Real-time Stock Market Analytics for Improving Deployment and Accessibility using PySpark and Docker

Suyeon Wang
wangsunny6143@gmail.com
Dong-A University
Busan, South Korea

Jaekyeong Kim
2143575@donga.ac.kr
Dong-A University
Busan, South Korea

Yoonsang Yang
yoonsang.developer@gmail.com
Dong-A University
Busan, South Korea

Jinseong Hwang
jinseong.dev@gmail.com
Dong-A University
Busan, South Korea

Jungkyu Han
jkhan@dau.ac.kr
Dong-A University
Busan, South Korea

Sejin Chun
sjchun@dau.ac.kr
Dong-A University
Busan, South Korea

ABSTRACT

Making timely-decisions amid the massive influx of financial data is one of the essential features of stock market analytics. Many stock market analytics should provide functionalities that compute multiple technical indicators simultaneously and detect breakout situations. The DEBS 2022 Grand Challenge (DEBS22 GC) competition requires to answering two types of queries: technical trend indicators and detection of crossover patterns. In response to the competition, we propose a real-time stock market analytic solution using PySpark and Docker. Our solution calculates the technical trend indicator—*Exponential Moving Average*(EMA)—in real-time with the window function. With the technical indicators computed, we detect the *breakout pattern* that helps determine either buy or sell stocks. Our solution not only improves the speed of deploying applications using a Docker container image but also can be accessed easily via a web-based Jupyter notebook.

CCS CONCEPTS

• **General and reference** → *Performance*; • **Information systems** → **Stream management**.

KEYWORDS

stock market analytics, apache spark, docker, crossover, moving average, timeseries database

ACM Reference Format:
Suyeon Wang, Jaekyeong Kim, Yoonsang Yang, Jinseong Hwang, Jungkyu Han, and Sejin Chun. 2022. Real-time Stock Market Analytics for Improving Deployment and Accessibility using PySpark and Docker. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27-June 30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524860.3539652

1 INTRODUCTION

Making real-time decisions from massive streams of financial data plays a key role in modern stock market analytics. The analytics has to process streams of information on both prices and transactions within a few seconds during trading hours. During recent two years, the COVID-19 pandemic has impacted the rapid growth of the stock market in terms of market size and the number of transactions [2].

The DEBS22 GC requires participants to propose a solution that generates answers to two types of queries. The first query asks to calculate an exponential moving average (EMA), which is one of the popular technical indicators that provide advice on the trend of stock trading. The EMA places a greater significance on the most recent data points as an extended version of the moving average (MA). The second query asks the participants to determine whether to either buy or sell a stock at each crossover. The crossover refers to a point where two EMAs of different time intervals cross. The crossover is used to identify a breakout pattern in which the price of one or more securities will either rise or go down. In other words, a *bullish* pattern is an upward trend in the price of a security, whereas a *bearish* pattern is a downtrend in the price of a security.

In response to the DEBS22 GC, we propose an extensible and practical solution, which is built on top of PySpark[1] and Docker[2]. Our solution can answer the two queries they want to solve and provides extensions to improve the reproducibility and accessibility of our solutions. Specifically, our solutions encompasses components and extensions w.r.t functional and non-functional requirements, respectively. The first of the main components calculates the EMAs extracted by applying a window function to financial data streams. The second one detects the breakout pattern from the crossovers with the EMAs and generates a buy, sell, or stay advice event per symbol. Finally, the last component delivers the generated events to a remote evaluation platform.

The remainder of this paper is organized as follows. Section 2 introduces the technical background and Section 3 describes the datasets and business questions. Section 4 presents our solution in detail. In Section 5, we evaluate the results of the proposed solution. Section 6 describes extended modules for achieving non-functional requirements. Section 7 addresses challenges faced in the current version of our solution. The final section presents our conclusions and future directions.

[1]https://spark.apache.org/docs/latest/api/python/
[2]https://www.docker.com/

2 TECHNICAL BACKGROUND

This section describes the PySpark and Docker used in the proposed solution for the DEBS 2022 GC.

2.1 PySpark: Collaboration of Apache Spark and Python

PySpark allows users to write Apache Spark applications using Python API[6]. Apache Spark, which is an open-source and distributed data processing framework, is widely used for large-scale data analytics. Hence, Python is a general-purpose, high-level programming language in support of various and powerful external libraries such as Keras[3]. Thus, PySpark supports many features of streaming analytics based on Apache Spark and works with a wide range of external libraries.

We leverage key data structures and methods in the structured streaming programming model[4]. A streaming dataframe, window function, and watermark operator are utilized. First, the streaming dataframe represents an unbounded input table where tick data that arrives at the system are treated as *data streams*. Second, the window function extracts a portion of the input stream, e.g., events in the last 5 minutes. This function can compute an aggregate function in an incremental fashion[8]. The computation combines historical data of the previous windows with new data of the current window for aggregation whenever new data arrive. Last, the watermark operator handles events that arrive at the system lately. This operator waits for processing late events up to a specified threshold on how late the data is expected.

2.2 Docker: Enabling Rapid Deployment

Docker can facilitate both the reproducibility and the rapid deployment of applications. It is a virtualization platform that composes software in a package called a container[9]. In addition, users can make up multiple new containers horizontally to perform testing under different configurations in execution environments.

3 DATASETS AND BUSINESS QUESTIONS

This section introduces the stock market datasets provided by DEBS22 GC and describes business queries 1 and 2 in detail.

3.1 Infront Financial Datasets

Infront financial dataset[3] contains 289 million tick data events from November 8 to 14th, 2021. Specifically, the dataset has 5,504 equities and real-time trading events on the European stock exchanges in Paris, Amsterdam, and Frankfurt. The attribute names in the dataset schema are $< id, SecType, Last, Trading\ time, Trading\ date >$. In detail, *id* indicates a unique identifier for a symbol with respective exchange, e.g., Paris(FR); *SecType* indicates a security type to decide either an equity or index; *Last* indicates the stock trade price of the last event; Both $Trading\ time$ and $Trading\ date$ indicates date and time values at which the last event is created by event sources, respectively. Since the dataset size is enormous, e.g., about 24.9GB, the dataset has to be processed in multiple batches of messages as price event streams.

Listing 1 shows the schema of event messages retrieved from an evaluation platform. Specifically, the variables of *symbol*, *sectype*, *lasttradeprice*, *lastTrade* in the message structure are identical to the attribute names of *id*, *SecType*, *Last*, and $cov(Trading\ Date, Trading\ Time)$ in the dataset, respectively. Here, the *lastUpdate* indicates an ingestion time that captures the time at which a price event was received in our solution. The function *cov* returns timestamp-typed data by combining values of both arguments into one. Note that we utilize the schema to process streams of price events throughout this paper.

Listing 1: The schema of event messages

```
message Event {
  string symbol = 1;
  SecType sectype = 2;
  float lasttradeprice = 3;
  google.protobuf.Timestamp lastUpdate = 4;
  google.protobuf.Timestamp lastTrade = 5;
}
```

3.2 Business Queries

3.2.1 Query 1: EMA as a technical indicator. Query 1 asks to calculate the latest EMA per symbol in a continuous fashion. Especially, the purpose of EMA is to track a price of a given stock with time. The EMA gives more importance to recent price data while less importance to previous price data. The EMA indicator produces buy and sell signals.

The conditional statement of the EMA is defined as the follows:

$$EMA^j_{w_i} = \left[Close_{w_i} \cdot \left(\frac{2}{1+j}\right)\right] + \underbrace{EMA^j_{w_{i-1}}}_{\text{prev. window}} \cdot \left[1 - \left(\frac{2}{1+j}\right)\right] \quad (1)$$

where w_i denotes i-th tumbling window with 5 minutes duration, j denotes smoothing factor for EMA with $j \in \{38, 100\}$, $Close_{w_i}$ denotes last price event observed within window w_i, and $EMA^j_{w_{i-1}}$ denotes the EMA calculated within the previous window w_{i-1}.

The EMA reacts more significantly to the last price event than changes in the previous ones. Specifically, multiple price events are observed within a fixed width of windows. Each window is a tumbling window that does not overlap with each other. The EMA of the current window is calculated by giving more weighing on the latest price while giving less weighing on the last EMA of the previous window.

Here, we denote a short-(and long-) term exponential moving average with 38(and 100) days of the smoothing factor by EMA38(and EMA100), respectively.

3.2.2 Query 2: Breakout pattern detection. Query 2 asks to detect whether or not a breakout pattern happens from a crossover point. The bullish pattern signals a change in the uptrend, especially when the short-term moving average crosses above the long-term. A buy advice event is triggered when detecting the bullish pattern. On the other hand, the bearish pattern signals a change in the downtrend, especially when the short-term moving average crosses below the

[3]https://keras.io/
[4]https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

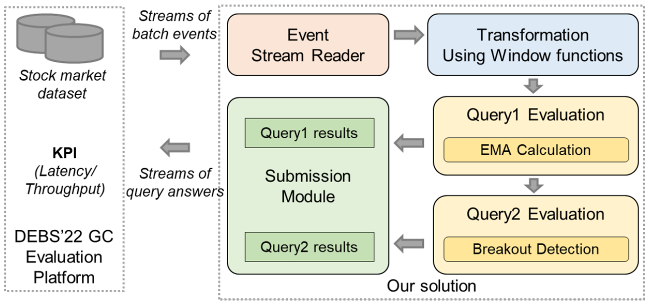

Figure 1: The overall process of the proposed solution

long-term. A sell advice event is triggered when the bearish pattern happens.

The conditional statements of both bullish and bearish patterns are defined as follows:

- Bullish pattern: can be detected **iff** $EMA^{38}_{w_i} > EMA^{100}_{w_i}$ and $EMA^{38}_{w_{i-1}} \leq EMA^{100}_{w_{i-1}}$. Subsequently, a *buy event* has to be generated when the short-term EMA crosses above the long-term EMA.
- Bearish pattern: can be detected **iff** $EMA^{38}_{w_i} < EMA^{100}_{w_i}$ and $EMA^{38}_{w_{i-1}} \geq EMA^{100}_{w_{i-1}}$. Subsequently, a *sell event* has to be generated when the long-term EMA crosses above the short-term EMA.

Here, the $EMA^{38}_{w_i}$ and $EMA^{100}_{w_i}$ indicates the smooth factors of 38 and 100 days, respectively, The w_i indicates an instance of i-th tumbling window.

4 OUR SOLUTION

Figure 1 shows the overall process of the proposed solution. The overall process consists of five stages. In the first stage, our solution receives input streams of batch events from the DEBS22 GC evaluation platform. The second stage performs the data transformation with the window function. In the third stage, our solution produces results of Query 1 using the EMA calculation component. The fourth stage detects a breakout pattern to produce results of Query 2. At the same time, the results of both queries are delivered sequentially to the evaluation platform via the submission module. The evaluation platform measures KPIs of the latency and the throughput.

4.1 Reading streams of stock market data

The event stream reader receives price event streams for each batch per request. The price events are loaded into memory, and each has 10,000 records. This component converts price event streams into CSV files and stores them into a persistent disk whenever events arrive. Thus, the price events are appended into the streaming dataframe.

It is worth noting that Apache Spark assigns a specified path of stored files to a streaming source. Then, it creates a streaming dataframe automatically from the streaming source.

4.2 Transforming data using window functions

This component extracts the last price events of the current window using window functions to calculate both EMA38 and EMA100. Specifically, tumbling windows with a length of five minutes and a period of five minutes have to be applied to the streaming dataframe. The component extracts a record of the last price event of the current window whenever the window is triggered.

Here, we used the watermark function to process late events. In particular, we set the wait interval of the watermark function to five minutes so that the window function accepts the data which arrives within five minutes. In other words, data that arrive later than five minutes are discarded.

4.3 Calculating EMAs

This component performs calculations of both EMA38 and EMA100 in Query 1. To help your understanding, let us consider Equation (1) again. It is important to keep track of computed values of both EMA38 and EMA100 to speed up calculating the EMA value of the next window.

For this calculation, the EMA values of all symbols were initialized with 0.0. At the initial window, none of the symbols has EMA values. We create an *EMA* table that traces the latest values of computed EMAs of all symbols for consecutive calculations of each window instance. The latest values per symbol in the table are regarded as EMA values of the previous window in calculating EMAs of the current window.

To calculate the current EMA values, it finds the latest EMA values of given symbols in the table. Then, it applies Equation (1) to both the found values and the last price event of the current window. After that, we update both the previous and the current EMA values in the table with the calculated results. We renew values of both EMA38 and EMA100 relating to symbols only required for the evaluation to alleviate the overhead of the EMA table search. Finally, we send the latest values of both EMA38 and EMA100 for only lookup symbols by DEBS22 GC.

4.4 Detecting crossover patterns

This component identifies the crossover pattern per symbol with both EMA38 and EMA100 values of consecutive windows. According to the program logic of Query 2, we create a buy advice event when a bullish pattern occurs. And we create a sell advice event when a bearish pattern occurs. For other cases, a stay advice event has to be generated. The difference between the corresponding values of EMA38 and EMA100 may incur breakout patterns. We create a *diff* table that records the difference values constantly. The table is selectively updated only for symbols of interest. We select only both sell and buy advice events(except for the stay advice events). Then we filter them out to obtain the last three advice events for only lookup symbols by DEBS22 GC. Finally, this component sends the filtered events to the submission module.

4.5 Submitting answers continuously to the evaluation platform

The submission module encodes the query results into messages(as shown in Table3.1) and conveys them to the evaluation platform through gRPC API[5]. Especially, the message of Query 1 contains multiple numbers of records consisting of symbol, EMA38, and EMA100 indicators corresponding to the sequence number of the batch and the benchmark id. The message of Query 2 contains the last three crossover events for lookup symbols corresponding to the sequence number of the batch and the benchmark id. In addition, each crossover includes the symbol, the timestamp, the security type, and the signal type of either buy or sell advice event. The evaluation is terminated when the last batch number is reached.

5 EVALUATION

5.1 Experimental setup

The execution environment of our solution was built on the top of PySpark 3.2.0, Java 8, and Linux. We utilized the structured streaming programming model that allows users to support an incremental event aggregation using the window function and handle late events using the watermark functions. We created a Docker container file to enable users to establish the same execution environment like ours and deploy our solution on the user execution environment. To improve the reproducibility, accessibility, and usability, we generated a Jupyter notebook[6] file that enables users to test our solution via a web browser.

We developed our solution as an open-source project, and it can be accessed on our Github[7]. Moreover, both source codes and quick tutorials for three extensions in Section 6 are available without any restrictions.

5.2 Experimental results

We as `Group-4` evaluated our solution in terms of latency and throughput as key performance indicators(KPIs). The latency is defined as the elapsed time from start time at which each batch of price events is received from the evaluation system to the finish time at which the query result is submitted. Each batch had 10,000 price events and a list of lookup symbols required for the submission. Up to 2000 batches, we measured an average latency of all price events received.

As the results, we obtained 10,115 and 11,214 milliseconds of the average latencies for results of queries 1 and 2, respectively. In terms of maximum latency, our solution took up to 47 and 49 seconds for executing queries 1 and 2, respectively. In the 1000th and above batches, the query execution took more than 10 seconds, compared to the previous batches. In the next section, we will discuss why the performance degradation occurs.

6 EXTENSIONS

In this section, we present three extended modules besides our solution. Likewise, all source codes can be accessed at Github[7].

[5]https://grpc.io/
[6]https://jupyter.org/
[7]https://github.com/developsu/debs2022gc

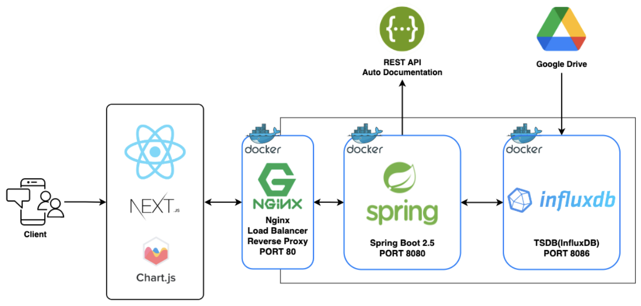

Figure 2: The Architecture of `StockInflux`

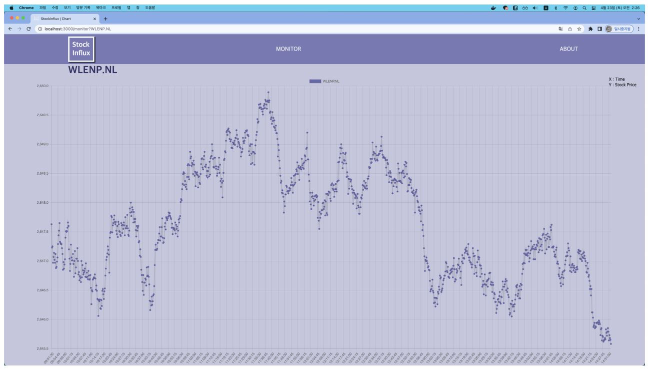

Figure 3: The screenshot of the real-time quotes monitor

6.1 `StockInflux`: Scalable and Fast WebApp

Figure 2 shows our architecture of `StockInflux` web application. The goal of `StockInflux` is to provide users with real-time decision-making in stock market analysis in terms of scalability and persistence. Currently, it is an early stage, so only the real-time quotes monitor is available now. Figure 3 shows the screenshot of the real-time quotes monitor. The real-time quotes monitor is a web-based dashboard that enables users to quickly view price trends at a glance depending on a selected symbol.

We expected `StockInflux` to be fast and scalable. For high-performance on client-side, we utilized React.js[8] for refreshing web-pages asynchronously, Chart.js[9] for drawing charts, and Next.js[10] for supporting both hybrid static and server rendering. To enhance the performance on server-side, we implemented `StockInflux` based on Spring Boot framework[11] supporting a multi-threading for computations of multiple technical indicators. Last but not least, we used an open-source time-series database InfluxDB[12]. This time-series database speeds up both retrieval and search of price events using the time series index.

[8]https://reactjs.org/
[9]https://www.chartjs.org/
[10]https://nextjs.org/
[11]https://spring.io/projects/spring-boot
[12]https://www.influxdata.com/

6.2 Jupyter Notebook Publishing

Our solution is easily accessible via the Jupyter Notebook. The users can test our solution quickly without professional skills. For instance, users can not only install our solution but also execute it within a few clicks at Google Colab[13]. The distribution of using the Jupyter Notebook encourages participations of users who want to add on various kinds of technical indicators. Besides, we expect it to be combined with existing Python-based deep learning libraries[7], e.g., stock prediction models[5].

6.3 Docker Integration

We ensured the reproducibility of our solution by creating a Docker image file. The Docker image file contained all the commands to build the execution environment where our solution runs. So the created image contained the gRPC library, PySpark, and so on. In addition, we included `pyngrok` library[14] to permit external access to the Apache Spark UI for performance monitoring. We also containerized all components of `StockInflux` and created a script file in form of docker-compose. For an efficient communication between containers, we configured a network bridge that operates independently to external network. Therefore, our Docker integration enables users to set up their execution environment identical to ours with a few commands.

7 CHALLENGES FACED

Despite many advantages obtained by using PySpark, our solution suffers from performance degradation for query execution. The current version of PySpark does not support the stateful operation that can maintain a summary of previous data[4]. So we convert the Spark dataframe into a Pandas dataframe to keep track of price events of the previous window. Such as conversion requires an expensive cost, e.g., about 800 milliseconds. The cost was a significant factor in lowering the latency. To minimize the overhead of the conversion, we used the conversion once for query execution.

There is a strategy to solve the problem. The strategy is to perform the stateful streaming aggregation with the `mapGroupWithState` operator. Still, the operator is not supported in the current version of PySpark, but is supported in pure Apache Spark.

8 CONCLUSION AND FUTURE WORK

To conclude, we have implemented our solution that provides real-time answers to two business queries by DEBS 2022 GC. In response to the functional requirements, the proposed solution has calculated the EMA indicators using the window, watermark, and aggregation functions. Moreover, we have identified breakout patterns and decided either buy or sell advice events. To achieve non-functional requirements of high accessibility and fast reproducibility, we have developed the PySpark-based solution and created a container image for running our solution. Besides, we have implemented StockInflux for analyzing the stock market streams via the real-time charting view.

In future research, we plan to develop stateful operations to improve the performance of our aggregation methods. In addition, the optimization of concurrent streaming queries[10] will be investigated so that a variety of technical indicators can be handled.

[13]https://colab.research.google.com

[14]https://pyngrok.readthedocs.io/en/latest/index.html

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea Grant funded by the Korean Government(No. 2021R1F1A1050937, No. 2021R1G1A1092248)

REFERENCES

[1] Sebastian Frischbier, Jawad Tahir, Christoph Doblander, Arne Hormann, Ruben Mayer, and Hans-Arno Jacobsen. 2022. The DEBS 2022 Grand Challenge: Detecting Trading Trends in Financial Tick Data. In The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27-June 30, 2022, Copenhagen. ACM, New York, NY, USA. https://doi.org/10.1145/3524860.3539645

[2] Chris Bradley and Peter Stumpner. 2021. The impact of COVID-19 on capital markets, one year in. McKinsey.

[3] Sebastian Frischbier, Jawad Tahir, Christoph Doblander, Arne Hormann, Ruben Mayer, and Hans-Arno Jacobsen. 2022. DEBS 2022 Grand Challenge Data Set: Trading Data. https://doi.org/10.5281/zenodo.6382482

[4] Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch, and Avigdor Gal. 2014. Scalable stateful stream processing for smart grids. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, 276-281.

[5] Vadlamani Ravi, Dadabada Pradeepkumar, and Kalyanmoy Deb. 2017. Financial time series prediction using hybrids of chaos theory, multi-layer perceptron and multi-objective evolutionary algorithms. Swarm and Evolutionary Computation 36, 136-149.

[6] Jules S. Damji, Brooke Wenig, Tathagata Das, and Denny Lee. 2020. Learning Spark. O'Reilly Media.

[7] Anand Gupta, Hardeo Kumar Thakur, Ritvik Shrivastava, Pulkit Kumar, and Sreyashi Nag. 2017. A big data analysis framework using apache spark and deep learning. In 2017 IEEE international conference on data mining workshops (ICDMW), 9-16.

[8] Pedro Silva, Wang Yue, and Tilmann Rabl. Incremental stream query analytics. 2020. In Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems(DEBS), 187-192.

[9] Vincenzo Gulisano, Daniel Jorde, Ruben Mayer, Hannaneh Najdataei, and Dimitris Palyvos-Giannas. 2020. The DEBS 2020 grand challenge. In Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, 183-186.

[10] Sejin Chun, Seungjun Yoon, Jooik Jung, and Kyong-Ho Lee, 2019. Planning Operators of Concurrent RDF Stream Processing Queries. International Journal of Web and Grid Services (IJWGS), Volume 15. No. 1.