Detecting Trading Trends in Streaming Financial Data using Apache Flink

Emmanouil Kritharakis Shengyao Luo Vivek Unnikrishnan Karan Vombatkere
{ekrithar, jaxluo, viveku, kvombat}@bu.edu
Boston University, Boston, USA

ABSTRACT

Modern financial analytics rely on high-volume streams of event notifications that report live market fluctuations based on supply and demand. Accurately identifying trends or breakout patterns based on the Exponential Moving Average (EMA) in the development of an instrument's price early on is an important challenge, so as to buy while the price is low and sell before a downtrend begins.

This paper aims to solve the above challenge with a distributed, event-streaming solution built using Apache Flink. We present and implement a solution that leverages customized window operators to calculate the EMA and find breakout patterns, using event generation parallelism to facilitate the rapid processing of the input stream uses sinks to collect and output results, and scales easily on a distributed Flink cluster. We empirically test our design on metrics specified by the benchmarking platform for the DEBS 2022 Grand Challenge and observe a throughput of 45 batches per second and an average latency of 120 ms.

CCS CONCEPTS

• **Information systems → Data stream mining**; • **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

Apache Flink, Stream Processing, Financial Data

ACM Reference Format:
Emmanouil Kritharakis Shengyao Luo Vivek Unnikrishnan Karan Vombatkere. 2022. Detecting Trading Trends in Streaming Financial Data using Apache Flink. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3524860.3539647

1 INTRODUCTION

The DEBS Grand Challenge is an annual competition in which participants compete with the goal of building scalable, distributed and event-based systems to solve real-world problems. This year's challenge involves analyzing streaming financial data in the form of event notifications containing market data about trades. By tracking the Exponential Moving Average (EMA) - a quantitative indicator to identify trends in the market - the task is to detect crossover events or breakout patterns, and automate Buy and Sell advice for instruments in a streaming setting. Breakout patterns describe meaningful changes in the development of instruments that can indicate the start of a market trend. There are two main breakout patterns we are interested in: Bullish breakouts which correspond to an upcoming rise in price, and Bearish breakouts that correspond to a potential downtrend.

This paper uses Apache Flink [5, 6], an open-source, unified stream-processing and batch-processing framework. Our approach first identifies trends in the EMA for individual equities using event aggregation over tumbling window operators, and then uses the calculated EMA trends to predict buy/sell advises upon detecting specific breakout patterns. The code is publicly available on Github [1].

The main objective is to compute two separate, sequential queries - Query 1 and Query 2 - that are computed over 5-minute tumbling windows. For each window w_i, Query 1 computes two different EMA values, $EMA^{38}_{w_i}$, $EMA^{100}_{w_i}$ (which correspond to smoothing factors of 38 and 100 respectively) which serve as quantitative indicators of trends in the market data. Query 1 passes this EMA data stream keyed by symbol to Query 2, which then tracks these two different exponential moving average values for each window, for each symbol and detects the two breakout patterns.

The data provided by the competition is a stream of event notifications that is pre-filtered to contain pricing information per equity symbol for each instrument [8]. All events notifications are time-stamped with a global CEST timestamp, and all definitions considered are per symbol. As per the requirements of the challenge all calculations are based on events grouped into tumbling windows of 5 minutes' length, and we calculate the required EMA's for all symbols but only generate output for a specific set of symbols that have been subscribed to.

Our solution leverages Flink's streaming dataflow architecture, and successfully processes tens of millions of streaming financial event notifications in just a few minutes. Our design runs on a distributed Flink cluster and allows for easy monitoring of real-time metrics on a dashboard.

2 BACKGROUND

2.1 Apache Flink

Apache Flink is an open-source stream processing framework that allows for efficient computation of real-time events. It offers reliable and stable performance, fast data processing and easy-to-use APIs. Flink uses *operators*, that are essentially layers of processing logic that sequentially implement operations on a stream of data.

Flink operators allow for an arbitrary number of parallel instances, allowing for a scalable, distributed design. This architecture

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539647

requires a specific partitioning of data after a transformation, with the most common ones being *Key By (Hash)* and *Rebalance*. *Key By* allows for a certain chunk of the transformed data to be hashed and sent downstream to a specific instance. *Rebalance* uses the round-robin algorithm to distribute data equally among partitions.

In our solution, the data is partitioned using *Key By* and *Rebalance* since we want each trading symbol to be sent to the same *operator*. We use *window* operators to implement Query 1 and Query 2. *Windows* group events into buckets and enable bounded processing of unbounded input streams [9]. This allows for computation to happen as if the stream was segmented into separate pieces of finite data while ingesting a continuous data stream. The properties of the custom *windows* are then manually defined using custom *triggers* and *evictors* using the Flink framework.

The *trigger* is a piece of logic that determines when to invoke the *window function*. It can be configured to be a function of time, number of stream elements, etc. The *evictor* of the *window* decides when to remove elements from a *window*. The *process window function* is the main logic of the *window*, and performs the actual operations on chunks of the data stream as per the rules defined by the *trigger* and *evictor*. We evaluate both a custom window architecture and Flink's tumbling windows in our solution.

In our design we also use *watermarks* and *buffer timeouts*. *Watermarks* tell operators that no elements with a timestamp older or equal to the watermark timestamp should arrive at the operator. They are emitted at the sources and propagate through the operators of the topology. *Buffer timeouts* allow us to fine tune our solution and improve latency, by ensuring that at any stage no window waits too long for data. By manipulating the watermark strategy and buffer timeout we achieve a high degree of control over the datastream's flow through the window operators.

2.2 Exponential Moving Average (EMA)

The exponential moving average (EMA) is a common indicator that continuously tracks pricing. It is a type of weighted moving average that gives more weight to recent price data. In the challenge, we measure non-overlapping five-minute windows by the formula below, where w_i corresponds to the time window, $Close_{w_i} \cdot$ is the last price event observed within window i, and j is the smoothing factor for the EMA.

$$EMA^{j}_{w_i} = \left[Close_{w_i} \cdot \left(\frac{2}{1+j}\right)\right] + EMA^{j}_{w_{i-1}}\left[1 - \left(\frac{2}{1+j}\right)\right]$$

In Query 1, we calculate two EMA values with smoothing factor 38 and 100 for each window. The initial EMA values are set to 0 for both smoothing factors.

2.3 Breakout patterns

In trading markets, breakout patterns represent meaningful changes in the development of a price that indicates a beginning of rising/-falling trend. A bullish breakout occurs at the start of a price uptrend, and a bearish breakout occurs at the start of a price downtrend. Estimating these breakout patterns is an important task, since based on these trends traders can choose to buy (bullish breakout) or sell (bearish breakout) financial instruments to maximize revenue.

We use the following definitions of the two breakout patterns as defined by the DEBS 2022 challenge:

$$\text{Bullish Breakout} = (EMA^{38}_{w_i} > EMA^{100}_{w_i}) \wedge (EMA^{38}_{w_{i-1}} \leq EMA^{100}_{w_{i-1}})$$

$$\text{Bearish Breakout} = (EMA^{38}_{w_i} < EMA^{100}_{w_i}) \wedge (EMA^{38}_{w_{i-1}} \geq EMA^{100}_{w_{i-1}})$$

3 DESIGN & IMPLEMENTATION

In this section we discuss how the solution was designed and implemented. We will go through each operator as well as the visualization aspect of the solution shown in Figure 1. We focused on two aspects during the design of the operators: The first is that every operator should have exactly one responsibility so that it can be easily decoupled from other operators. The second is that the design should be scalable so that when faced with higher load, horizontal scaling of the number of task manager should subsequently reduce back pressure and improve performance.

3.1 Event generator

The client provided by the DEBS Grand Challenge provides with a stream of events in the form of batches. These batches are of a fixed size and contains the list of events and the specific symbols that need to be reported back to the benchmark. The initial implementation included a generic loop logic which continuously gets the next batch and sends the mapped custom `Stock Measurement` event to the next operator. The limitation of this approach was that we were processing one batch at a time leading to decreased throughput. In order to tackle this problem, we decided to design the operator to emit batches instead of the list of Stock measurement objects. As seen in the code snippet below, the source client defined by the `challengeClient` variable gets the next batch using the benchmark variable defined by `newBenchmark` which is then converted to an `IncomingBatch` object and sent to the next stream

```
while (true) {
 Batch batch = challengeClient.nextBatch(newBenchmark);
 ctx.collect(new IncomingBatch(batch));
}
```

Each batch is then processed by the Event Generator operator to map the events in the batch to the custom `Stock Measurement` objects. By controlling the parallelism of the Event Generator operator, we can control the input batch throughput. In the code snippet below, Flink's `FlatMap` operator is used with custom Flat map function defined by the `EventGenerator` class.

```
incomingBatchDataStream
  .flatMap(new EventGenerator(benchmark))
  .name("Event Generator");
```

3.2 Query 1

The stream of `Stock Measurement` events are processed by the Query 1 operator. The Query 1 operator is responsible for the calculation of the EMA values per symbol. Currently, as per requirement, the EMA values are calculated for the smoothing factors of 38 and 100 using the formula defined in 2.2 section. The initial design used the default Tumbling window operator provided by Flink with the custom watermark strategy that was defined. As

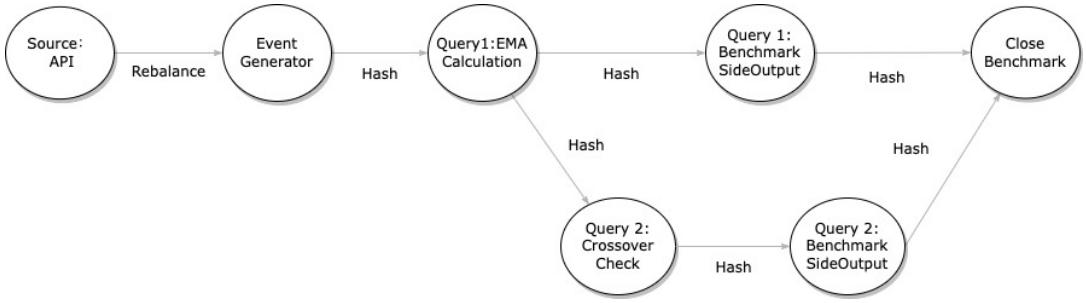

Figure 1: Design Data Flow Architecture

defined in Section 2.1, the watermark strategy ensures the correct ordering of events. Using the event timestamp as well as the source of the stock measurement, the maximum timestamp out of all the sources was emitted as the watermark. By default, Flink emits the watermark every 200 ms. But we noticed that due to the window being triggered only at the end of the configured window time, the latency of the application is high. Rather the ideal solution would be to process each element and check if it is the last event of the batch and then pass it to the respective window. Thus we created a custom window operator which provides a function for processing individual events as well as a function which executes when the configured time is reached. This enabled the application to send results for a symbol in a batch as soon as its processing is ended rather than waiting for the end of the window, which reduced the latency. The details of the latency decrease can be found in Section 5. The code snippet below shows the Query 1 operator with a custom watermark strategy keyed by the symbol of the incoming stock event.

```
incomingBatchDataStream
  measurements
  .assignTimestampsAndWatermarks(
    WatermarkStrategy
    .forGenerator(context ->
      new StockMeasurementWatermarkGenerator())
    .withTimestampAssigner(new
      StockMeasurementWaterMark()))
  .keyBy(StockMeasurement::getSymbol);
```

The Query 1 operator also uses Value State, Flink's custom object to store and retrieve values scoped to the key of the window, to store the intermediate EMA values. The operator sends messages to 2 operators down the chain The first one is the Q1 Side output operator which deals with sending the benchmarks for each batch of symbols. The second is the Query 2 operator which will use the EMA values to check Bullish and Bearish patterns.

3.3 Query 2

The stream of updated values coming from Query 1 in the form of `EMAStream` objects. Similar to query 1, the stream is keyed by symbol and every element is processed separately. For each element, the advice for the EMA values is calculated by using the formula defined in Section 2.2. If a valid advice is returned, the crossover event is created and updated to the value state of the symbol. Similar to query 1, if the event signifies the end of the batch, then a benchmark event is sent to the Q2 Side output operator. The code snippet below shows the Query 2 operator keyed by the symbol of the incoming event and processed using the custom process function defined by the `FindBreakPattern` class.

```
emaDataStream.keyBy(EmaStream::getSymbol)
  .keyStream.process(new FindBreakPattern())
  .name("Query 2: Crossover Check");
```

3.4 Sink

In our design we have two sinks indicated Q1 Side output and Q2 Side output. Side output is a special Flink operator that will enable the user to send a separate stream of events apart from the main output stream. This enabled us to send benchmark events to both the side outputs whenever we find that the batch has ended during the event processing in Query 1 or Query 2 operators. The initial design included directly reporting these benchmarks per symbol but due to requirements we changed the reporting to per batch. Due to this a custom counting window per batch sequence identifier needed to be created. This counting window kept track of the symbols coming per batch and when it hit the total number of symbols in the batch, the entire batch of benchmark were sent using a single API call. The advantages of this approach was reduced network calls and accurate reporting of query data as well. The snippet below shows the 2 sinks that are defined. Both the sinks are keyed by the batch Id of the result. The Global windows created are triggered using the custom triggers defined by `ResultQ1Trigger` and `ResultQ2Trigger` for each of the sinks. The triggered windows are processed by the custom window functions defined by `ResultQ1ProcessWindow` and `ResultQ2ProcessWindow` respectively.

```
emaDataStream.getSideOutput(benchMarkQ1)
 .keyBy(ResultQ1Wrapper::getBatchSeqId)
 .window(GlobalWindows.create())
 .trigger(new ResultQ1Trigger())
 .process(new ResultQ1ProcessWindow(benchmark))
 .name("Q1 Benchmark SideOutput");

q2OutputStream.getSideOutput(benchMarkQ2)
 .keyBy(ResultQ2Wrapper::getBatchSeqId)
 .window(GlobalWindows.create())
 .trigger(new ResultQ2Trigger())
 .process(new ResultQ2ProcessWindow(benchmark))
 .name("Q2 Benchmark SideOutput");
```

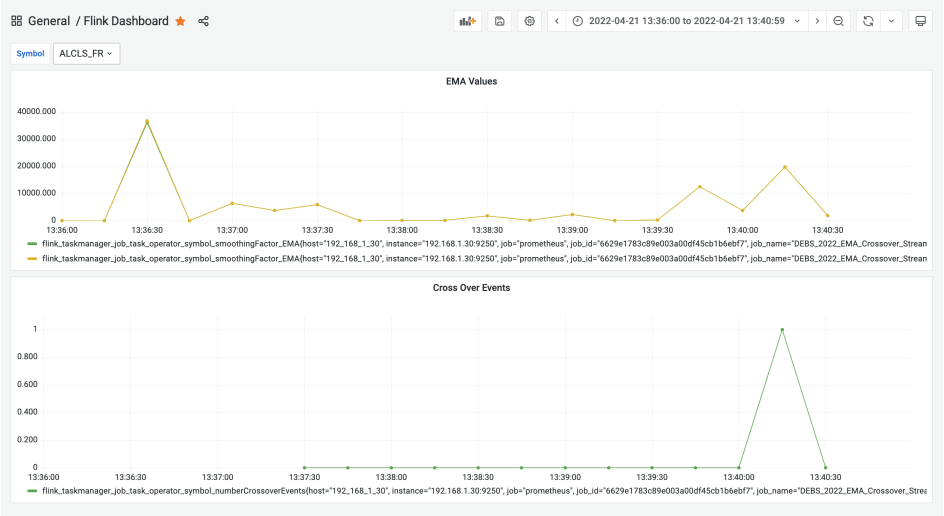

Figure 2: The Grafana dashboard showing two plots for the selected symbol ALCLS_FR. The top plot shows the EMA 38 and 100 values while the below plot shows the number of cross over events for the symbol. The legend below each plot shows the metrics from which the plot has been calculated.

3.5 Distributed Flink configuration

In order to ensure scalability of the solution, the operators needed to be designed so that it can run on a distributed Flink cluster. There were a few challenges faced for achieving this. The first is that few of the classes provided for benchmarking are not serializable. The second was that intermediate event size needed to be controlled since larger events lead to increased network latency. For the first problem, we decided to make sure every operator had its own copy of the benchmarking client by initializing the same during the creation of the operator. This ensured that the results published were accurate and every operator can function individually without any dependencies. For the second problem, we have gone over it in detail in Section 5. We also ran multiple performance tests by tuning Flink parameters including parallelism, number of task slots and buffer timeout. The results of these tests are explained in detail in Section 4.

3.6 Visualization

For the visualization, we decided to use Flink's inbuilt metrics registry to store the different values we are calculating during the job. There are two metrics that are being recorded: EMA values for each symbol and smoothing factor and the number of crossover events for each symbol. The reporting of metrics can be enabled or disabled based on the visualization flag in the code. The Flink's metrics system is an inbuilt registry which stores existing metrics as well as enables addition of custom metrics. We have registered 3 gauges currently in Flink's metrics registry: EMA value with smoothing factor of 38 for each symbol, EMA value with smoothing factor of 100 for each symbol and the number of crossover events for each symbol. Separately Prometheus [4] and Grafana [3] were setup in the master node. Prometheus is a time series database which is used for storing and querying metrics, while Grafana is a visualization tool that can be linked to multiple data sources to create plots based on difference metrics. The metrics registry is connected to Prometheus so that we can store the time series data for every metric in Flink. Grafana is then used with Prometheus as its data source so that results of various queries can be shown in a dashboard as in Figure 2 The symbol can be selected from the drop down provided in the dashboard and the time span can be adjusted from the top right menu. Based on the selected symbol, Grafana queries the metrics from Prometheus and displays the results.

4 RESULTS

We deploy our experiments on 3 virtual machines (VMs) provided by the challenge committee. Each VM consists of 8GB ram memory, 4 core Intel Core Processor (Haswell, no TSX, IBRS) CPUs and 16GB virtual storage system. Ubuntu 20.04, Java 11, Maven 3.8.5 and Apache Flink 1.14.3 were installed on all 3 VMs, thus giving us a total of 12 CPU cores.

We evaluate our streaming architecture in terms of *throughput* and *latency*. *Throughput* is defined as the average number of batch responses per second from queries 1 and 2 to the GRPC server. We measure *latency* per batch as the average time between a request for a new batch and the submission of results to the GRPC server for both queries.

We evaluate the performance of our streaming architecture using the benchmarking platform provided by the DEBS 2022 challenge committee. In particular, we evaluate our custom window architecture, and compare it with Flink's tumbling window as well as fine tune the buffer timeout hyperparameter. We run both our custom window architecture and the tumbling window solution on the provided *evaluation* dataset which consists of almost 60 million event notifications [7]. We run each experiment 3 times and report average results.

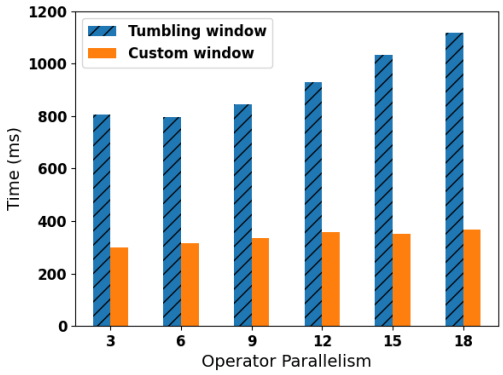

Figure 3: End to End Latency per Batch Comparison Between Custom and Tumbling Window

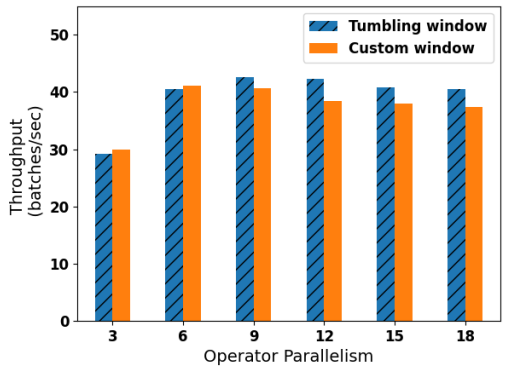

Figure 4: End to End Throughput per Batch Comparison Between Custom and Tumbling Window

4.1 Effect of parallelism on throughput and latency

Identifying the parallelism of a distributed approach which maximizes the latency and throughput metrics plays an important role to the end-to-end solution. Towards this direction, we investigate how these measurements will be affected in our custom window approach over multiple VMs configuring the total number of task slots per virtual machine. In particular, we progressively increase the number of task slots for each virtual machine and trace the latency and throughput results. In this experiment, we keep the parallelism of source and sink to 1 and set all the parallelism of all the other operators to an increasing value from 3 to 18. In addition, we compare the custom window throughput and latency with the default tumbling window to measure metrics differences between those two attempts. As it is easily observed by Figures 3 and 4, the throughput and latency of our solution are optimized when each working operator of the pipeline has a parallelism of 6. This result makes sense intuitively, since having 4 CPUs per VM the total number of threads we can adjust is 12 (3 VMs in total). The individual operators except source and sink are 5, so if each one is parallelized by 12 we notice CPU congestion and as an outcome, the throughput results start to decrease for operator parallelism options above 12. Following the same logic, we notice that operator parallelism of 6 presents the best overall results in terms of latency and throughput. We also observe that while both solutions provide similar results on throughput, the custom window solution gives us a significant reduction in latency, due to the more sophisticated trigger-eviction scheme.

4.2 Exploring the effect of buffer timeout

Having decided which operator parallelism improves the most both latency and throughput we moved towards fine tuning hyperparameters of Flink framework. One of the most impactful configuration was buffer timeout. Therefore, we vary the buffer timeout hyperparameter programmatically within Flink on our custom window solution, in an attempt to further improve our solution. Figure 5 shows that reducing the buffer timeout further helps decrease our latency by up to 2x. Flink uses a default buffer timeout of 100ms, but we customize our solution to have a buffer timeout of 10ms.

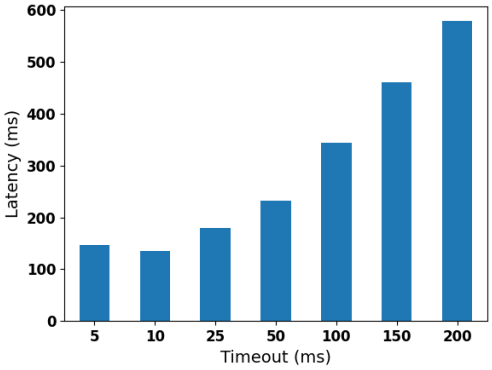

Figure 5: Fine tuning buffer timeout hyperparameter reduces the total latency up to 2x from its default value (100ms to 10ms comparison).

5 LESSONS LEARNED

In this section, we would like to share how we addressed problems faced towards the solution of DEBS challenge. We believe that Flink developers might find them useful working on future projects embedded with GRPC servers.

Custom Windowing decreases latency up to 3x. The initial implementation used Flink's default windowing function. Since the window gets triggered only when the event exceeds the watermark, the batches that were completed within the window are closed only when the window process function is triggered. This led to higher latency numbers in all the experiments. By implementing a custom window function, every event that comes in was checked individually and the batch was closed as soon as an end of batch event was detected, thus lowering end-to-end latency of the application as shown in Figure 3.

Reducing size of intermediate objects increases throughput and decreases latency by 10x. As stated in Flink's "Data Types and Serialization" documentation page [2], if the object does not follow the prerequisites of being serializable, the object is treated as a Generic type object and serialized using the Kyro Serializer in Flink. This led to a larger object size lead to much higher transfer times of the events between operators. By correcting the classes used and making it simpler to serialize, we reduced the overall size of intermediate objects by more than 90% leading to a dramatic increase in throughput and decrease in latency.

Event generator parallelism boosts throughput up to 3x. Our first design architectures were based on a single event generator connecting the GRPC client with the source of the Flink framework. In particular, we passed the events of an inspected batch sequentially to Flink's source operator. It was easily noticeable that although the latency metrics were close to 1 minute (no other microbenchmarks configured) the throughput metrics had an upper threshold of 15 batches per second. To address this problem, we parallelize the event generators by passing the whole batch to the Flink framework and delegate the work with a Flatmap operator assigning one batch per worker. The final outcome of our design choice leads to a 3x throughput gain.

Buffer timeout reduction decreases latency up to 2x. Fine-tuning the microbenchmarks of the Flink framework lead to a 2 times latency improvement. As buffer timeout time, Flink developers define the maximum time frequency in milliseconds for the flushing of the output buffers to the next operator. Decreasing this hyperparameter, we pass the results at a faster rate to the next operator. As depicted in Figure 5, the progressive reduction of the initial default value of 100ms up to 10ms lead to a halved latency time. It should be mentioned that a further decrease of buffer timeout value had a negative result in increasing the latency metric. Therefore, we conclude that decreasing the timeout value further than a threshold will not benefit performance.

6 CONCLUSION

In this paper, we present the design and implementation of a customized, streaming Apache Flink application that detects trading trends in financial data. As outlined by the 2022 DEBS challenge, it first calculates the relevant EMA values, then identifies breakout patterns, and finally submits financial advice (Buy/Sell) for each symbol that has been subscribed to. Our solution includes a metrics visualization element, runs on a distributed Flink cluster and was tested for its throughput and latency on large amounts of streaming data in real-time.

ACKNOWLEDGMENTS

We would like to thank professor Vasiliki Kalavri for her mentorship and support throughout the project. Furthermore, Emmanouil Kritharakis is supported by the Onassis Scholarship [Scholarship ID: F ZR 030/1-2021/2022].

REFERENCES

[1] 2022. *DEBS-22 Grand Challenge, Group-11 GitHub Source code.* https://github.com/kvombatkere/DEBS22-Group11
[2] 2022. *Flink framework: Data Types & Serialization.* https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/fault-tolerance/serialization/types_serialization/
[3] 2022. *Grafana: The open observability platform, Documentation.* https://grafana.com/docs
[4] 2022. *Prometheus: Monitoring System & Time series Database, Documentation.* https://prometheus.io/docs
[5] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.
[6] 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).
[7] 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
[8] 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 *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS '22).* Association for Computing Machinery, New York, NY, USA.
[9] F. Hueske and V. Kalavri. 2019. *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications.* O'Reilly Media, Incorporated. https://books.google.com/books?id=64GHAQAACAAJ