# Real-time Analysis of Market Data Leveraging Apache Flink

Cecilia Calavaro
c.calavaro@yahoo.it
University of Rome Tor Vergata
Rome, Italy

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

Valeria Cardellini
cardellini@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

## ABSTRACT

In this paper, we present a solution to the DEBS 2022 Grand Challenge (GC). According to the GC requirements, the proposed software continuously observes notifications about financial instruments being traded, aiming to timely detect breakout patterns. Our solution leverages Apache Flink, an open-source, scalable stream processing platform, which allows us to process incoming data streams with low latency and exploit the parallelism offered by the underlying computing infrastructure.

## CCS CONCEPTS

• **Information systems → Data analytics**; **Stream management**.

## KEYWORDS

Financial trading data, data analytics, stream processing

**ACM Reference Format:**
Cecilia Calavaro, Gabriele Russo Russo, and Valeria Cardellini. 2022. Real-time Analysis of Market Data Leveraging 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, 4 pages. https://doi.org/10.1145/3524860.3539650

## 1 INTRODUCTION

Financial markets generate huge amounts of real-time data associated with various kinds of events (e.g., bids, trades), which are commonly referred to as *market data*. For instance, Infront reported 24 billions of daily event notifications processed on average in 2021, with an increase of 33% with respect to 2019 [6]. Managing such an enormous volume of data is a technically challenging task of paramount importance for financial traders and analysts, who must continuously observe and react to complex market dynamics (e.g., to identify investment opportunities).

The *DEBS 2022 Grand Challenge* (GC) [8] revolves around the analysis of financial market data in real time. In particular, the goal of the GC is to efficiently compute specific trend indicators and detect patterns resembling those used by traders to decide on buying or selling [12], thus providing a tool to support traders in their decisions. For this purpose, the GC relies on a data set [7] of real-world tick data provided by Infront Financial Technology, which comprises information about more than 5,000 financial instruments being traded on three major exchanges over the course of a week.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*DEBS '22, June 27-July 1, 2022, Copenhagen, Denmark*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539650

In this context, the GC requires participants to develop a software solution that continuously computes answers to two *queries* on the given data set. The first query (Query 1) hinges on one of the most essential indicators used in technical analysis to identify trends, namely the exponential moving average (EMA). The second query (Query 2) builds on top of Query 1 and aims to detect interesting breakout patterns by observing the per-symbol EMA computed at different intervals. Both the queries must be computed by grouping incoming events in 5-minute non-overlapping windows.

In this paper, we present a solution to both the queries. Our solution relies on two well-known open-source distributed frameworks: Apache Kafka for data ingestion and Apache Flink for stream processing. Event notifications replayed from the data set are pushed into Kafka topics. A Flink application consumes messages from the topics and computes the required indicators for both the queries.

The remainder of this paper is organized as follows. In Sec. 2 we provide background information about the market data analysis performed in the GC as well as the framework we used for implementing the solution. In Sec. 3 we describe the solution we developed and report evaluation results in Sec. 4. We conclude in Sec. 5 with some final remarks.

## 2 BACKGROUND

### 2.1 Real-Time Detection of Breakout Patterns

Nowadays, tracking and evaluating quantitative financial indicators is a fundamental activity for traders aiming to identify trends in the development of instrument price in a timely manner. As prices can rapidly increase and decrease, quickly identifying the start of a trend is crucial to be able to buy (or sell) so as to maximize the profits (or, at least, minimize the loss). Furthermore, exploiting predictive analytics, applications may even help traders to identify trends *before* they actually start.

The DEBS GC defines two queries to be computed against financial market data, which revolve around the aforementioned goals. Query 1 involves one of the most essential indicators used to identify trends, i.e., the exponential moving average (EMA). To provide an answer to Query 1, we need to compute the EMA per symbol for every event window. Therefore, the latest observed price in the current window is weighted and summed to the previously computed EMA. Such quantitative indicators are then exploited in Query 2, where we aim to detect trends in the instrument prices. Indeed, by tracking the EMA computed over different time intervals, we can identify breakout patterns for each symbol.

In general, breakout patterns describe relevant variations in the development of a price that possibly indicate the start of a trend, even if only temporary [5]. We are interested in detecting two different patterns: *bullish breakouts*, when the price is starting to rise, and *bearish breakouts*, when the price is decreasing. Properly identifying these trends in a timely manner allows traders to monetize this knowledge by immediately buying, in case of a bullish breakout, or selling, in case of a bearish breakout, to maximize revenue.

Relying on the computed EMA over a shorter time interval $j_1$ and a longer time interval $j_2$, we can detect a bullish breakout pattern for a symbol when the EMA associated with $j_1$ starts to overtake the EMA with $j_2$. When this happens, we should immediately notify the trader so as to benefit from a relatively low price. In this aim, within the GC intervals have the granularity of minutes and, specifically, we have $j_1 = 38$ and $j_2 = 100$.

Based on analogous reasoning, we detect a bearish breakout pattern for a symbol when the EMA associated with $j_2$ starts to overtake the EMA associated with $j_1$. When this happens, we should immediately generate a sell advice notification so that a trader can still sell at a relatively high price. As per the GC instructions, we again adopt a granularity of minutes and use the same EMA intervals specified above for the bullish breakout.

The GC require participants to provide responses to both the queries. In order to evaluate and benchmark the solutions, the platform used by GC organizers mimics the usual behavior of traders using market terminal solutions, as they subscribe to sets of symbols for which they want to track particular events and opportunities (e.g., buy and sell advice notifications). Therefore, GC solution must process batches of trading events and handle subscriptions to symbols. To solve Query 1, the solution is expected to provide the latest EMA for $j_1$ and $j_2$ for each subscribed symbol in every batch. To solve Query 2, the solution is expected to provide the last 3 crossovers (buy/sell) notifications with the associated timestamps for each subscribed symbol. For both the queries, incoming events must be processed in 5-minute non-overlapping (i.e., tumbling) windows. Furthermore, windows cannot be evaluated before the next window starts.

## 2.2 Apache Flink

Apache Flink [3] is an open-source, distributed engine for stateful computation over both *bounded* and *unbounded* data sets (i.e., it supports both batch and streaming computations). Flink has gained a lot of traction over the years as it combines high-level processing APIs, which ease development efforts, and a scalable computation engine, which promises high throughput and low latency for a wide range of use cases. Not surprisingly, Flink has been frequently adopted by DEBS GC participants (e.g., [9–11, 13, 14]).

Flink architecture comprises three key components, namely the client, the TaskManager and the JobManager. The client allows users to submit and manage jobs to a running cluster. Actual data processing is carried out within one or more distributed TaskManagers. The JobManager is a centralized entity, responsible for coordinating the distributed execution of jobs in the cluster by communicating with the TaskManagers, as well as scheduling components (specifically, subtasks, which are the actual execution units in Flink) to the TaskManagers.

As regards application development, Flink offers various levels of abstractions to define the processing logic as a series of data-centric transformations [1], ranging from a low-level stream processing API, where users can manually define the logic applied to each incoming event, to higher-level SQL-oriented APIs. Furthermore, specialized libraries are available to solve common analytics use-cases (e.g, complex event processing and graph analytics). To support the definition of streaming applications, Flink has native mechanisms to easily cope with state management, windowing and late events, which represent key challenges for streaming systems.

Flink provides support to ingest data streams from different sources; specifically, it provides an Apache Kafka connector for reading data from and writing data to Kafka topics with exactly-once guarantees. Kafka is currently the most popular open-source distributed framework used to ingest data streams into the processing platforms.

Fault tolerance plays a key role in the design of Flink, which features an efficient and closely integrated mechanism to take state snapshots at run-time [2]. By means of this mechanism, frequent computation checkpoints can be created during execution, enabling quick recovery in case of failures.

## 3 PROPOSED SOLUTION

In this section we describe our solution to the DEBS GC. We first give a high-level overview of the approach. Then, we describe the Flink streaming topology used for data processing, before discussing implementation details.

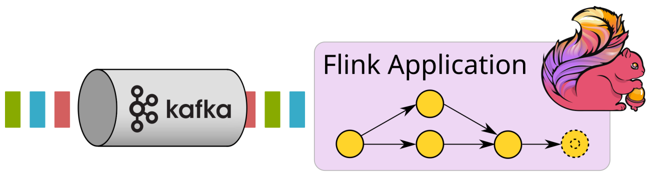

**Figure 1: Overview of the solution.**

### 3.1 Overview

As illustrated in Fig. 1, our solution relies on Apache Kafka for data ingestion and Apache Flink for data processing. Incoming trading data are pushed to a Kafka topic and consumed by a Flink job. In such a scenario, Flink provides consistent data movement and computation, while Kafka provides data durability and exactly-once delivery guarantees.

Our solution comprises two main application components, which, respectively, act as the *producer* of the streaming data and the *consumer*, from Kafka point of view. The producer reads data from the gRPC API provided within the GC and sends them to a Kafka topic. These events are consumed by the Flink application, which subscribes to the Kafka topic and processes incoming data.

The Kafka Producer, given the address of a Kafka cluster and the name of the topic we want to use, creates a client to connect to a gRPC server provided by the GC. This connection provides means for both retrieving batches containing input events and publishing the results of the computation, as well as evaluating the solution performance. Batches of incoming events are parsed before they are sent to Kafka for analysis.

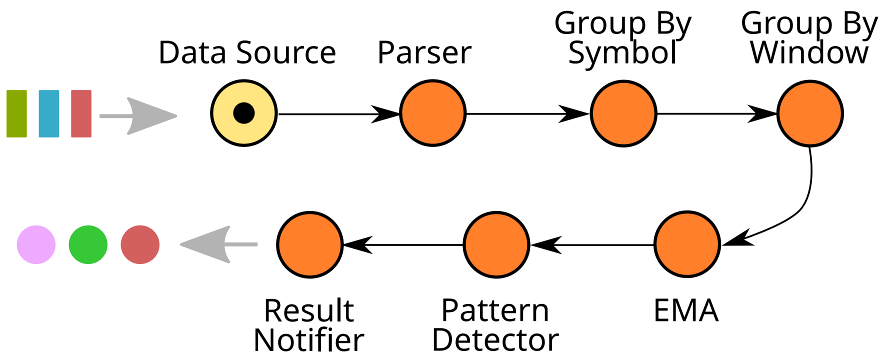

**Figure 2: Stream processing application used within our solution.**

The Flink job is fed with data from the Kafka topic. The notion of time used within our application relies on the event time associated with the input data stream, i.e., the time when each individual event occurred, as specified in the GC data set. In this scenario, the progress of time depends on the data and does not depend on any wall clock time. Event timestamps are typically embedded within the records before they enter Flink and can be retrieved as needed. Specifically, when reading data from Kafka, we automatically extract the message timestamp for each event.

The integration between Kafka and Flink is easily realized thanks to the built-in Kafka connector provided by Flink. In the next section, we will describe the streaming topology we use for data processing.

## 3.2 Flink Topology

The core data processing tasks required by our solution are performed by a Flink streaming application. Figure 2 provides an illustration of the logical building blocks comprising the streaming topology. Incoming events are retrieved by a data source from Kafka and then parsed. Since the Query 1 indicators must be computed separately for each traded symbol, we group incoming events based on their associated symbol. Then, according to the GC requirements, we group events into non-overlapping 5-minute windows. As soon as each window is complete, we compute the EMA indicators required for Query 1. The computed EMA are also used to identify the breakout patterns for Query 2 and produce notifications as the result.

Considering the definition of the topology described above in Flink, the first operator applied to the incoming stream is a `map`, which parses string-serialized events and extracts the event attributes useful for computation. Parsed events are then processed by a `keyBy` operator, which produces a keyed stream, partitioning the events by symbol.

The keyed stream flows into 5-minute tumbling windows. We apply custom aggregation and processing functions to each window, so as to specify the computation that we want to perform on each event window once the system determines that it is complete, i.e., ready for processing. Then, we apply a `windowAll` that collects the output of previous window functions for all the symbols. The last operator is a custom process function that properly prepares output results, before sending them to a Kafka sink, where they are published.

## 3.3 Implementation Details

The source code of our solution has been publicly released.[1] The programming language used for implementation is Java.

Input event parsing is realized by extending the MapFunction class with the class MapFunctionEvent, in order to parse strings received by the data source and return a DataStream of Event objects. The Event class was realized to save each record and its attributes useful to computation. The core query computation occurs inside window functions, implemented by means of two classes *MyAggregateFunction* and *MyProcessWindowFunction*. The first one aims to aggregate intermediate results within a window, so its `add()` method (used for aggregation) is called for each new event entering the window. The second class instead is used to evaluate windows when they are complete.

An AggregateFunction requires an Accumulator object to be defined. We created a new class *MyAccumulator* to keep track of the last price for each symbol within the current window and the batches where each symbol appears within the window. These data are eventually passed to MyProcessWindowFunction class, which computes the required indicators. Specifically, this class uses the following key-valued state information to keep track of relevant data for query computation:

- count of window number;
- EMA(38) and EMA(100) computed in the previous window for each symbol;
- previous crossovers (buy/sell) notifications for each symbol.

As regards Query 1, since the result is expected to be returned per batch, we need to keep track of all symbols with their EMAs despite the batch they belong to. Indeed, a batch might have a variable size that could be longer or shorter than window length, but a window is evaluated only when event timestamps reach its end. Therefore, in order to compute EMAs we call the `computeEMA()` method, which first checks whether it is the first time ever that a given symbol (i.e., key of the stream) appears. If it does, the last EMA for that symbol is set to 0 as required, otherwise the last EMA is retrieved. Then, we update the EMA and store the corresponding result with

[1]https://github.com/ceciliacal/DEBS_2022

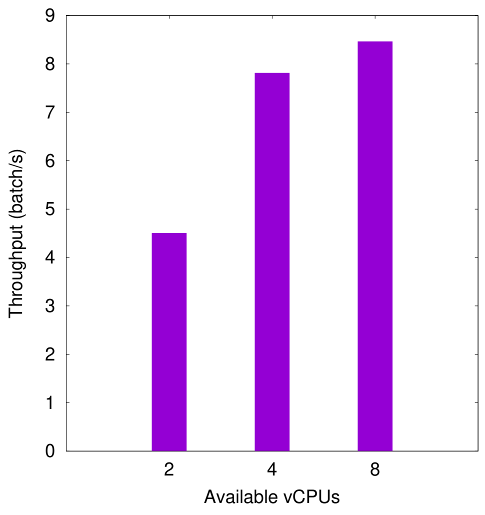

**Figure 3: Throughput of our solution.**

the proper key for future computation. This approach is clearly adopted to compute both EMA(38) and EMA(100).

Query 2 requires us to detect breakout patterns. To find a bullish pattern we check if EMA(38) is larger than EMA(100) in the current window and EMA(38) is less or equal than EMA(100) in the previous window; otherwise, we possibly discover a bearish pattern if EMA(38) is less than EMA(100) in the current window and EMA(38) is larger than or equal to EMA(100) in the previous one. If any pattern is found, the current window end timestamp is included in the generated notification.

Our solution can be configured to run under different parallelism settings, thanks to Flink fine-grained parallelism configurability. In particular, depending on the computing infrastructure in use, our software can scale from a single-core machine to multiple cores and nodes.

## 4 EVALUATION

To evaluate the performance of our solution, we deploy it on a single Amazon EC2 `c4.2xlarge` instance, equipped with 8 vCPUs and 16 GB of memory. We also run Kafka on the same machine. For this purpose, we used Docker containers to run Kafka and Zookeeper and deployed them using *docker-compose*.

We consider different settings by varying the number of vCPUs available to our application components, from 2 to 8 (i.e., all the vCPUs of the machine). Figure 3 shows the throughput achieved in the experiments. Our solution manages to process 8.46 event batches per second when allowed to use all the vCPUs available on the machine. When restricted to just 2 vCPUs, the solution achieves about half the throughput and, specifically, 4.49 batches per second. When using 4 vCPUs, the measured throughput is 7.81 batches per second.

## 5 CONCLUSION

In this paper we presented our solution to the DEBS 2022 Grand Challenge. The topic of the challenge revolved around financial market data, which require continuous and timely monitoring of price variations to detect patterns of interest and enable informed decision-making. Our solution relies on Apache Kafka message queues to ingest market data and Apache Flink for stream processing. The built-in abstractions provided by Flink well supported us in developing a solution to the challenge. As future work, we plan to further optimize the performance of the proposed solution, also by exploiting application elasticity to enable seamless scaling [4].

## REFERENCES

[1] Apache Flink. 2022. Apache Flink Documentation. https://nightlies.apache.org/flink/flink-docs-stable/
[2] 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. https://doi.org/10.14778/3137765.3137777
[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Database Engineering Bulletin* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf
[4] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Run-Time Adaptation of Data Stream Processing Systems: The State of the Art. *Comput. Surveys* (2022), 36 pages. https://doi.org/10.1145/3514496
[5] J. Chen. 2022. The Anatomy of Trading Breakouts. https://www.investopedia.com/articles/trading/08/trading-breakouts.asp
[6] Sebastian Frischbier, Mario Paic, Alexander Echler, and Christian Roth. 2019. Managing the Complexity of Processing Financial Data at Scale - An Experience Report. In *Complex Systems Design & Management*. Springer International Publishing, Cham, Switzerland, 14–26. https://doi.org/10.1007/978-3-030-34843-4_2
[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*. ACM, New York, NY, USA. https://doi.org/10.1145/3524860.3539645
[9] Giacomo Marciani, Marco Piu, Michele Porretta, Matteo Nardelli, and Valeria Cardellini. 2016. Real-Time Analysis of Social Networks Leveraging the Flink Framework. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, DEBS '16*. ACM, New York, NY, USA, 386–389. https://doi.org/10.1145/2933267.2933517
[10] Josip Marić, Krešimir Pripužić, and Martina Antonić. 2021. DEBS Grand Challenge: Real-Time Detection of Air Quality Improvement with Apache Flink. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, DEBS'21*. ACM, 148–153. https://doi.org/10.1145/3465480.3466930
[11] Nicolo Rivetti, Yann Busnel, and Avigdor Gal. 2017. FlinkMan: Anomaly Detection in Manufacturing Equipment with Apache Flink: Grand Challenge. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems, DEBS '17*. ACM, New York, NY, USA, 274–279. https://doi.org/10.1145/3093742.3095099
[12] S. Seth. 2021. Basics of Algorithmic Trading: Concepts and Examples. https://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp
[13] Gengtao Xu, Jing Qin, and Runqi Tian. 2020. Optimized Parallel Implementation of Sequential Clustering-Based Event Detection. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, DEBS '20*. ACM, New York, NY, USA, 208–213. https://doi.org/10.1145/3401025.3401760
[14] Zongshun Zhang and Ethan Timoteo Go. 2020. Anomaly Detection for NILM Task with Apache Flink. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, DEBS '20*. ACM, New York, NY, USA, 199–203. https://doi.org/10.1145/3401025.3401758