Streaming & Predictions of Financial Data using Kafka - Group 10

Members:
Gigliozzi, Nicholas
Kulik, Alexey
McCormick, Ernesto
Sarwar, Arsalan
Tasneem, Zarrin

Objective

The objective of our project was to use Apache Kafka as a central event storage system to send and receive live data from the Stock Market.

Approach

We divided our teams into 3 categories:

1. A team that created a python application that will continuously pull data from a financial system and publish to Apache Kafka[1].

2. A team that created an infrastructure to support this solution.
    a. Two customized installations (one primary and one backup) of Apache Kafka at team members' houses including direct connection as a producer and consumer as well as API interfaces.
    b. Create DNS entries to publish the server on the internet.

3. A team that created a machine learning model by processing messages from the primary Kafka server and creating insights & visualizations on price prediction.

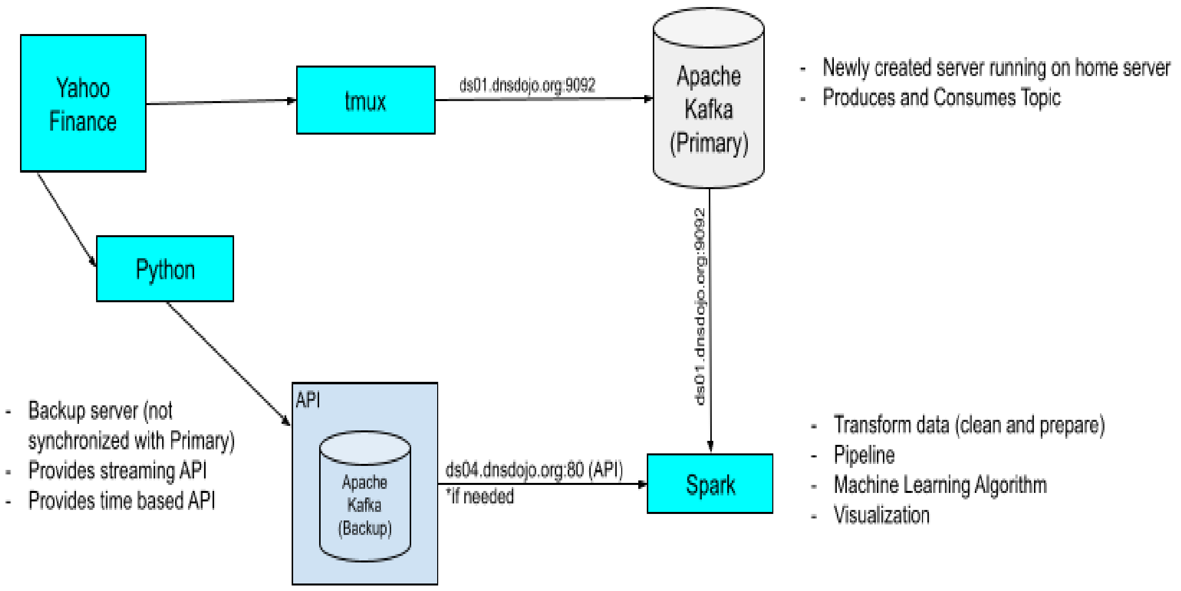

Yahoo Finance
tmux
ds01.dnsdojo.org:9092
Apache Kafka (Primary)
- Newly created server running on home server
- Produces and Consumes Topic
Python
ds01.dnsdojo.org:9092
- Backup server (not synchronized with Primary)
- Provides streaming API
- Provides time based API
API
Apache Kafka (Backup)
ds04.dnsdojo.org:80 (API)
*if needed
Spark
- Transform data (clean and prepare)
- Pipeline
- Machine Learning Algorithm
- Visualization

Figure 1: Conceptual architecture for this project

[1] https://kafka.apache.org/

Analysis

Based on team setup, each team was tasked to create a mini independent application which either provided data or consumed data. Having Kafka as the central management system allowed for this loose coupling of applications and allowed for team members to work on their time and schedule. Once the applications were built, there was a lot of coordination required to make the system work end to end with some rework as needed.

The journey each team took had a lot of learnings and lessons, which are outlined below.

1. Yahoo Finance Python Application

Due to stock market APIs getting more restricted in recent times we have elected to use Yliveticker[2] (Yahoo Finance[3] Live Ticker) API which pulls stock quotes including Volume and price. Yahoo Finance API pulls data from the Yahoo Finance every 5 seconds and does not have any restrictions on how many requests can be sent in a minute, which ended up being ideal for our project. We built a data set from these API requests that can "trace" and predict the most recent price fluctuations of a stock through Machine Learning.

The main difficulty of choosing an API is to find the API that fits our needs. Throughout our search we have come across a lot of stock market APIs that either could not pull live data, or had many restrictions on what kind, and how often requests can be sent. We found yliveticker API which could pull not only the price and volume quotes but also timestamps, market hours, daily volume, and price change from previous close. All of those data sets were instrumental in creating the most accurate prediction model. First 15 minutes of the stock market are the most volatile and generally unpredictable. While price volatility and volume drops off heavily by 12 PM, and it picks up close to market close from 3 pm to 4 pm, having exact information of what time frame stock was being traded in, adds value to our prediction model.

To make the loading script work in background we used tmux[4] that was already installed in our Ubuntu home servers, to be able to detach from the terminal without interrupting the program.

2. Kafka Installation

Two of our team members installed Apache Kafka servers on their home systems. One member installed it on his personal Ubuntu machine which acted as a primary server. Another member installed it on their Windows 2019 Server which acted as a backup server.

[2] https://pypi.org/project/yliveticker/
[3] https://ca.finance.yahoo.com/
[4] https://github.com/tmux/tmux/wik

Why two different Kafka Servers?

At the very beginning we found that although it is a common practice to bring data from Kafka to Spark, the documentation of Kafka, Spark and the intersection of both was scarce. In addition, we haven't had any experience using Kafka, hence, we expected a challenging setup process.

We decided to create one basic setup of Kafka to gather information as soon as possible and understand the process of streaming the information from Kafka to Spark. In the other server we learned about the details and complexities of configuring Kafka in a way that looks as close as possible as a real deployment, with proper security and authentication.

In order to make the setup form Kafka, we explored the alternative of using an instance of Kafka in Confluent[5] (a Kafka cloud service provider) and we successfully managed to set up one topic with schema, but we were discouraged by the 30 day limit of the trial. Also we required another server (or cloud function) to load the information into Kafka. Confluent didn't provide by itself any execution capability to load the data from the Yahoo API. That's why we end up using two home servers to run both the kafka servers and the python scripts to load the data to Kafka.

In the Appendix of this document it is possible to see the process of setting up the basic Kafka server.

Advanced configuration of the Kafka Server

The backup Kafka server (version 2.13-3.4.0, available https://kafka.apache.org/downloads) was installed on Windows 2019 Server. The server was installed with:

- Topic called ds04_messages_2 (_2 and previous versions became corrupted due to exploration of authentication for topic)
- A "DNS A" record/alias was created to resolve the IP address from ds04.dnsdojo.org to the server (using dyndns service)
- Clear text password authentication mechanism was configured by modifying the following files in the config folder

    - Server.properties
        - SASL_Plaintext security added
        - Listeners configured
        - Log directory configured
    - Zookeeper.properties
        - SASL_Plaintext security added
        - Log and Data directory configured
    - Producer.properties
        - SASL_Plaintext security added

[5] https://www.confluent.io

        - Listener configured
    - Consumer.properties
        - SASL_Plaintext security added
        - Listener configured

- To enable authentication on the Kafka server a new file called kafka_server_jaas.conf was created in the config directory with the username and password. This file was then added to the kafka server start-up script (found in bin\windows folder) as a PATH variable.
- All modified config files are provided as part of the project in a zip file where all the code is provided.
- There are other methods of authentication (e.g. certificate etc) which are more secure and should be used in production environments. Our intent was to show the possibility of authentication.

When all these configurations were applied, then the Kafka server was started using the following commands

Startup Zookeeper[6]:

```
E:\Apps\kafka_2.13-3.4.0>.\bin\windows\zookeeper-server-start.bat
.\config\zookeeper.properties
```

Startup Kakfa:

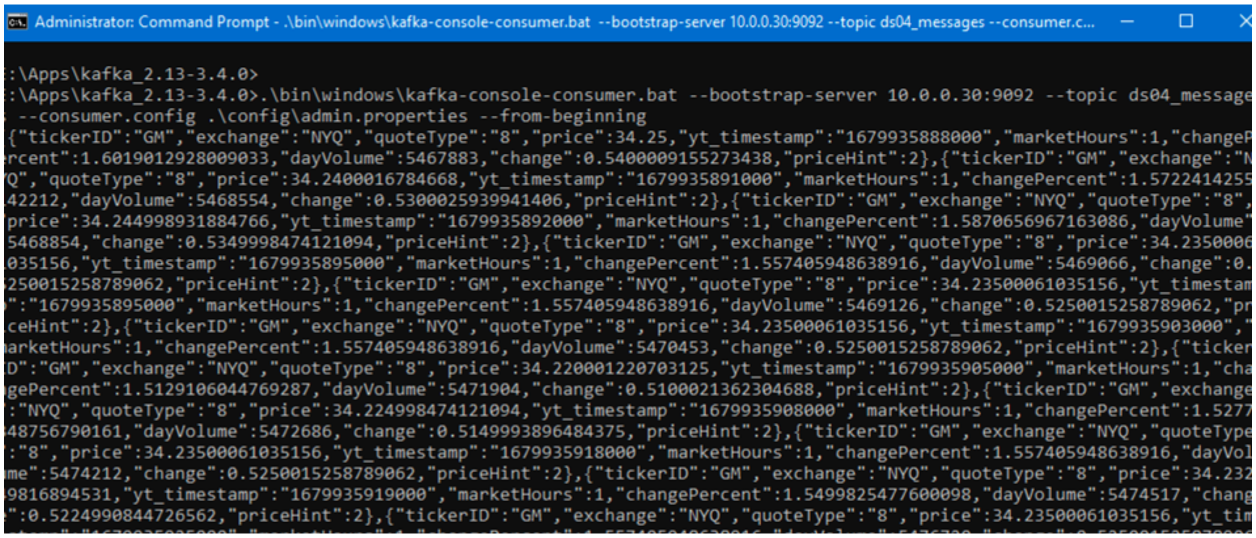
```
E:\Apps\kafka_2.13-3.4.0>.\bin\windows\kafka-server-start.bat .\config\server.properties
```

Since the server is configured with password authentication and kafka has a CLI, a special command file (admin.properties) has to be passed into every kafka command from the console that has username and password embedded into it. E.g.

To show all messages in a topic, we must pass admin.properties for proper authentication

```
E:\Apps\kafka_2.13-3.4.0>.\bin\windows\kafka-console-consumer.bat --bootstrap-server
10.0.0.30:9092 --topic ds04_messages --consumer.config .\config\admin.properties
--from-beginning
```

[6] https://zookeeper.apache.org/



```
Administrator: Command Prompt - .\bin\windows\kafka-console-consumer.bat  --bootstrap-server 10.0.0.30:9092 --topic ds04_messages --consumer.c...

:\Apps\kafka_2.13-3.4.0>
:\Apps\kafka_2.13-3.4.0>.\bin\windows\kafka-console-consumer.bat --bootstrap-server 10.0.0.30:9092 --topic ds04_message
 --consumer.config .\config\admin.properties --from-beginning
{"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.25,"yt_timestamp":"1679935888000","marketHours":1,"changeP
rcent":1.6019012928009033,"dayVolume":5467883,"change":0.5400009155273438,"priceHint":2},{"tickerID":"GM","exchange":"N
Q","quoteType":"8","price":34.2400016784668,"yt_timestamp":"1679935891000","marketHours":1,"changePercent":1.5722414255
42212,"dayVolume":5468554,"change":0.5300025939941406,"priceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType":"8",
price":34.244998931884766,"yt_timestamp":"1679935892000","marketHours":1,"changePercent":1.5870656967163086,"dayVolume"
5468854,"change":0.5349998474121094,"priceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.2350006
035156,"yt_timestamp":"1679935895000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5469066,"change":0.
25001525878906 2,"priceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestam
":"1679935895000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5469126,"change":0.5250015258789062,"pr
ceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestamp":"1679935903000","
arketHours":1,"changePercent":1.557405948638916,"dayVolume":5470453,"change":0.5250015258789062,"priceHint":2},{"ticker
D":"GM","exchange":"NYQ","quoteType":"8","price":34.220001220703125,"yt_timestamp":"1679935905000","marketHours":1,"cha
gePercent":1.5129106044769287,"dayVolume":5471904,"change":0.5100021362304688,"priceHint":2},{"tickerID":"GM","exchange
:"NYQ","quoteType":"8","price":34.224998474121094,"yt_timestamp":"1679935908000","marketHours":1,"changePercent":1.5277
48756790161,"dayVolume":5472686,"change":0.5149993896484375,"priceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType
:"8","price":34.23500061035156,"yt_timestamp":"1679935918000","marketHours":1,"changePercent":1.557405948638916,"dayVol
me":5474212,"change":0.5250015258789062,"priceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.232
9816894531,"yt_timestamp":"1679935919000","marketHours":1,"changePercent":1.5499825477600098,"dayVolume":5474517,"chang
":0.5224990844726562,"priceHint":2},{"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_tim
```

Figure 2: All messages stored in the topic ds04_messages_02 on backup server using authentication

One way to get data is directly from Kafka over port 9092. This was adopted for the primary Kakfa server. For the secondary, we decided to develop API's around Kafka to provide an alternative way to import/export data from Kafka. These API's were developed in .NET Core 7.0 API services (code included in the zip file). These API's were also deployed to the Windows 2019 server with their own DNS registered URL. Each API was secured by providing a static Bearer token. The token HAS to be sent in the header for authentication to work.

Two API's were developed

1. An API to read data from a data stream and push messages to Kafka (ds04-producer.dnsdojo.org/api/v1/Producer) and a SQL database (called DS04_LiveTracker) at the same time.

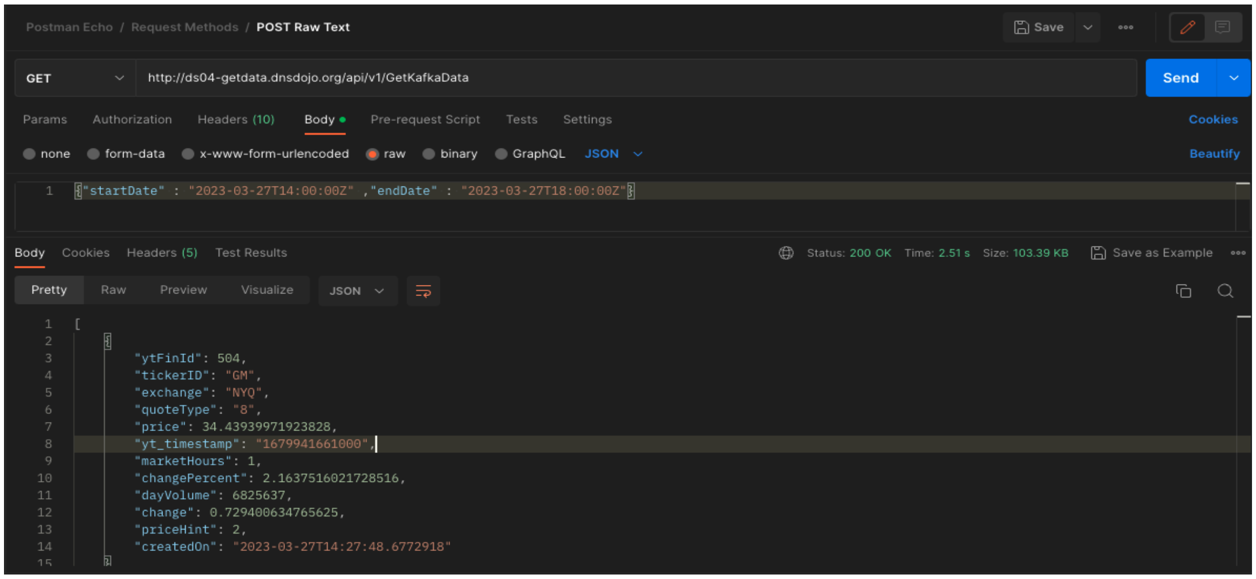
2. An API with two endpoints to retrieve data from SQL databases based on

    a. A time internal (http://ds04-getdata.dnsdojo.org/api/v1/GetKafkaData)
    b. Body of the API must include start and end time

       body = {"startDate": "2023-03-27T00:00:39.078Z","endDate": "2023-03-27T20:52:39.078Z"}

    c. Header MUST include the Bearer token for authentication

       my_headers = {"Content-Type" : "application/json", "Bearer" : "ea5afb228d2a4662bf80eb824ac43629"}


Postman Echo / Request Methods / POST Raw Text

GET http://ds04-getdata.dnsdojo.org/api/v1/GetKafkaData

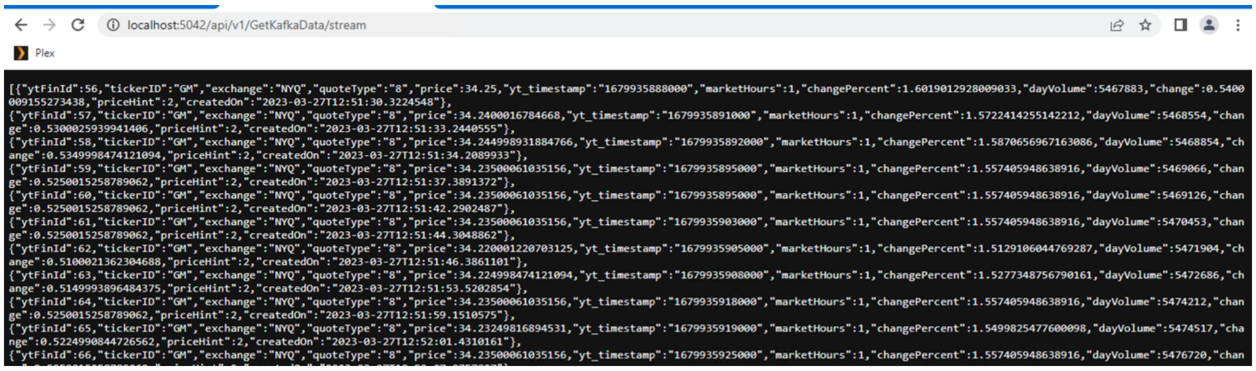
```
{"startDate" : "2023-03-27T14:00:00Z" ,"endDate" : "2023-03-27T18:00:00Z"}
```

Status: 200 OK Time: 2.51 s Size: 103.39 KB

```
[
    {
        "ytFinId": 504,
        "tickerID": "GM",
        "exchange": "NYQ",
        "quoteType": "8",
        "price": 34.43939971923828,
        "yt_timestamp": "1679941661000",
        "marketHours": 1,
        "changePercent": 2.1637516021728516,
        "dayVolume": 6825637,
        "change": 0.729400634765625,
        "priceHint": 2,
        "createdOn": "2023-03-27T14:27:48.6772918"
    }
```

Figure 3: Using API and authentication to get data from backup server using time range

d. Stream all available data (http://ds04-getdata.dnsdojo.org/api/v1/GetKafkaData/stream)


localhost:5042/api/v1/GetKafkaData/stream

```
[{"ytFinId":56,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.25,"yt_timestamp":"1679935888000","marketHours":1,"changePercent":1.6019012928009033,"dayVolume":5467883,"change":0.5400
009155273438,"priceHint":2,"createdOn":"2023-03-27T12:51:30.3224548"},
{"ytFinId":57,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.2400016784668,"yt_timestamp":"1679935891000","marketHours":1,"changePercent":1.5722414255142212,"dayVolume":5468554,"chan
ge":0.5300025939941406,"priceHint":2,"createdOn":"2023-03-27T12:51:33.2440555"},
{"ytFinId":58,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.244998931884766,"yt_timestamp":"1679935892000","marketHours":1,"changePercent":1.5870656967163086,"dayVolume":5468854,"ch
ange":0.5349998474121094,"priceHint":2,"createdOn":"2023-03-27T12:51:34.2089933"},
{"ytFinId":59,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestamp":"1679935895000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5469066,"chan
ge":0.5250015258789062,"priceHint":2,"createdOn":"2023-03-27T12:51:37.3891372"},
{"ytFinId":60,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestamp":"1679935895000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5469126,"chan
ge":0.5250015258789062,"priceHint":2,"createdOn":"2023-03-27T12:51:42.2902487"},
{"ytFinId":61,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestamp":"1679935903000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5470453,"chan
ge":0.5250015258789062,"priceHint":2,"createdOn":"2023-03-27T12:51:44.3048862"},
{"ytFinId":62,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.220001220703125,"yt_timestamp":"1679935905000","marketHours":1,"changePercent":1.5129106044769287,"dayVolume":5471904,"ch
ange":0.5100021362304688,"priceHint":2,"createdOn":"2023-03-27T12:51:46.3861101"},
{"ytFinId":63,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.224998474121094,"yt_timestamp":"1679935908000","marketHours":1,"changePercent":1.5277348756790161,"dayVolume":5472686,"ch
ange":0.5149993896484375,"priceHint":2,"createdOn":"2023-03-27T12:51:53.5202854"},
{"ytFinId":64,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestamp":"1679935918000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5474212,"chan
ge":0.5250015258789062,"priceHint":2,"createdOn":"2023-03-27T12:51:59.1510575"},
{"ytFinId":65,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23249816894531,"yt_timestamp":"1679935919000","marketHours":1,"changePercent":1.5499825477600098,"dayVolume":5474517,"cha
nge":0.5224990844726562,"priceHint":2,"createdOn":"2023-03-27T12:52:01.4310161"},
{"ytFinId":66,"tickerID":"GM","exchange":"NYQ","quoteType":"8","price":34.23500061035156,"yt_timestamp":"1679935925000","marketHours":1,"changePercent":1.557405948638916,"dayVolume":5476720,"chan
```

Figure 4: Using API to stream data from Kafka backup server

Please note that this API can be extremely slow over the internet. It does work over a certain time but due to some throttling by the ISP.

This authentication installation was extremely challenging. There is limited documentation available on how to configure and use Kafka server using password authentication. Most online searches result in paid Kafka as a service instead of local installation.

After searching **many** stack exchanges and trying various configurations, we were getting a lot of authentication errors. Eventually we ran across this document (https://docs.vmware.com/en/VMware-Smart-Assurance/10.1.5/deployment-scenarios.pdf) that allowed us to fix the errors by updating the config files and we were able to start the kafka server using authentication (Success!).

However, then connecting via code to kafka created continuous authentication errors in logs. Further stack exchange searches and a lot of trial and error, we still couldn't connect from python/spark code to the kafka server. Our consumer python code was below:

```
consumer = KafkaConsumer(
   'ds04_messages_2',
   bootstrap_servers='ds04.dnsdojo.org:9092',
   auto_offset_reset='earliest',
   sasl_mechanism = "PLAIN",
   sasl_plain_username = "admin",
   sasl_plain_password = "password",
   security_protocol = "SASL_PLAINTEXT"
)
```

Further research on the internet (15+ articles) we realized that we had to change some of the server setup to define SASL_PLAINTEXT as a mechanism and also protocol to match the python code. Once this configuration change was made we were able to connect our code successfully!

We have managed to keep our loading program and the kafka servers running uninterruptedly for one week and load 49,124 transactions (messages). Two concepts that we needed to learn were Data Serialization and Checkpoints to make the project work.

Serialization is the process of converting data into a format that can be easily transmitted and stored. In the context of streaming data from Kafka to Spark, serialization is important because it ensures that the data is transmitted efficiently and that it can be easily processed by Spark. The choice of serialization format can have a significant impact on the performance of the system, as some formats are more efficient than others.

Checkpoints are a mechanism that allows Spark to save the state of the streaming job at regular intervals. This means that if the job fails or is interrupted, it can be restarted from the last checkpoint, rather than from the beginning. Checkpoints also help to ensure that the output of the job is consistent and correct, even in the face of failures or interruptions.

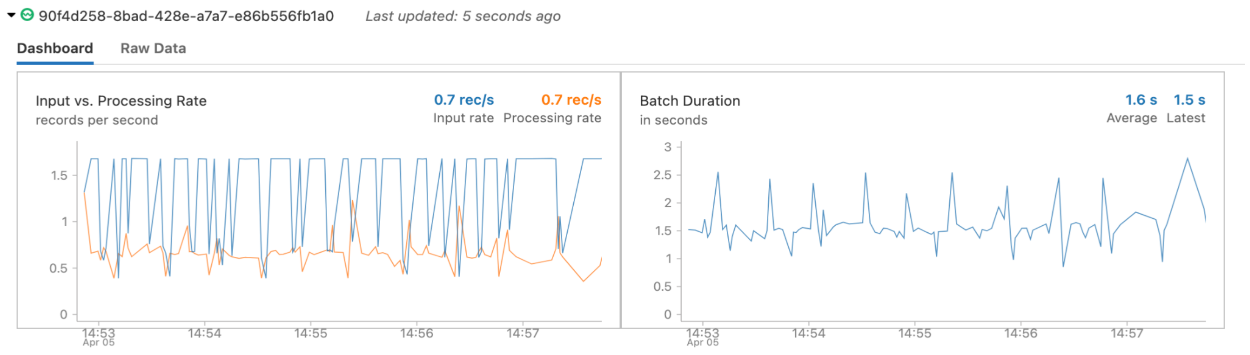


Figure 5: Snapshot of streaming metrics using Spark and Primary Kafka server

Streaming the data from Kafka to Spark

To stream data live from Kafka we choose to work with Databricks Community Edition, because of the familiarity gained in this course with this technology and the known capabilities of Spark to deal with data in motion, data transformation and ML that we thought was a good fit for this experiment.

To stream the data and write it in the Spark server we used pyspark.sql.functions for which the code can be found in the appendix.

We analyzed different alternatives of files formats for data persistence like:

- ORC (Optimized Row Columnar): ORC is a file format that is optimized for large-scale data processing. It is similar to Parquet in that it is a columnar format, but it has some additional features such as lightweight indexes and predicate pushdown that can further improve performance.
- Avro: A row-based file format that is designed for efficient data serialization and deserialization. It supports schema evolution, which makes it easy to evolve your data schema over time, and it has good support for complex data types.
- JSON: JSON is a text-based file format that is widely used for data interchange. While it is not as efficient as columnar formats like Parquet and ORC, it has the advantage of being human-readable and easy to work with.
- CSV: CSV (Comma Separated Values) is another text-based file format that is commonly used for data interchange. Like JSON, it is not as efficient as columnar formats, but it is widely supported and easy to work with.

We choose to use Parquet[7] because :

- Compression: Parquet supports various compression algorithms such as Snappy, Gzip, and LZO. By choosing an appropriate compression algorithm, you can significantly reduce the storage requirements and improve the performance of data processing. We were not sure about the limitations of our Databricks Community Edition environment and we thought that this feature would be convenient for our example and general Big Data enterprise applications.
- Schema evolution: Parquet supports schema evolution, which means that you can add or remove columns to the dataset without having to rewrite the entire dataset. This makes it easy to update your data schema over time as your data changes. We did use this feature in the first trials of this example.
- Predicate pushdown: Parquet supports predicate pushdown, which means that Spark can push down filter operations to the storage layer, allowing it to only read the data that is needed. This can significantly reduce the amount of data that needs to be read from storage, improving query performance.

[7] https://parquet.apache.org/

- Data types: Parquet supported all the datatypes required for this example.
- Scalability: Parquet is designed to scale to large datasets, and can efficiently store and process data in distributed environments. This makes it well-suited for use in Spark, which is designed to process large-scale data.

readStream and **writeStream**[8] Spark functions were key for building our real-time data pipeline example. This popular functions offer this benefits:

- Real-time processing: readStream and writeStream allow you to build real-time data processing pipelines that can continuously read and write data from and to streaming sources. This makes it possible to build applications that can process and respond to data in real-time, rather than relying on batch processing.
- Fault-tolerance: readStream and writeStream provide built-in support for fault-tolerance, which means that they can automatically recover from failures or interruptions in the data pipeline. This is important in streaming scenarios where data sources and processing nodes may fail or become temporarily unavailable.
- Ease of use: readStream and writeStream are designed to be easy to use and require minimal configuration. They provide a simple and consistent API that can be used to read and write data from and to various streaming sources, including Kafka, file systems, and socket streams.
- Scalability: readStream and writeStream are designed to be scalable and can handle large volumes of data from m
- Multiple sources in parallel. They provide built-in support for distributed processing, which means that you can easily scale your data pipelines to handle growing volumes of data.
- Integration with Spark ecosystem: readStream and writeStream integrate seamlessly with the broader Spark ecosystem, including libraries like Spark SQL, MLlib, and GraphX. This allows you to build complex data processing pipelines that can leverage the full power of Spark for analytics, machine learning, and graph processing.

We have managed to gather **126,913** messages from Yahoo Finance until April 13th 2023 in Kafka first and later to Spark.

3. Machine Learning and Visualizations

Machine Learning Models

[8] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Based on our learnings from previous courses, we would like to implement a few machine learning models, and then compare the accuracy of the different machine learning models. We have decided to use the linear regression, random forest model, GBT regressor, and decision tree as our models from the Pyspark library.

The code trains a random forest classifier on a dataset of stock market data, and then evaluates the accuracy of the trained model using the "accuracy" metric. The output of the code includes the accuracy of the model and the prediction results, which show the actual label values, predicted label values, and "change" values for each test data point.

The accuracy of the model represents the percentage of correctly predicted labels in the test data set. A higher accuracy score indicates that the model is better at predicting the labels. The prediction results show how well the trained model performed on the test data set. For each test data point, the label value is the actual value that the model is trying to predict, the prediction value is the value that the model predicted, and the "change" value is the feature value that was used to train the model. By comparing the label and prediction values, we can see where the model performed well and where it did not. We can also use this information to improve the model by adjusting the features or the model parameters.

The following steps were utilized to perform the random forest classification

1. Import necessary modules: window, first, col from pyspark.sql.functions, TimestampType from pyspark.sql.types, VectorAssembler from pyspark.ml.feature, RandomForestClassifier from pyspark.ml.classification, and MulticlassClassificationEvaluator from pyspark.ml.evaluation.
2. Define a window duration of 1 minute.
3. Read the data from a Parquet file into a DataFrame.
4. Cast the "timestamp" column to TimestampType.
5. Group the DataFrame by minute intervals based on the "timestamp" column.
6. Use agg to aggregate the "price", "marketHours", and "changePercent" columns, and use first to select the first value of each column. Rename the columns to "change", "marketHours", and "changePercent".
7. Cast the "change" column to integer type and use it as the label column.

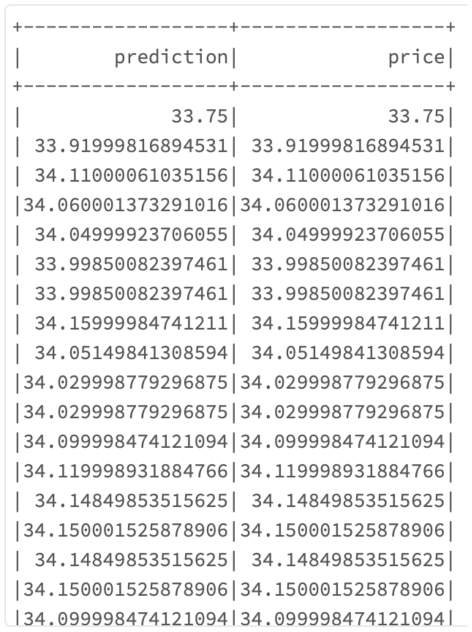
8. Create a VectorAssembler to combine the "change", "marketHours", and "changePercent" columns into a single "features" column.
9. Transform the DataFrame using the VectorAssembler.
10. Split the transformed DataFrame into training and test sets using the randomSplit method.
11. Train a random forest classifier on the training data.
12. Make predictions on the test data using the transform method of the trained model.
13. Evaluate the accuracy of the model using the MulticlassClassificationEvaluator and the "accuracy" metric.
14. Print the accuracy of the model.
15. Show the prediction results, including the label, prediction, and "change" columns, using the select method of the predictions DataFrame.

In this PySpark code, the window function is used to group the stock market data by minute intervals based on the timestamp column. The purpose of this function is to aggregate the data over time windows of one minute duration, allowing us to analyze the data at a higher level of granularity.

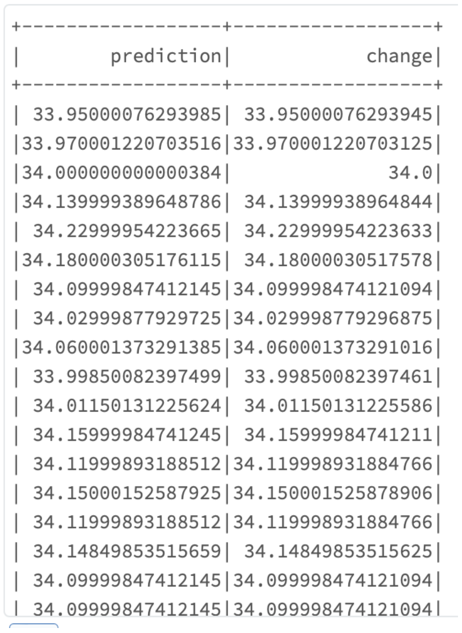
The window function is used to create time-based windows of data. It takes two arguments: the first argument is the column to be used as the timestamp for the window, and the second argument is the duration of the window. In this code, the window function groups the data by one-minute windows, so that all the data points within each minute are grouped together and aggregated.

Using window functions in this way can be useful for time series analysis, where we want to analyze data over time windows of a certain duration. It allows us to calculate summary statistics over specific time periods, which can be useful for identifying patterns, trends, and anomalies in the data.
A few graphs were created, such as line graph, and ROC curve to compare the different models and their prediction accuracies.

Our intention is not to make the best model, but instead the ability to create a template to ingest data from a live stream and make predictions.



```
+------------------+------------------+
|        prediction|             price|
+------------------+------------------+
|             33.75|             33.75|
| 33.91999816894531| 33.91999816894531|
| 34.11000061035156| 34.11000061035156|
|34.060001373291016|34.060001373291016|
| 34.04999923706055| 34.04999923706055|
| 33.99850082397461| 33.99850082397461|
| 33.99850082397461| 33.99850082397461|
| 34.15999984741211| 34.15999984741211|
| 34.05149841308594| 34.05149841308594|
|34.029998779296875|34.029998779296875|
|34.029998779296875|34.029998779296875|
|34.099998474121094|34.099998474121094|
|34.119998931884766|34.119998931884766|
| 34.14849853515625| 34.14849853515625|
|34.150001525878906|34.150001525878906|
| 34.14849853515625| 34.14849853515625|
|34.150001525878906|34.150001525878906|
|34.099998474121094|34.099998474121094|
```



```
+------------------+------------------+
|        prediction|            change|
+------------------+------------------+
| 33.95000076293985| 33.95000076293945|
|33.970001220703516|33.970001220703125|
|34.000000000000384|              34.0|
|34.139999389648786| 34.13999938964844|
| 34.22999954223665| 34.22999954223633|
|34.180000305176115| 34.18000030517578|
| 34.09999847412145|34.099998474121094|
| 34.02999877929725|34.029998779296875|
|34.060001373291385|34.060001373291016|
| 33.99850082397499| 33.99850082397461|
| 34.01150131225624| 34.01150131225586|
| 34.15999984741245| 34.15999984741211|
| 34.11999893188512|34.119998931884766|
| 34.15000152587925|34.150001525878906|
| 34.11999893188512|34.119998931884766|
| 34.14849853515659| 34.14849853515625|
| 34.09999847412145|34.099998474121094|
| 34.09999847412145|34.099998474121094|
```

Figure 6: Results and Changes of the Linear Regression for price

The accuracy of the linear regression is optimal with an RMSE value of: 0.3842973214859381.

The results above show that the data is utilized to train a regression model to predict the change in price during a window duration of 1 minute. The performance of the model is estimated using the root mean squared error (RMSE) and the predicted values are graphed against the actual values using matplotlib.

The printed output shows the model coefficients and intercept, as well as the calculated RMSE with a scatter plot of the predicted values against the actual values, which can be utilized to visually determine the accuracy of the model’s prediction.

The results of the model show how well a model can forecast the change in price based on the streamed data. A lower RMSE value, and a tighter clustering of the predicted values near the actual values in the plot indicates that the model makes accurate predictions.

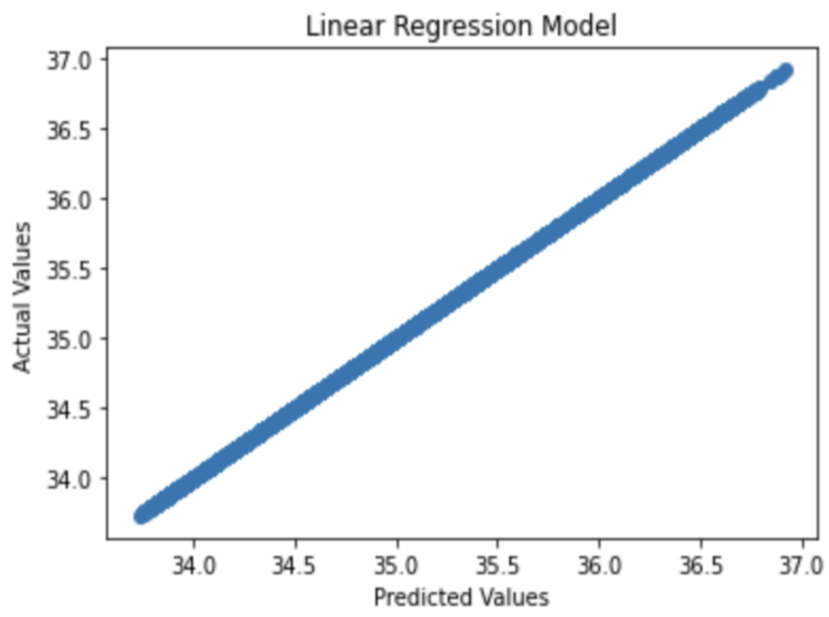
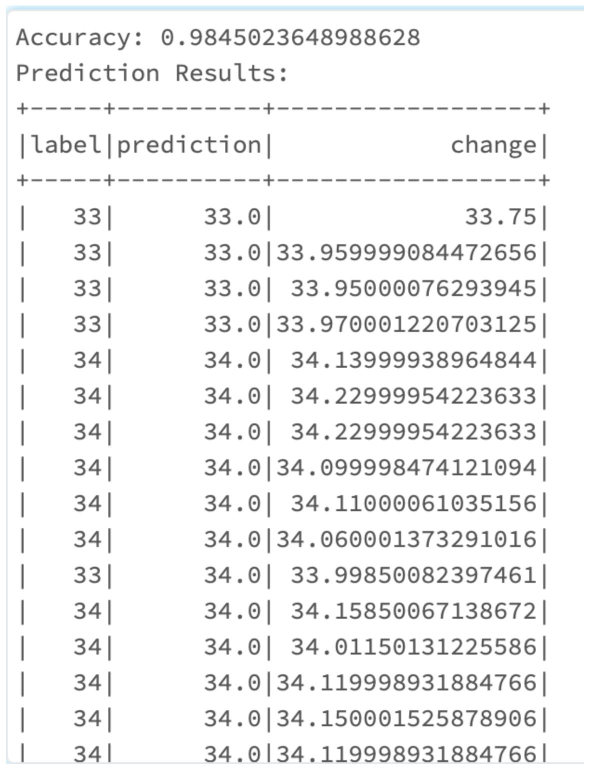
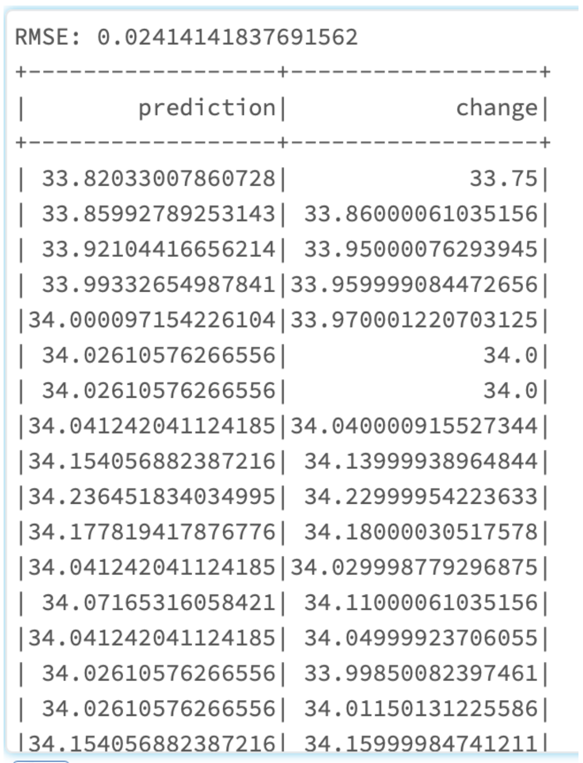


Figure 7 - Graph of the prediction results of the Linear Regression in terms of change



```
Accuracy: 0.9845023648988628
Prediction Results:
+-----+----------+------------------+
|label|prediction|            change|
+-----+----------+------------------+
|   33|      33.0|             33.75|
|   33|      33.0|33.959999084472656|
|   33|      33.0| 33.95000076293945|
|   33|      33.0|33.970001220703125|
|   34|      34.0| 34.13999938964844|
|   34|      34.0| 34.22999954223633|
|   34|      34.0| 34.22999954223633|
|   34|      34.0|34.099998474121094|
|   34|      34.0| 34.11000061035156|
|   34|      34.0|34.060001373291016|
|   33|      34.0| 33.99850082397461|
|   34|      34.0| 34.15850067138672|
|   34|      34.0| 34.01150131225586|
|   34|      34.0|34.119998931884766|
|   34|      34.0|34.150001525878906|
|   34|      34.0|34.119998931884766|
```



```
RMSE: 0.02414141837691562
+------------------+------------------+
|        prediction|            change|
+------------------+------------------+
| 33.82033007860728|             33.75|
| 33.85992789253143| 33.86000061035156|
| 33.92104416656214| 33.95000076293945|
| 33.99332654987841|33.959999084472656|
|34.000097154226104|33.970001220703125|
| 34.02610576266556|              34.0|
| 34.02610576266556|              34.0|
|34.041242041124185|34.040000915527344|
|34.154056882387216| 34.13999938964844|
|34.236451834034995| 34.22999954223633|
|34.177819417876776| 34.18000030517578|
|34.041242041124185|34.029998779296875|
| 34.07165316058421| 34.11000061035156|
|34.041242041124185| 34.04999923706055|
| 34.02610576266556| 33.99850082397461|
| 34.02610576266556| 34.01150131225586|
|34.154056882387216| 34.15999984741211|
```

Figure 8 - Prediction results of the Random Forest & Decision Tree Regressor in terms of change

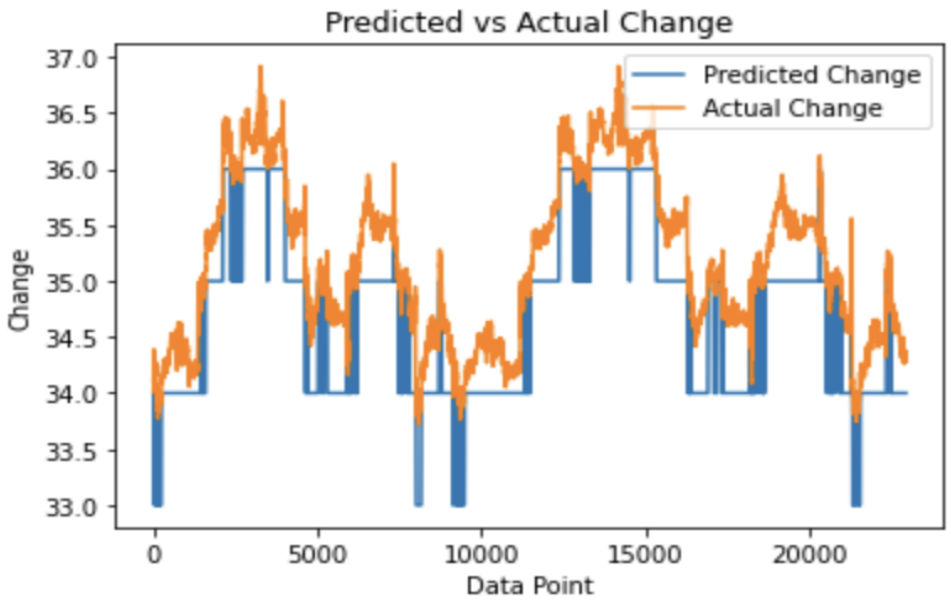


Figure 9 - line chart that compares the predicted and actual change values for a stock

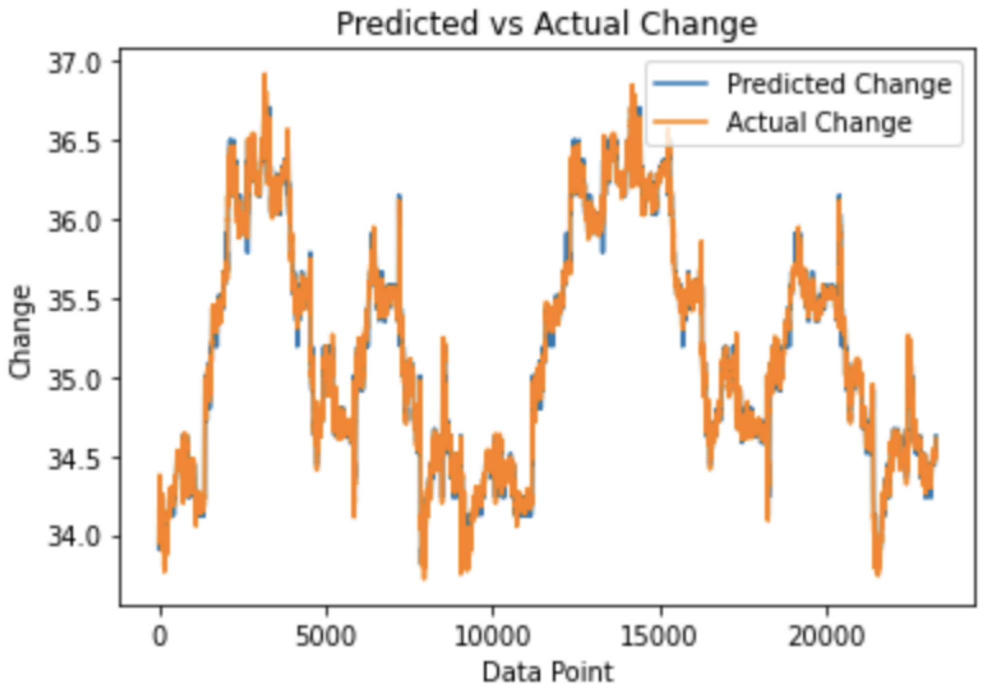


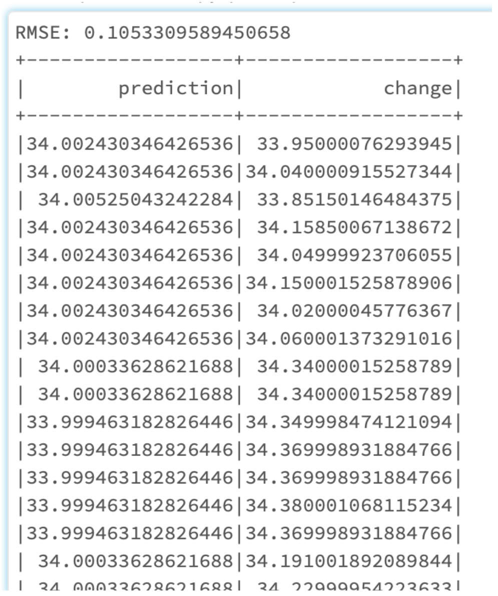
Figure 10 - line chart that compares the predicted and actual change values for a stock using Decision Tree Regressor

The second model, which was trained is a Random Forest Classifier model to forecast the change in stock prices based on the various features such as “change”, “marketHours”, and “changePercent”. The data is obtained from a parquet file, and grouped by minute intervals based on the timestamp column. The “change” column is cast to an integer type and utilized as the label column for the task of classification.

The accuracy of the trained model is determined by utilizing the MulticlassClassification Evaluator along with the “accuracy” metric. The accuracy of the model is printed to databricks. The predicted and actual change values are plotted by utilizing a line chart. The predicted change values are displayed in blue and the actual change values are portrayed in orange. The “Prediction Results” are also printed to databricks, which displays the actual change, the predicted change, and the label for each data point in the test set. The accuracy of the model is calculated as the ratio of the number of correct predictions to the total number of predictions made on the test data. The accuracy score shows how well the models performed when correctly predicting the direction of price changes

The next model, which was trained is decision tree regression model, which was utilized to forecast the price change of a stock based on the market hours, and change percentage and a window of 1 minute interval. The code assesses the performance of the model of the test dataset using the root mean squared error (RMSE) as the metric.

The results show the RMSE and accuracy of the model on the test dataset, and portray the predicted value, and the change value in a line chart. The chart can be utilized to visualize the performance of the model, and then compare the forecasts with the actual changes in price. If the predicted values are near the actual value, the results indicate that the model is performing well in predicting the price change.



```
RMSE: 0.1053309589450658
+------------------+------------------+
|        prediction|            change|
+------------------+------------------+
|34.002430346426536| 33.95000076293945|
|34.002430346426536|34.040000915527344|
| 34.00525043242284| 33.85150146484375|
|34.002430346426536| 34.15850067138672|
|34.002430346426536| 34.04999923706055|
|34.002430346426536|34.150001525878906|
|34.002430346426536| 34.02000045776367|
|34.002430346426536|34.060001373291016|
| 34.00033628621688| 34.34000015258789|
| 34.00033628621688| 34.34000015258789|
|33.999463182826446|34.349998474121094|
|33.999463182826446|34.369998931884766|
|33.999463182826446|34.369998931884766|
|33.999463182826446|34.380001068115234|
|33.999463182826446|34.369998931884766|
| 34.00033628621688|34.191001892089844|
| 34.00033628621688| 34.22999954223633|
```

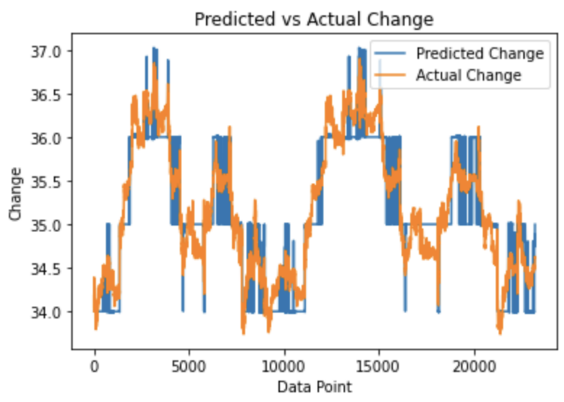


Figure 11 - Prediction results of the Gradient Boosted Regression in terms of change

The last model was trained using a Decision Tree Regressor model to forecast the change in stock prices based on the various features such as “change”, “marketHours”, and “changePercent”. The code utilizes data from a Parquet file, and performs many data preprocessing steps, such as filtering rows with invalid timestamps, casting the timestamp column to the TimestampType, and grouping the DataFrame within minute intervals based on the timestamp column.

Then, it creates a vector assembler to merge features and transforms the DataFrame by utilizing the vector assembler. The DataFrame, which is transformed and split into training and test sets, and a gradient boosting model is explained and trained on the training dataset. The model is utilized to make predictions on the test data, and its performance is assessed by utilizing the RMSE metric. The predicted, and actual values are displayed using a line chart, which is generated with matplotlib.

The line chart displays the predicted and actual change values for the test dataset. The x-axis represents the data, and the y-axis, the change values. The blue line represents the predicted change values, and orange line, the actual change values. The chart provides a visual interpretation of the model’s performance in forecasting the changes in the stock market. If the blue line flows near the orange line, the graph indicates that the model was successful in forecasting the changes in the stock market. If there is a huge difference between the two lines, the graph indicates that the model may not have performed well in conducting accurate forecasts

Machine Learning Using API

One of the advantages of the backup Kafka system described above is the large amount of historical data that it stored could be leveraged to provide an extensive training set for a machine learning model, while still allowing for recent data to be pulled in to make predictions on. To showcase the utility of a machine learning model with access to streaming data, the goal of this portion of the project was to train a model that could be used to predict the price of a stock in the very near future, in this case GM. Typical forecasting methods would use an autoregression model, however stock price auto-correlation rapidly decreases as time goes on. A better predictor of stock price would be to use trading volume in conjunction with the change in price over a short and recent timeframe.

The data as it was pulled from the API was in a .json format with a number of quantitative values and labels. The first step taken was to convert the .json file into a Pandas dataframe. Next the columns holding label data were removed since the scope of the project was limited to a single stock for simplicity. Once the columns were reduced to timestamps, price, volume, change, and market hours the timestamp was set to be the index for the table. Five new columns were calculated using the price information and finding the difference in price from one to five days. In this way the time dependent nature of each data point could be exploited. Next the data was split into feature and non-feature columns. The final step was to convert the features into a vectorized column to be used in the PySpark machine learning model.

Model

The model itself consisted of a pipeline to scale the initial data and a gradient boosting regressor. No hyperparameter tuning was carried out. RMSE was used to evaluate the effectiveness of the model on the test set after it was trained on a training portion of the data, which yielded an RMSE of approximately 0.005. Compared to the average stock price analyzed this would be an acceptable level of error to make conservative trades based on over the short term in conjunction with other external information.

Once the model was trained the API was used again, this time for the most recent data stored. The same transformations described above were applied to this new set with some additional operations. In order to use the machine learning model to make predictions on points in time that had not been directly collected through the API, five rows were added to the new dataframe of the same frequency period of the rest of the data. Since the frequencies used in this example meant that the data occurred on the same day as the pulled data, the stock volume for the day from the most recent data point was carried forward. Once the five new rows were added to the new dataframe the model was used to estimate the price of the five new periods.
Visualization

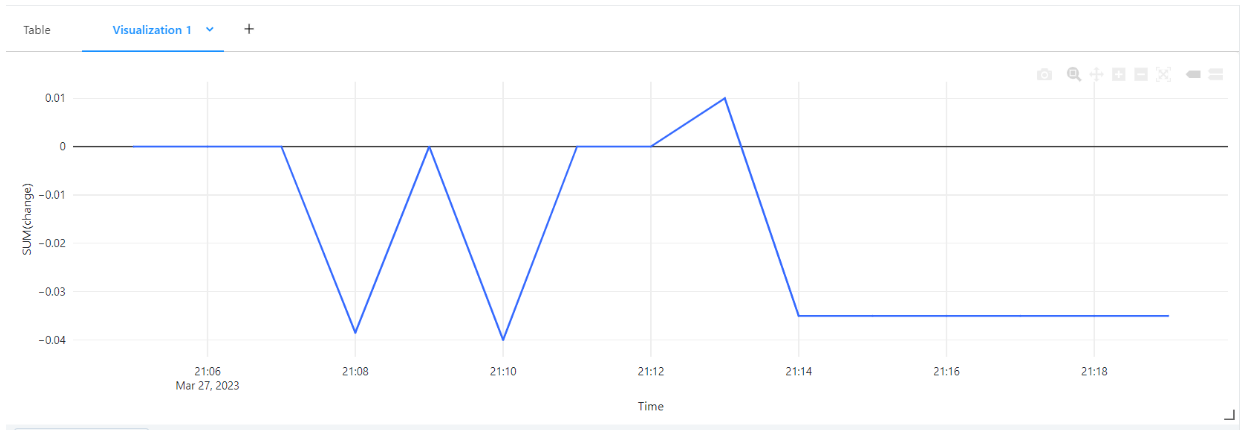


Figure 12 - Line graph to show change in predicted prices

Notebook available in code directory

Conclusion

In conclusion, we offer in this project a comprehensive solution for building a real-time data processing pipeline using Apache Kafka and Apache Spark. The project includes detailed instructions for setting up Kafka and Spark (basic and with authentication), as well as sample code for ingesting data from Kafka, processing it using Spark's Structured Streaming API, and writing the results to Parquet files.

The use of Kafka as a scalable and fault-tolerant message queue, combined with Spark's powerful streaming processing capabilities and Parquet's efficient columnar storage format, provides a robust and scalable framework for building real-time data pipelines that can handle large volumes of data from various sources. Additionally if there is a requirement to look at historic data, a backup server with APIs can be used to distribute the load and make for an efficient service.

It is important to become familiar with concepts like store offsets, checkpoints, topics, schemas and authentication that are key to make these two technologies work together to deliver business results.

Overall, we were able to successfully create a working sample of what we think is a common scenario when for predictive applications using Big Data in motion. We hope that this document could be a useful reference for developers and data engineers who are looking to build real-time data processing pipelines using Kafka and Spark.

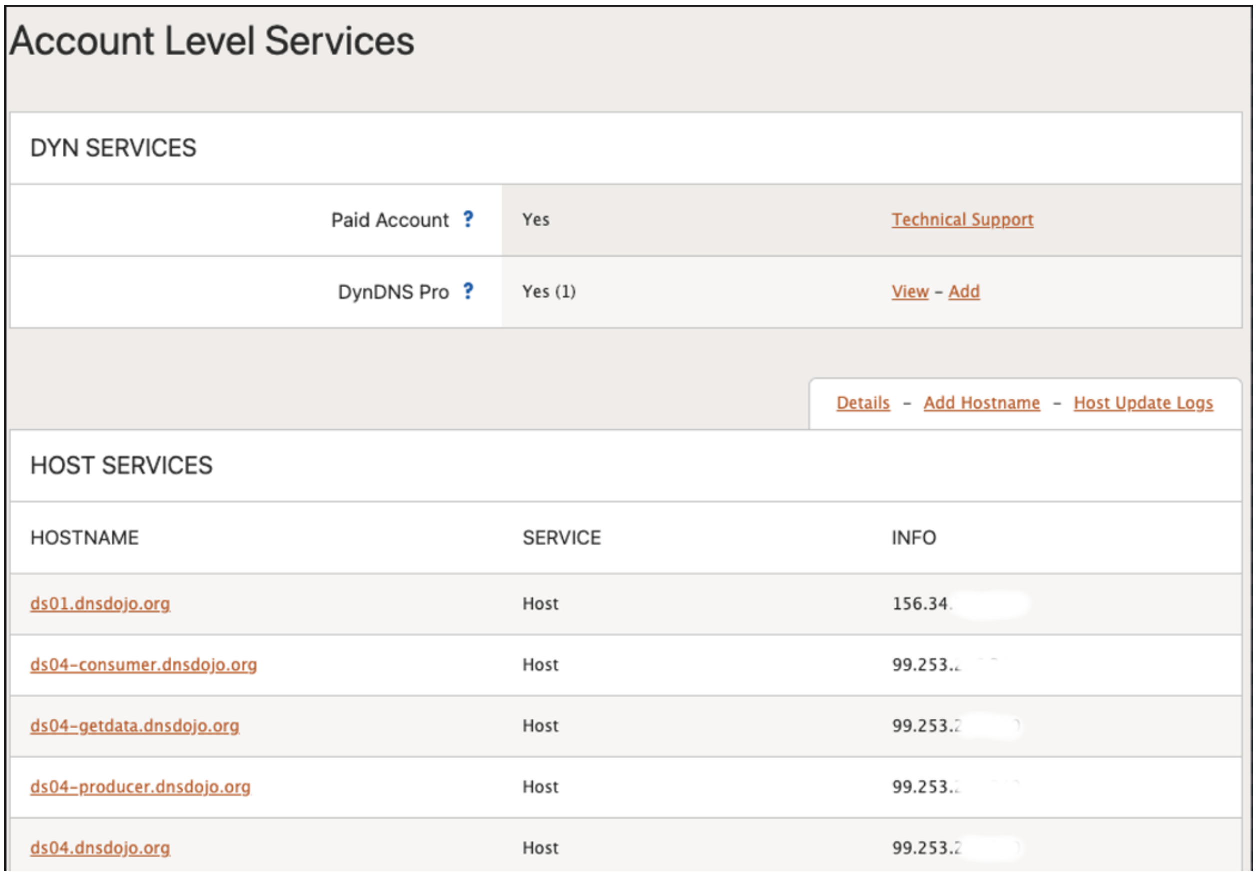
Please note that ALL URLs and Internet facing servers will be decommissioned as of May 14th, 2023 to reclaim the computing resourcing and avoid hacking attacks.

Appendix

DNS Setup

DynDns Services were used to register DNS A records instead of using direct IP addresses. Using direct IP addresses can create issues if the IP address of the Kafka server changes as they are hosted in colleagues' homes with dynamic IP addresses.

To overcome this, we used a DynDNS updater on the home server that automatically updates the IP addresses with DNS registrar if the dynamic IP changes. This way, the fully qualified addresses work without issues.



Account Level Services

DYN SERVICES

Paid Account ?	Yes	Technical Support
DynDNS Pro ?	Yes (1)	View – Add

Details – Add Hostname – Host Update Logs

HOST SERVICES

HOSTNAME	SERVICE	INFO
ds01.dnsdojo.org	Host	156.34.[redacted]
ds04-consumer.dnsdojo.org	Host	99.253.[redacted]
ds04-getdata.dnsdojo.org	Host	99.253.[redacted]
ds04-producer.dnsdojo.org	Host	99.253.[redacted]
ds04.dnsdojo.org	Host	99.253.[redacted]

Code to stream data from Kafka

```
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StructField, StringType, DoubleType,
TimestampType, IntegerType
```

Created a schema to be used later in the process of persisting the data in disk:

```
stocks_schema = StructType([StructField('id', StringType(), True),
StructField('exchange', StringType(), True),
StructField('quoteType', StringType(), True),
StructField('price', DoubleType(), True),
StructField('timestamp', TimestampType(), True),
StructField('marketHours', IntegerType(), True),
StructField('changePercent', DoubleType(), True),
StructField('dayVolume', IntegerType(), True),
StructField('change', DoubleType(), True),
StructField('priceHint', StringType(), True)])
```

Read the Kafka stream into a DataFrame (basic)

```
df = spark \
 .readStream \
 .format("kafka") \
 .option("kafka.bootstrap.servers", "156.34.212.211:9092") \
 .option("subscribe", "stocks") \
 .option("startingOffsets", "earliest") \
 .load() \
 .selectExpr("CAST(value AS STRING)") \
 .select(from_json("value", stocks_schema).alias("data")) \
 .select("data.*")
```

Read the Kafka stream into a DataFrame (advanced)

```
df = spark \
 .readStream \
 .format("kafka") \
 .option("kafka.sasl.mechanism", "PLAIN") \
 .option("kafka.security.protocol", "SASL_PLAINTEXT") \
 .option("kafka.sasl.jaas.config",
'kafkashaded.org.apache.kafka.common.security.plain.PlainLoginModule required
username="admin" password="password";') \
 .option("kafka.bootstrap.servers", "ds04.dnsdojo.org:9092") \
 .option("subscribe", "ds04_messages_2 ") \
 .option("startingOffsets", "earliest") \
 .load() \
 .selectExpr("CAST(value AS STRING)") \
 .select(from_json("value", stocks_schema).alias("data")) \
 .select("data.*")
```

Write the data to disk using parquet format

```
query = df \
 .writeStream \
 .format("parquet") \
 .option("path", "/dbfs/local_disk0/checkpoints2") \
 .option("checkpointLocation", "/dbfs/local_disk0/checkpoints2") \
 .start()
```

Validating that all the data arrived from the Kafka topic:

```
# Read a Parquet file into a DataFrame
df1 = spark.read.parquet("/dbfs/local_disk0/checkpoints2")

# Display the DataFrame
df1.show()
```

```
df1.count()
```

Kafka Server basic configuration

```
# Learn how to install Kafka in Ubuntu
https://tecadmin.net/how-to-install-apache-kafka-on-ubuntu-22-04/
https://kafka.apache.org/quickstart

sudo apt update

## Install java
sudo apt install default-jdk

#Check java
java --version
#Download and decompress file

wget https://dlcdn.apache.org/kafka/3.4.0/kafka_2.13-3.4.0.tgz


tar -xzf kafka_2.13-3.4.0.tgz
sudo mv kafka_2.13-3.4.0 /usr/local/kafka

## Edit zookeeper service
sudo nano /etc/systemd/system/zookeeper.service

### Add this to the file

[Unit]
Description=Apache Zookeeper server
Documentation=http://zookeeper.apache.org
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
ExecStart=/usr/local/kafka/bin/zookeeper-server-start.sh
/usr/local/kafka/config/zookeeper.properties
ExecStop=/usr/local/kafka/bin/zookeeper-server-stop.sh
Restart=on-abnormal

[Install]
WantedBy=multi-user.target

### Create a systemd unit file for the Kafka service:

sudo nano /etc/systemd/system/kafka.service
```

### Make sure to set the correct JAVA_HOME path as per the Java installed on your system.

```
[Unit]
Description=Apache Kafka Server
Documentation=http://kafka.apache.org/documentation.html
Requires=zookeeper.service

[Service]
Type=simple
Environment="JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64"
ExecStart=/usr/local/kafka/bin/kafka-server-start.sh
/usr/local/kafka/config/server.properties
ExecStop=/usr/local/kafka/bin/kafka-server-stop.sh

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload

sudo systemctl start zookeeper
sudo systemctl start kafka


sudo systemctl status zookeeper
sudo systemctl status kafka
```

## Create topic in Kafka

```
cd /usr/local/kafka
bin/kafka-topics.sh --create --bootstrap-server 192.168.2.149:9092 :9092
--replication-factor 1 --partitions 1 --topic testTopic
```

### Replication factor
https://blog.clairvoyantsoft.com/steps-to-increase-the-replication-factor-of-a-kafka-topic-a516aefd7e7e

### Check topic created

```
bin/kafka-topics.sh --list --bootstrap-server 192.168.2.149:9092
```

## Check stream

### In one terminal

```
bin/kafka-console-producer.sh --topic testTopic --bootstrap-server
localhost:9092
```

```
This is my first event
This is my second event
```

### In other terminal

```
bin/kafka-console-consumer.sh --topic inventory --from-beginning
--bootstrap-server 192.168.2.149:9092
This is my first event
This is my second event
```

## Terminate Kafka event

```
rm -rf /tmp/kafka-logs /tmp/zookeeper /tmp/kraft-combined-logs
```

## How to delete kafka topic

https://contactsunny.medium.com/manually-delete-apache-kafka-topics-424c7e016ff3

```
cd /usr/local/kafka
/usr/local/kafka/bin$ ./kafka-topics.sh --bootstrap-server
192.168.2.149:9092 --delete --topic testTopic
```

## Produce to Kafka from other machine with python

https://www.confluent.io/blog/kafka-client-cannot-connect-to-broker-on-aws-on-docker-etc/
https://www.youtube.com/watch?v=VHngQ-moXIE

# In kafka directory:

```
cd config
nano server.properties
```

### Insert this lines

https://www.confluent.io/blog/kafka-listeners-explained/

```
listeners=INTERNAL://0.0.0.0:19092,EXTERNAL://0.0.0.0:9092
listener.security.protocol.map=INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
advertised.listeners=INTERNAL://192.168.2.149:19092,EXTERNAL://156.34.212.
211:9092
inter.broker.listener.name=INTERNAL
```

### Run this

```
conda install -c conda-forge kafka-python

from kafka import KafkaConsumer

TOPIC_NAME = "testTopic"
```

```
consumer = KafkaConsumer(TOPIC_NAME,
bootstrap_servers=['192.168.2.149:9092'])
for message in consumer:
    print(message)
```

## Star kafka automatically after reboot

https://stackoverflow.com/questions/34512287/how-to-automatically-start-kafka-upon-system-startup-in-ubuntu

## Setup Kafka Schema

https://medium.com/slalom-technology/introduction-to-schema-registry-in-kafka-915ccf06b902

Load data from Yahoo Finance to Kafka Producer

```
import json
from kafka import KafkaProducer
import yliveticker
import logging
import time
```

Using logging library we are creating a function to log the process of loading info to kafka

```
logging.basicConfig(
   filename="stocks_producer.log",
   level=logging.INFO,
   format="""%(asctime)s:%(levelname)-8s%(message)s""",
)
```

The next function takes the message from the Yahoo Finance API and send it to the kafka topic using a producer

```
def on_new_msg(ws, message: dict) -> None:
   print(message)
   producer.send(topic_name, message)
```

Here we are defining the name of the kafka topic and and configuring the kafka producer with the server address and the serializer

```
topic_name = "stocks"
logging.info("Initializing kafka producer.")
producer = KafkaProducer(
   bootstrap_servers=["192.168.2.149:19092"],
value_serializer=json_serializer
)
```

This function integrates the functions of logging, reading data from Yahoo Finance using yliveticker python library and sending the message to kafka using a function with the KafkaProducer

```
def start_yahoo_streaming(tickers: list) -> None:
    """Recursive function that starts yahoo finance streaming and if error
occurs restarts itself."""
    logging.info(f"Starting streaming with {tickers} casthags.")
    yliveticker.YLiveTicker(on_ticker=on_new_msg, ticker_names=tickers)

    logging.error(f"Streaming stopped.")
    time.sleep(2)
    logging.info("Restarting streaming.")
    print("Restarting streaming.")
    start_yahoo_streaming(tickers)
```

This function start the program with the define tickers to stream

```
if __name__ == "__main__":
    tickers = ["GM"]
    start_yahoo_streaming(tickers)
```

Code Directory Explanation

Group10-APICode_dotNet7.zip

- Visual Studio 2022 Community Edition with .NET Core 7
- API Solution Code with three projects
    - DS04_Consumer
        - Consume data via API to load into Kakfa
    - DS04_GetData
        - Post data from Kafka to SQL Server
    - DS04_Producer
        - Provide two API's to publish data
            a. Provide data using date range
            b. Provide data via streaming

Group Project ML Visualization and ML.ipynb

- Code to ingest data from backup Kafka server API using a date range
- Perform Machine Learning on this acquired data
- Link:
  https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/2755299084972461/3220010408843720/4289408415433743/latest.html
  https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/7457838676122636/1818018706218555/8862643189347633/latest.html

GetData_To_Kafka.ipynb

- Code to get data from Yahoo Live Ticker into Kafka

GetData_From_Kafka.ipynb

- API code to get data from backup server using API and start and end time stamps