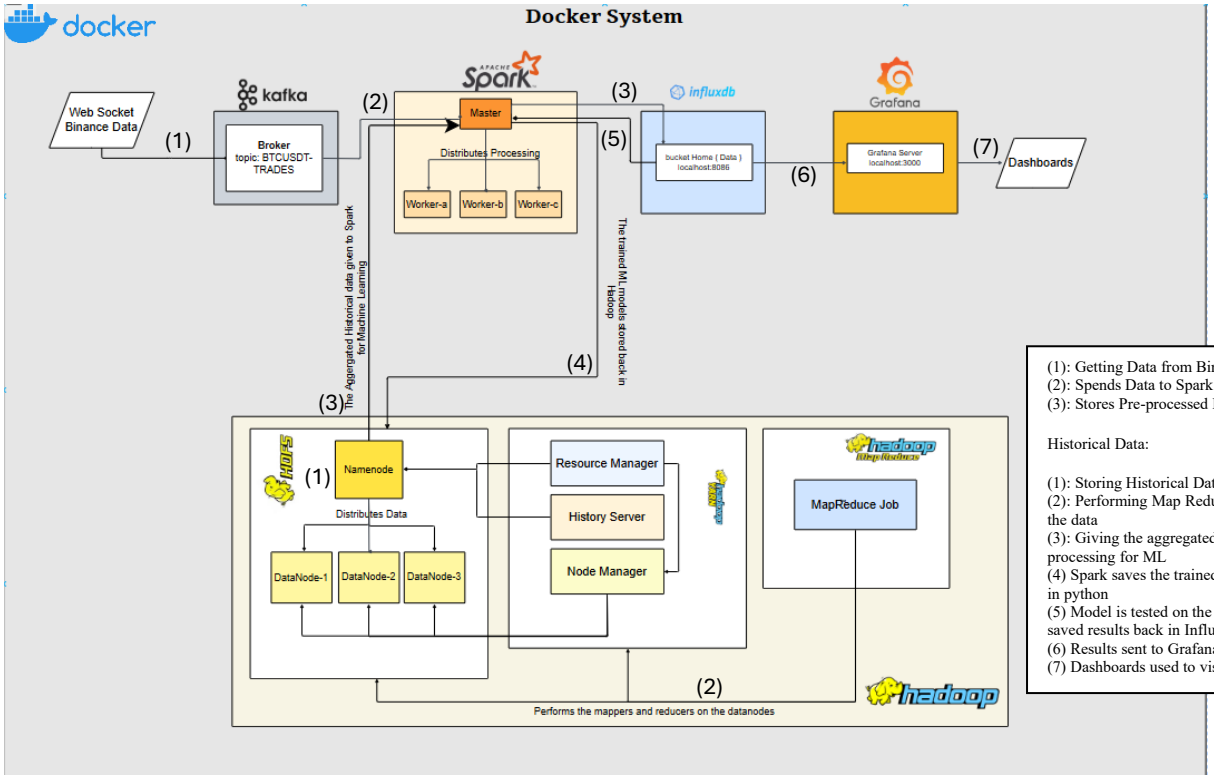
- The BTC rate is very sensitive and its value fluctuates based on the seasonal trends. In order to have a model that can accurately predict or forecast the avg_price, we need to train our model on let's say 1 year trades data. Due to the limits of our CPU and its resources, we cannot store such a huge data in our HDFS, and train our model on it. Which is why, we just trained our model on 13 days data (1st April to 13th April). Which means our model doesn't know about the rate patter for the end of the month, which could be a reason it is not performing well.
- Our model isn't generalizing well, and this could be due to the hyperparameter tuning of random forest and GBT regressor. We have set the bin size to 128, which can be considered the reason why our model performed pretty well on the training data, but failed to perform properly on the testing data.

Solutions:

- Improve your architecture, by adding more data nodes, spark-workers, Kafka brokers, replication factors, (to improve scalability and fault tolerance), so that our model can be ready to handle huge size of data like yearly binance data, which we can then train our model with.
- When tuning the parameters of the tree models, we can add regularizes like l1 or l2 that makes sure our model doesn't overfit.

Overall, our architecture seems to capture all the goals we had kept in mind. From using distributed processing, to distributed storage, to using structured query language for influx, we have incorporated many tools and setups together. Due to resource constraints, we weren't able to build an efficient and accurate Machine Learning model. But still we were able to build a full pipeline for handling streaming of real time data.

The overall architecture of our model:



References

https://github.com/hanashah-01/Docker-Hadoop-With-Python-Mapreduce

https://selectfrom.dev/executing-python-mapreduce-jobs-using-hadoop-in-a-docker-container-on-apple-silicon-mac-b59a5eda892d

https://issues.apache.org/jira/browse/HADOOP-13620

https://data.binance.vision/?prefix=data/spot/daily/trades/

https://online.visual-paradigm.com/diagrams/features/use-case-diagram-software/

https://www.lucidchart.com/