A High-Performance Processing System for Monitoring Stock Market Data Stream

Kevin Li, Daniel Fernandez, David Klingler, Yuhan Gao, Jacob Rivera, Kia Teymourian
The University of Texas at Austin
Austin, TX, USA
{kevinali,daniel.fernandez,davidklingler,yg6952,jacobrivera}@utexas.edu,kiat@cs.utexas.edu

ABSTRACT

High-performance real-time monitoring of the stock market data stream is one of the challenging use cases of stream processing systems. By monitoring real-time stock price patterns via high-throughput event stream processing, real-time stock trading advice can be generated. In this paper, we describe our implementation of a system for the DEBS 2022 Grand Challenge to extract breakout patterns from real-time stock market data. Breakout patterns, based on exponential moving averages crossover events, indicate potential bullish or bearish trends, giving investors insight and the opportunity to buy or sell at optimal times. We report details of our high-performance implementation to extract such patterns from a real-time stream to generate stock trading advice event notification outputs. Furthermore, we report the architectural design of our system for parallel processing of data stream, which we implemented from scratch using Python and Java programming languages.

ACM Reference Format:
Kevin Li, Daniel Fernandez, David Klingler, Yuhan Gao, Jacob Rivera, Kia Teymourian. 2022. A High-Performance Processing System for Monitoring Stock Market Data Stream. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27-July 1, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524860.3539651

1 INTRODUCTION

The trading of stock shares can be triggered by high complex event patterns that are specified based on patterns in the event stream of the stock market. Complex patterns are are specified based on the history of the event stream, for example, by utilizing moving averages of stock prices and the trading volumes. The real-time extraction of such complex patterns to trigger buy or sell trading actions is the task of high-performance event stream processing systems.

The 2022 DEBS Grand Challenge [5] describes a system implementation based on two specific queries on the stock market event streams. The first query is defined to compute the Exponential Moving Average (EMA) with two different smoothing factors of 38 and 100. An exponential Moving Average is one of the moving averages and is defined as follows:

$$EMA_t = \begin{cases} Y_0 & t = 0 \\ \alpha Y_t + (1-\alpha) EMA_{t-1} & t > 0 \end{cases}$$

The coefficient α represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. For this challenge $\alpha = \frac{2}{1+j}$ where j is a smoothing factor with $j \in \{38, 100\}$. We use EMA_{38} to refer to the exponential moving average with a smoothing factor of 38 and EMA_{100} for a smoothing factor of 100.

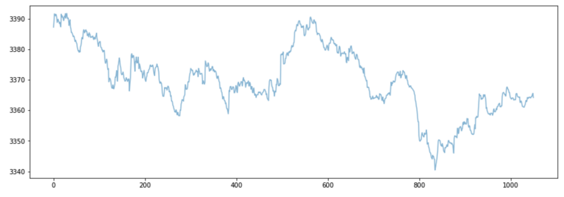

Figure 1: An Example of Stock Price Fluctuations Over Time

Figure 1 illustrates an example of stock market price changes over time. The graph shows 1000 events with different prices. Figure 2 depicts the values of the exponential moving average with smoothing factors of 38 and 100. We can observe in Figure 2 how the value grows exponentially and how the two EMA_{38} and EMA_{100} are different from each other.

Query 2 of the DEBS 2022 challenge [5] is specified based on the results of Query 1. Each time that EMA_{38} breaks out of the value of EMA_{100} a stock buy advice notification should be generated and if EMA_{100} goes under the EMA_{38} a stock sell advice. For a further detailed description of the DEBS 2022 Grand Challenge, we would refer the readers to [5].

Figure 3 visualizes the same graph as we have in Figure 2 by zooming into the graph (range 0 to 400 events on x-axis) to see the values and their differentiations. One important observation in Figures 3 and 2 is that EMA_{38} and EMA_{100} have a large difference in the first 200 events. The reason for this difference is that the exponential moving average is specified based on the history of events to increase the value exponentially and requires a warm-up phase. Based on this observation, one can improve the performance of the Query 2 by skipping the first 200 events because EMA_{38} and EMA_{100} still have a large difference.

The main system development challenge task is to design a system that can process the stream of events with high throughput and low latency. Many open source and commercial stream processing systems are developed that one can use to develop this

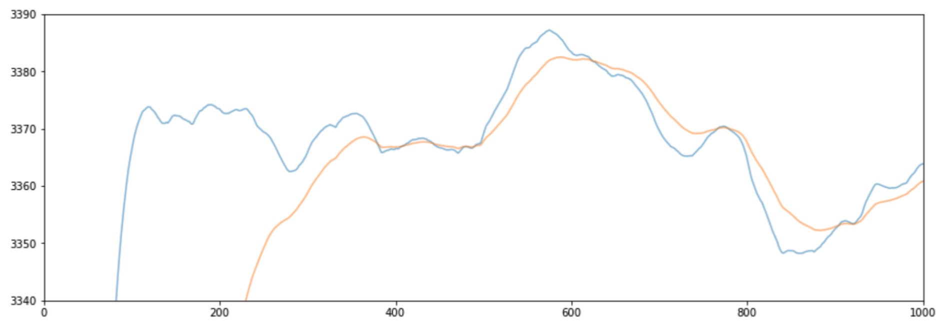

Figure 2: Example of Query 2 - Buy and Sell advice based on Breakout Patterns of EMA 38 and 100

challenge. Esper event stream processing system [3] is a system that can detect complex events based on pre-defined temporal logic patterns. In this task we do not have a complex pattern to extract and computations are basic computations of EMA 38 and 100, following with a check of the values for Query 2 to submit a sell or buy advice. Other systems like Apache Storm [1], Apache Spark Streaming [8] or Apache Flink streaming [2] have been developed to achieve high-scalability in data stream processing. Also, different benchmarks are developed [7] which compare these systems with each other regarding specific data stream processing tasks.

After considering all of the existing systems and their overhead trade-offs, we decided to implement the DEBS Grand Challenge from scratch, without using any of the existing systems like Apache Spark or Flink. Most of these systems have a different target like higher scalability than higher performance, so that they have a large start-up delay and additional cluster management costs.

Some examples of the performance overheads that systems like Apache Spark and Flink have are:

- Most of these systems are designed in a distributed cluster computing setting which includes cluster managements like communications between the master process/es and worker processes. Many processes are started as JobManager and TaskManager to coordinate different task executions.
- These systems are designed to optimize the trade-off between higher scalability and higher stream throughput/latency performance. The described DEBS 2022 challenge does not include a very large data stream that we would need to design a high-scalable data stream processing system. But our main system design goal is to achieve high throughput and low-latency. Systems like Spark or Flink include internal data processing pipelines including batch processing and blocking data exchanges.
- Such systems use special types of control events, like checkpoint barriers, watermarks signaling or iteration barriers to provide guarantee of a FIFO order of events, to generate snapshots of the data stream for example, or be able to do stream recovery [4]. In our system we can make sure that our stream processing is a stateful processing, and windows are correctly generated from the stream batches. And no additional internal batch generations are needed.
- The cluster stream processing systems have many configuration parameters which modify the performance stream processing. For example, the configuration parameters control the number of cluster executor processes and threads. With our implementation, we can better control the system parameters like number of processes or threads.
- The DEBS Challenge queries are simple patterns based that we can implement in a programming language in an optimize form, and there is no need to have a complex query processing and query optimization system.

We have implemented the Challenge queries as our initial prototype system in python and run multiple tests to check if the data streams are processed correctly, and the correct results are submitted to the DEBS 2022 evaluation system. Our Python implementation includes many iterations over the single event objects in a window stream. A vectorization of the many iterations could improve the performance by running mapping bulk operations to match the query 1 and query 2. Also, we experimented with running multi-threading to process the events.

In a subsequent step, we implemented the same architecture in Java to achieve higher processing performance (a better performance can be achieved by using a system programming language like C++ or Rust). Our implementation in Python and Java are available on Github[1]

The next subsequent sections describe details of our implementation. Section 2 describes our conceptual design for a stream processing system using multiple processing threads on a single machine with multiple CPU cores. Further we describe how the same architecture can be extended to process the data stream on a cluster of machines (multi-processing). Section 4 provides a brief description of important implementation details and Section 5 provides a brief overview of example experiment results.

2 STREAM PROCESSING ARCHITECTURE

Our data stream processing architecture is based on the event producers and consumers concept that can be used to distribute the processing tasks to multiple threads (on a single machine), or separated processes (distribution on many commodity machines), or a combination of both. Figure 4 depicts our system architecture. The producers read the data stream batches from the remote network

[1]https://github.com/kiat/debs2022 , last updated June, 2022

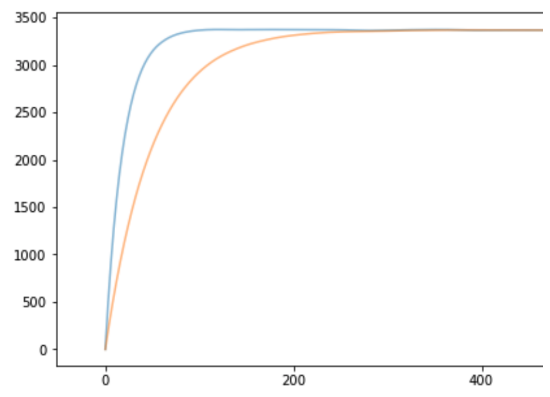

Figure 3: Exponential Moving Average of 38 and 100 for the first 400 events.

API (provided by DEBS 2022 Challenge [5]), and store them in a buffer queue in main memory. The buffer queue has a specific size limitation that can be configured based on the available machine's RAM. The producer will read the data batches and store them in the queue until the queue size reaches the specific size limit. Once the queue limit is reached, the producer thread or process is suspended until the queue is available again to receive data. We can use a lock-free queue to cache the incoming data stream before the main processing step.

The processing consumers access batches from the queue and process the events to generate Query 1 (values of EMA_{38} and EMA_{100}) and the subsequent Query 2 results. For the computation of *EMAs*, each consumer is required to know about the last *EMA* value of the previous batch for a given stock symbol. These consumer processes are the major query processing units that we call consumers for the sake of simplicity in this paper.

To make sure that there is no single bottleneck or synchronization lock point in this processing architecture, we made each of the consumers (processing units) be responsible for the monitoring and the computation of a sub-set of stocks symbol. Each consumer accesses the stream batches from the main memory cache and processes only the symbols (stock market data events) that this consumer is responsible for. This can be implemented by using hash functions on stock symbol and mapping it to the unique identification number of the consumer. In our implementation, the hashing and distribution of the symbols to the consumers is processed in parallel by the producers.

Each consumer has a hash table that it uses to memoize the previous values of the moving averages for different stock symbols. In order to optimize for memory and time complexity, the only data stored per symbol include: the latest event (closing price) for the current window, the EMA_{38} and EMA_{100} of the previous window, and the last three crossover events. Each consumer has to access events of each batch and compute those stock symbols that the consumer is responsible for. With this approach, each consumer can work independently of any other consumer in a functional form. Access to each data batch from the queue should be implemented in an efficient manner so that each consumer does not read the entire batch to filter out the stock symbols it is responsible for.

Figures 5 and 6 illustrate the system design if we have included an event dispatcher and an event submitter. We do not have a dispatcher and submitter because of efficiency reasons and because of the correctness of query processing. If we dispatch the event batches as shown in Figure 5, for example, by round-robin batch distribution, then each consumer has to work on all stock symbols or a subset of them.

If consumers work on all of the events, then each consumer is required to know the previous moving averages from the immediate past batch to compute the new *EMAs* for the current data batch; implying a strict dependence on the preceding consumer before the current consumer can proceed. This issue would cause incorrect processing, because a consumer should work in a parallel distributed design, without any dependencies on each other's results. In the former case, if the consumers work on a subset of stocks, we would need to pass every batch to all of the consumers, and it is better to do this task in an integrated form with reading from the queue.

Figure 7 illustrates how sequential data stream processing can work to process the batches in multiple serial event consumers. Each processing unit will pick up and process a subset of stock market events. In a single machine setup, the overhead of passing the event batches to the next consumer is very small, but in a distributed setting, the overhead of passing data batches to the next consumer over the network is very high because of data serialization costs. Comparing the architecture of Figure 7 and 5, we preferred to implement the system parallel format shown in Figure 5.

If the stream of data is terminated, the stream processing system will terminate, as producers cannot generate more event batches and there are no event batches in the buffer queue.

3 DATA STREAM WINDOWING

Based on DEBS Challenge [5] description, our system has to compute on events grouped into windows of 5 minute length, tumbling, non-overlapping windows. The system clock timer starts when it receives the first event and restarts every 5 minutes. As a preprocessing step, our system reads the events into main memory and generates data batches, which are then cached. The data batches are thereafter passed to the next processing step for the subsequent computation.

4 IMPLEMENTATION DETAILS

The described architecture can be implemented using multiple threads on a single machine with multiple CPUs, or it can be distributed over a cluster of machines, each with multiple CPU cores. Our first rapid alpha implementation of the system was a multi-threaded Python implementation to make sure that we understood the challenge tasks and are able to process the queries in the correct form. Later, we implemented the whole system again in Java. Both of these implementations are open sourced on Github[2].

Our Java implementation includes two simple threads, one is the event producer (or event batch Reader Thread that communicates over the DEBS Challenge API). The Reader reads the stream of event

[2]https://github.com/kiat/debs2022 , last update June, 2022

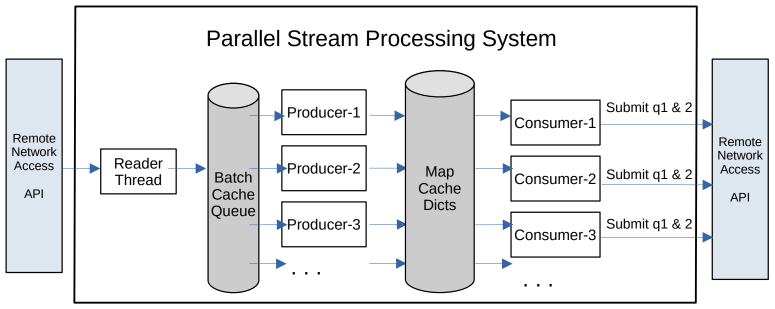

Figure 4: Parallel Processing of Events By a Set of Producers and Consumers without a Data Dispatcher.

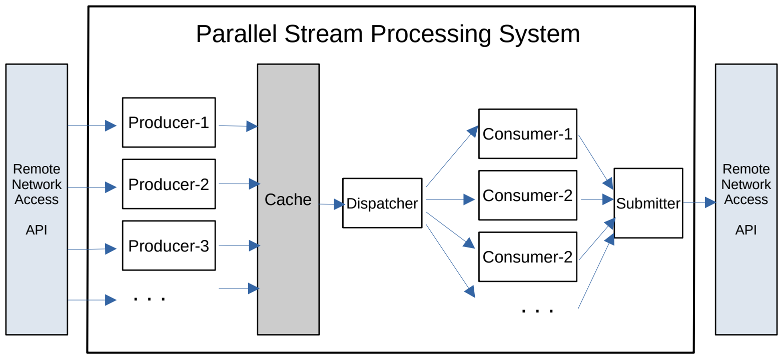

Figure 5: Event Dispatcher and Submitter Architecture

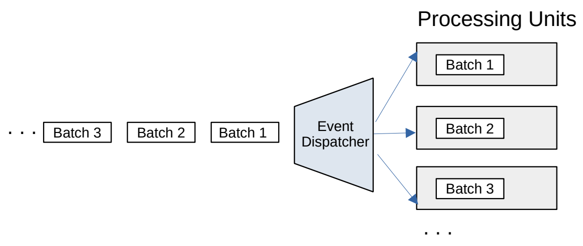

Figure 6: An Event Data Dispatcher

batches, creates and adds event batch windows (5 min. windows) to an intermediate cache. In parallel, another consumer thread processes the cached event windows data for the results of query 1 and query 2. We have integrated processing the query 2 with the Query 1 task within the consumer thread. The caches are java lists of batches and a java HashMap that caches event windows for each stock symbols. To achieve better performance we pre-allocate both of these caches Java ArrayList and HashMap with a specific size. The two threads are synchronized over a simple lock mechanism and thread notification. The results of both queries are submitted by the consumer thread to the DEBS Challenge API.

4.1 Distributed Cluster Implementation.

The architecture in Figure 5 can be implemented on a cluster of machines. The cache to store the stream batches can be implemented using shared memory on a single machine or using a service bus like Apache Kafka[3] or Apache Camel[4] to store the events and let multiple consumer clients access the streaming data batches.

Each consumer can subscribe to the event bus (e.g., Kafka) and receive only those stock prices that the specific consumer is responsible for. In this way, there are no dependencies between the consumers and there is no need to pass over data batches to the next consumer; also, each consumer can submit the query result to the output sink.

One other implementation detail is to use a high-efficient data model and serialization framework for the data transmission between the processing nodes. Studies have shown that choosing the correct data model and serialization have a huge impact on data processing performance [6].

We suggest implementing the proposed architectural solution in a system programming language, like C++ or Rust, because of the statically typed variables and manually managed memory without an overhead of automated garbage collection. As described in Section 2, the proposed architecture can be implemented using multiple threads on a single machine with multiple CPUs or distributed over

[3]https://kafka.apache.org/
[4]https://camel.apache.org/

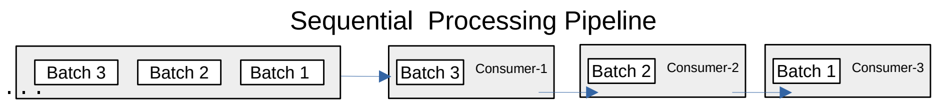

Figure 7: A Sequential Processing Pipeline. Each consumer reads a batch and passes to the next consumer.

a cluster of machines each with multiple CPU cores. Our first rapid alpha implementation of the system is in multi-threaded Python code to make sure that we understand the challenge tasks and be able to process the queries in the correct form.

5 EXPERIMENTS AND EVALUATION

We have run multiple experiments to be able to select the right stream processing architecture for this task. Our experiments include a set of experiments with varying numbers of event producers and consumer threads, as well as other experiments with different numbers of threads and queue buffer sizes.

The main computation for Query 1 and Query 2 is a very small computation to get the last prices from a dictionary of stock symbols and then compare the two values of EMA_{38} and EMA_{100} to identify breakouts. We have profiled our implementation and confirmed that the main implemented system is an I/O bounded application, rather than a CPU bounded application, because of the quick computation for Query 1 and 2. The most time is invested to get the data from the network (which is improved by having multiple producers), followed by organizing the stock price dictionaries and iterating over the single events in an event batch.

By using vectorized bulk operations, we can avoid the linear time of iterating over the batch of events. This is implemented by using vectorization within each of our multi-threaded consumers.

We have experimented with different numbers of producers and consumers. Figure 8 depicts Query 2 throughput results for different numbers of producers and consumers using small data batch of size 1k. These results are produced by using the benchmark system provided by the DEBS 2022 Grand Challenge[5].

Our Java implementation can achieve higher performance because of some language features (Like static variable types and improved iterations over event batches). According to the DEBS 2022 evaluation [5] server, our Java implementation can achieve a latency of 19.99 seconds and a 88.21 batch per seconds with an event batch size of 10k (events arriving in batches of 10k).

6 CONCLUSION

In this paper, we described our different architectures for a stock market data stream processing to implement the DEBS 2022 Grand Challenge. Our implementation and evaluation are limited due to the time constraints we had for this work. One potential improvement would be to implement it using a system programming language. Also, it would be interesting to evaluate how a from-scratch-implemented system would perform in this case compared to some large-scale cluster-based stream processing systems like Apache Storm or Flink.

[5]https://github.com/jawadtahir/CHALLENGER last update June, 2022

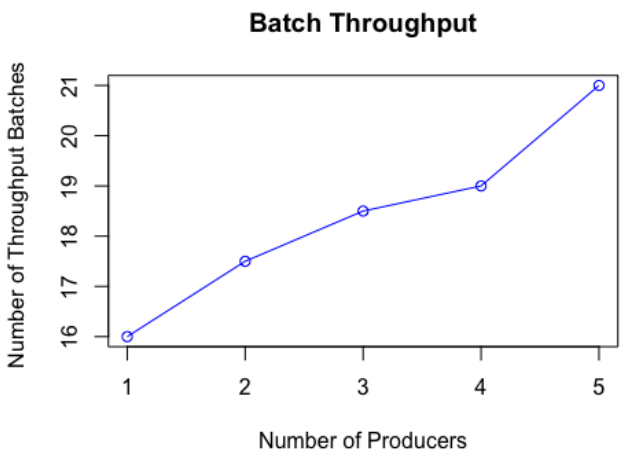

Figure 8: Query 2 throughput for different number of Producers. (Batch Size 1k - Multi-threaded Python implementation)

7 ACKNOWLEDGMENT

We would like to thank the organizers of the DEBS 2022 Grand Challenge for developing this implementation challenge.

REFERENCES

[1] Adnan Akbar, George Kousiouris, Haris Pervaiz, Juan Sancho, Paula Ta-Shma, Francois Carrez, and Klaus Moessner. 2018. Real-Time Probabilistic Data Fusion for Large-Scale IoT Applications. *IEEE Access* 6 (2018), 10015–10027. https://doi.org/10.1109/ACCESS.2018.2804623

[2] Alexander Alexandrov et al. 2014. The Stratosphere platform for big data analytics. *Proceedings of the VLDB Endowment* 23, 6 (2014).

[3] Thomas Bernhardt and Alexandre Vasseur. 2007. Esper: Event Stream Processing and Correlation. Online article. http://onjava.com/pub/a/onjava/2007/03/07/esper-event-stream-processing-and-correlation.html

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[5] 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-July 1, 2022, Copenhagen. ACM, New York, NY, USA.*

[6] Sourav Sikdar, Kia Teymourian, and Chris Jermaine. 2017. An experimental comparison of complex object implementations for big data systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017.* ACM, 432–444. https://doi.org/10.1145/3127479.3129248

[7] Lemi Isaac Yoseke Laku, Alaelddin Fuad Yousif Mohammed, Fawaz AL-Hazemi, and Chan-Hyun Youn. 2019. Performance Evaluation of Apache Storm With Writing Scripts. In *2019 21st International Conference on Advanced Communication Technology (ICACT).* 728–733. https://doi.org/10.23919/ICACT.2019.8701904

[8] Matei Zaharia et al. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud.* 1–10.