Workshop Report
Stock Price Prediction System using Finnhub Data-stream and Apache Spark

Anastasiya Merkushova and Raphael Waltenspül

University of Basel
Distributed Information Systems (DIS) course
Spring Semester 2024

Abstract. Due to the increasing popularity of large language models, transformer models have become widely discussed and applied across various domains. It is not surprising, therefore, that efforts are being made to utilize these models for predicting stock market prices. In this article, we introduce the *Finnhub Stock Estimator* (FSE).
The FSE implements a stock price estimator that utilizes streamed stock prices. Specifically, the FSE streams a subset of $n = 3$ stock prices from the S&P 500 Index and trains a *Long Short-Term Memory* (LSTM) network to predict stock prices within this subset.
Several challenges arise when performing such estimations. Firstly, it is essential to receive and stream current stock prices in real-time as comprehensively as possible, followed by immediate processing to further train the model and estimate price trends. To achieve this, a dedicated hardware and software system architecture that operates in a decentralized manner is required.
This paper focuses on the architecture of a system that enables these requirements. The LSTM model demonstrates the functionality of the system and serves as a model for future implementations.
In conclusion, the advantages and limitations of the proposed system are discussed, and areas for future improvement and development are identified.

1 Introduction

The live estimation of stock market values has extensive requirements. Data must be streamed in *real time*[1] and at the highest possible sampling rate. Data loss should be prevented. The streamed data should be pre-processed. Ideally, minimum, maximum, mean, open, and close prices should be determined in the stream. Furthermore, the LSTM model should be able to be trained with the data stream and also estimate a value in real time. The estimated value should be stored in a database and later visualized via a user interface (UI).

To achieve this, we require a dedicated hardware and software system landscape that functions in a decentralized manner. This paper focuses on the architecture of such a system that meets these requirements. Specifically, it examines the data source from Finnhub, the stream producer using Kafka, the consumer and worker utilizing Spark, the cluster manager with Hadoop, the TensorFlow model, the database, and the UI.

1.1 Related Work

The following are relevant articles related to this paper. Due to the workshop nature of this work, these are commonly tutorials, documentation, and papers related to these topics.

Finnhub Stock Estimator Finnhub's mission is to democratize financial data, and it advertises with the slogan, *"we are proud to offer a FREE real-time API for stocks, forex, and cryptocurrencies."* It provides an extensive free API to stream data [1]. Our data stream is built based on this API.

Kafka Producer Apache Kafka is a distributed streaming platform designed for high-throughput, low-latency data streaming. It serves as a bridge between data producers and consumers, ensuring that data is transmitted efficiently. In our architecture, Kafka acts as the stream producer, capturing real-time stock data from Finnhub and distributing it to various consumers for processing and analysis. The core components of Kafka include topics, producers, consumers, and brokers, which work together to manage and distribute the data streams.

Spark Spark applications run as independent process groups on a cluster, coordinated by the `SparkContext` object in the main program. To run in a cluster, the `SparkContext` can connect to different types of cluster managers that allocate resources for applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that perform computations and store

[1] In the context of this work, we refer to this as *soft real time*. In this case, a response time is only statistically guaranteed. Although such systems typically process all incoming inputs quickly enough, this is not guaranteed.

data for the application. Then, the application code is sent to the executors. Finally, `SparkContext` sends tasks to the executors for execution [2].

A more detailed description of working with Apache Spark can be found in the documentation, *Apache Spark - A Unified Engine for Large-Scale Data Analytics* [2].

Hadoop To set up Spark in cluster mode, we decided to utilize Hadoop and YARN. The setup can be quite complex; therefore, we primarily followed the article *Running Spark on Top of a Hadoop YARN Cluster* [3].

TensorFlow LSTM Training and predicting models is a vast field. We chose to use an LSTM following the conference paper *Stock Market Price Prediction Using LSTM RNN* [4]. The implementation of prediction and training of the LSTM follows the article [5], which is based on the paper.

MongoDB MongoDB is a NoSQL database known for its scalability, flexibility, and ease of use in handling large volumes of unstructured data. It stores data in JSON-like documents, making it ideal for applications requiring real-time analytics and data visualization. In our system, MongoDB is used to store the processed and predicted stock values, providing a robust backend for querying and retrieval.

Streamlit Streamlit is an open-source app framework specifically designed for creating and sharing data applications. It allows developers to build web applications for data science and machine learning projects with minimal effort, using simple Python scripts. In this system, Streamlit is employed to create a user interface for visualizing real-time stock predictions and historical data stored in MongoDB.

2 Concepts

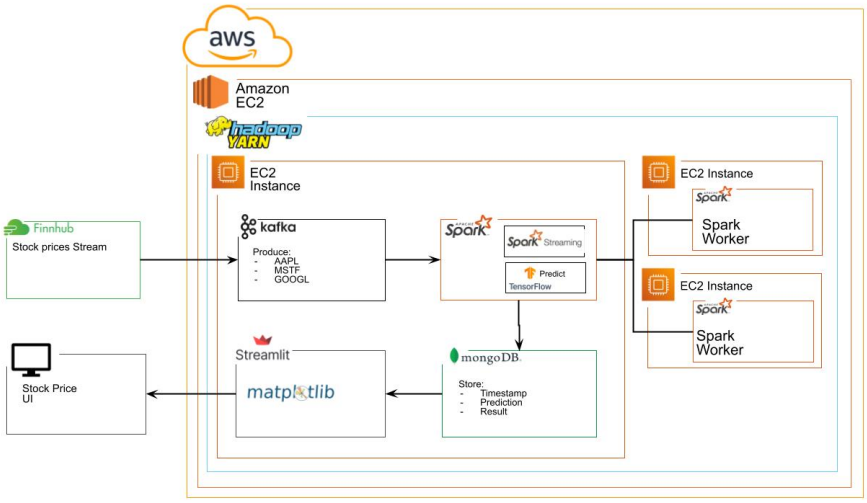

Fig. 1: **FSE System Architecture**: The architecture demonstrates the flow from data ingestion to visualization. Finnhub data is collected via API and ingested into an Apache Kafka topic. Apache Spark processes the streaming data, chunking it and computing needed values (minimum, maximum, mean, open, and close values). These chunks are fed into an LSTM network for real-time training and prediction. The predictions are stored in a MongoDB database, and it is retrieving these predictions upon user query, rendering the requested plots on a Streamlit webpage.

The core concept of this workshop is processing stream data utilizing a distributed architecture on multiple Amazon EC2 instances using Apache Spark. Given this prerequisite, we developed the following system architecture to address our problem[2]. The architecture of the FSE system is illustrated in fig. 1. Finnhub data is collected using an API and ingested into an Apache Kafka topic. The system streams time-series data, and the Spark stream processor consumes these data and collects them into chunks of a given time interval (in our specific case, minutes). It then determines the minimum, maximum, mean, open, and close values.

These chunks are then fed to the LSTM network, which is implemented in a producer-consumer structure, allowing for training and prediction simultaneously, following the approaches of papers [5, 7, 6]. The predictions are then stored in a MongoDB database.

[2] As a result of the work on the project and the experience gained through it and the exercises, there were adjustments to the architecture compared to the project proposal. These are listed in the appendix.

When a user queries the data, a it is retrieving the predictions from the MongoDB and renders a webpage containing the requested plot using streamlit.

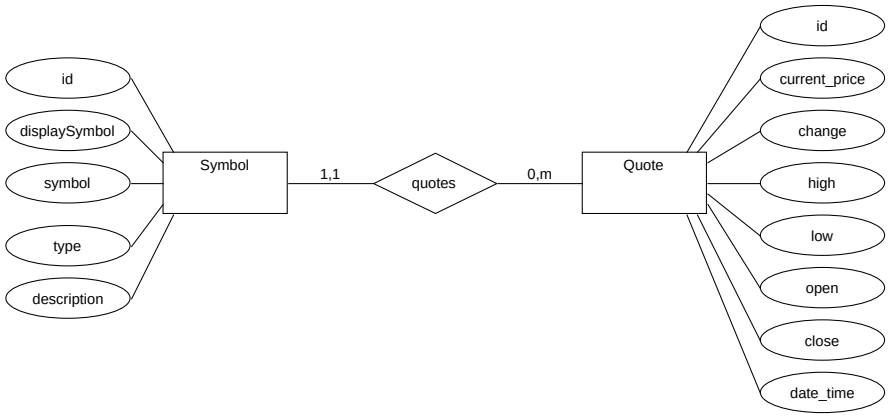

Fig. 2: **ER Diagram**: The data model consists of a set of *Symbols*, which is manually defined to select the companies to track. Each *Quote* contains the current price along with metrics such as open, close, high, and low prices. The *Quote* data is streamed at a frequency of $f = 166 \cdot 10^{-3}$ Hz, with the datetime stamp applied upon data retrieval.

The data model, depicted in fig. 2, consists of a set of $n = 3$ *Symbols*, and for each Symbol, a stream (time series) of *Quotes*. The selection of the *Symbols* is done manually, allowing for the choice of companies to track. Each *Quote* includes the current price along with additional metrics such as open, close, high, and low prices. The *Quote* data is streamed at a frequency of $f = 166 \times 10^{-3} Hz$, the maximum allowed by the free quota. The date-time stamp is set upon data retrieval.

3 Implementation

3.1 Kafka

Apache ZooKeeper and Apache Kafka Broker are the core of the Kafka system. **The Kafka Broker** acts as the central node, handling the receipt, storage, and distribution of message streams, while managing the interaction between producers (event generators) and consumers (event processors) to ensure efficient message flow. **Apache ZooKeeper** manages the Kafka cluster by tracking brokers and their topics, ensuring smooth coordination, high availability, and fault tolerance within the system.

In our system, we have implemented a `KafkaProducer.py` responsible for reading the Finnhub stock data and pushing messages to a specific topic within the Kafka Broker. The producer then transforms each entry in the dataset into JSON format, as shown in **??**, before sending it to the broker. Kafka consumers, which will be discussed in detail in the Spark subsection, subscribe to these topics and receive all messages related to the subscribed topics. This enables consumers to process and analyze the streaming data in real-time.

3.2 Spark

The Apache Spark components run on a clustered node consisting of two Spark workers and one main node. The node is set up on a Hadoop cluster using YARN as the resource manager. To process the data, we run two main processes: the Spark `stream_processor.py` and the `collector.py`. The stream processor reads data from Kafka and stores it on the HDFS distributed file system. Additionally, the processor sets a *watermark* with $\Delta t = 600s$ on the ingested time to ensure that no duplicates are stored. The collector reads the data from the distributed filesystem, calculates the min, max, mean, open, and close values, and feeds the data into the LSTM model.

3.3 TensorFlow LSTM

The model is implemented in `ModelWrapper.py`. This model wrapper essentially starts two threads: a training thread and a prediction thread. If new training data is available, it is provided by the Spark collector into a blocking queue. Due to the nature of LSTM models, the current model can easily be updated using the TensorFlow fitting library. The prediction thread predicts the stock prices using the Keras model for the next 10 epochs. Upon successful prediction, the dataframe is stored in MongoDB.

3.4 Streamlit

The user interface for visualizing stock predictions is implemented using Streamlit. Since Streamlit is designed for ease of use, it utilizes only one script, `app.py`,

which connects to databases, retrieves predictions, and visualizes them. Moreover, users can input their desired stock ticker symbol and view historical data in a customizable date range. The application uses yfinance to retrieve real-time historical stock data.

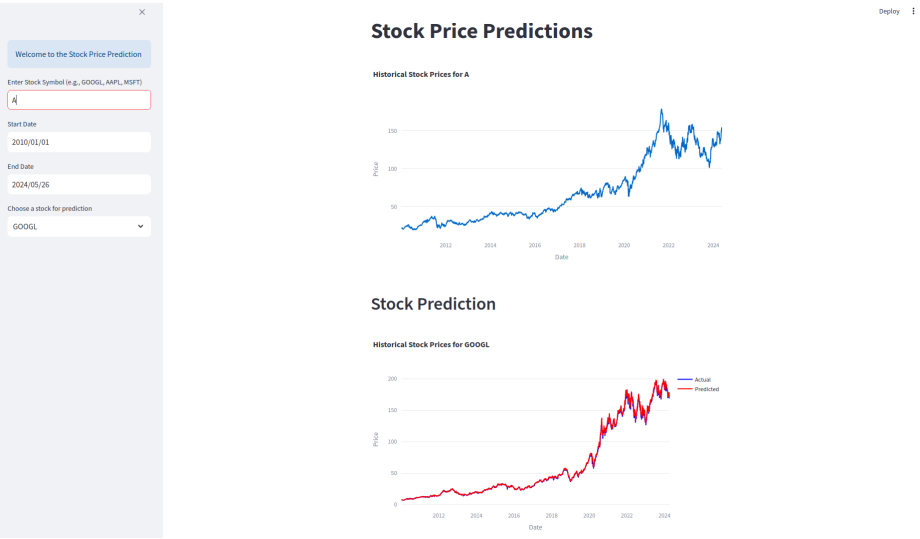

Fig. 3: **Streamlit Website**

3.5 Run the FSE

To run the FSE one have to setup first the system section 3.5.1 and then run the software stack section 3.5.2.

The FSE source is handed in in the structure shown in fig. 4:

Setup To setup the whole Software stack one has to perform the following steps. Every step is documented with detailed commands in the appendix.

1. Setup the AWS EC2 instances.
2. Install Java.
3. Install Python.
4. Install Hadoop with YARN.
5. Install Kafka.
6. Install Spark.
7. Install MongoDb.

```
./finnhub-stock-estimator
├─ dashboard
│  └─ app.py → App to run the Steramlit Web Ui
├─ doc
│  ├─ images
│  │  └─ *.png → Contains all images for readme.md
│  ├─ proceedings
│  │  ├─ ...
│  │  ├─ proceedings.pdf → Contains the report without appendix.
│  │  └─ ...
│  └─ proceedings-apendixx
│     ├─ ...
│     ├─ proceedings.pdf → Contains the report with appendix.
│     └─ ...
├─ estimator
│  └─ lstm-model.keras→ Contains a well trained keras lstm model.
├─ finnhub_producer
│  ├─ ...
│  ├─ FinnhubProducer.py → Runs the Kafka Producer.
│  └─ ...
├─ jars
│  ├─ ...
│  ├─ *.jar → Containing external libraries for spark
│  └─ ...
├─ predictor
│  ├─ ...
│  ├─ estiamtor-kafka.py → Runs the lstm train and predict thread
│  └─ ...
├─ processed
│  └─ part*.csv → From predictor already predicted files
└─ StreamProcessor
   ├─ stream_processor.py → Runs the stream processor
   └─ collector.py → Runs the collector.
```

Fig. 4: **Folder Structure**: In the shown folder structure all relevant implementations are shown.

Startup Follow for startup the follwing steps.
1.) Start the Kafka Server and producer

```
zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
```

```
kafka-server-start.sh $KAFKA_HOME/config/server.properties
```

```
FinnhubProducer.py
```

2.) Start the Spark Consumer

```
spark-submit --packages org.apache.spark:spark-avro_2.12:3.5.1 org
    ↪.mongodb.spark:mongo-spark-connector_2.12:10.3.0 org.apache.
    ↪spark:spark-sql-kafka-0-10_2.12:3.5.1 ./StreamProcessor/
    ↪stream_processor.py
sudo apt update
```

3.) Download jar and add manually:

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2
    ↪.12:3.5.1 --jars ./jars/spark-avro_2.12-3.5.1.jar ./
    ↪StreamProcessor/stream_processor.py
```

4.) Start the collector:

```
spark-submit --packages ./StreamProcessor/collector.py
```

5.) Start MongoDB:

```
sudo systemctl start mongod
```

6.) Run the Modell predictor

```
spark-submit ./StreamProcessor/estimator-kafka.py
```

7.) Start the Streamlit Website

```
streamlit run ./dashboard/app.py
```

4 Conclusion

In this project, we designed and implemented an application known as the Finnhub Stock Estimator (FSE). This system efficiently processes incoming streaming data from Finnhub through Kafka, then applies transformations and analysis using Apache Spark to predict stock behaviors. Given the learning goals of this project, we outline the challenges and insights gained in this section.

Setting up a distributed cluster is inherently challenging and requires a great deal of patience. Multiple interdependent components must be configured correctly. In our case, the setup included an EC2 cluster on Amazon, followed by Hadoop with YARN, Spark, Kafka, our stream processor, the LSTM model, the database, and the website. Many potential issues can arise within this hierarchy, such as incorrect network settings, configurations, or software versions. Software version dependencies, in particular, proved to be a significant challenge. For instance, in the latest Spark version 5.1, the MongoDB drivers did not function correctly, whereas in version 4.1, the JSON serializer was incompatible.

From the outset, we did not expect to build a highly accurate price estimator. However, we were pleasantly surprised when the first predictions were generated. Despite this initial success, it quickly became apparent that the model's predictions could be erratic and often yielded unreliable results.

Nevertheless, building a system that processes and evaluates data continuously was an interesting and valuable experience. This project could serve as a foundation for further work, such as continuous feature co-embedding on brain wave measurements.

References

1. Finnhub Finnhub API Documentation. (2024,5,23), https://finnhub.io/docs/api
2. Spark, A. Cluster Mode Overview. (2024,5,22), https://spark.apache.org/docs/latest/cluster-overview.html
3. Akamai Running Spark on Top of a Hadoop YARN Cluster. (2024,5,21), https://www.linode.com/docs/guides/install-configure-run-spark-on-top-of-hadoop-yarn-cluster/
4. Pawar, K., Jalem, R. & Tiwari, V. Stock market price prediction using LSTM RNN. *Emerging Trends In Expert Applications And Security: Proceedings Of ICETEAS 2018*. pp. 493-503 (2019)
5. ParamRaval Stock Price Prediction using Machine Learning with Source Code. (2024,5,9), https://www.projectpro.io/article/stock-price-prediction-using-machine-learning-project/571
6. A. Moghar et al., 2020, Stock Market Prediction Using LSTM Recurrent Neural Network.
7. S. Sreelekshmy, et al, 2017, Stock price prediction using LSTM, RNN and CNN-sliding window model,