DEBS Grand Challenge 2022: Detecting Technical Trading Patterns in Financial Data with Apache Flink

Quan Pham, Quang Nguyen, Ryte Richard, Shekhar Sharma, Xavier Ruiz[*]
{qvpham, nddq, ryte, shekhars, xruiz}@bu.edu
Boston University
Boston, MA, USA

ABSTRACT

The DEBS Grand Challenge is an annual competition in which participants strive to build the fastest and most scalable distributed and event-based systems that solve a practical problem. For the year 2022, the challenge focuses on real-time complex event processing of real-world high-volume financial trading data. The goal is to efficiently compute specific trend indicators and detect patterns that would assist real-life traders in deciding to buy or sell on the financial markets. This paper aims to solve the above challenge using Apache Flink [1] – an open-source, unified stream-processing and batch-processing framework developed by the Apache Software Foundation.

CCS CONCEPTS

• **Information systems → Data stream mining**.

KEYWORDS

stream processing, event-driven systems, Apache Flink

ACM Reference Format:
Quan Pham, Quang Nguyen, Ryte Richard, Shekhar Sharma, Xavier Ruiz. 2022. DEBS Grand Challenge 2022: Detecting Technical Trading Patterns in Financial Data with 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, 5 pages. https://doi.org/10.1145/3524860.3539648

1 INTRODUCTION

The DEBS Grand Challenge is a yearly competition in which participants solve a particular task by performing stream data processing techniques on given event-based data. The 2022 DEBS Grand Challenge [3] provides real-world high-volume streams of financial trading data and sets the goal for participants to efficiently compute trend indicators and detect patterns used by real-life traders in making trading decisions (i.e., buying or selling) on the financial markets [2]. Besides fundamental trading, algorithmic trading is a widely-used approach that allows traders, analysts, and other stakeholders to identify trends, whether upwards or downwards, in the price development of a symbol of their interest early on. This paper uses Apache Flink [1] to connect with the provided data sources and uses custom windows to process the market data and compute the solutions. Our code is available on GitHub [6].

The data [4] provided by the competition includes 289 million tick data events for 5504 equities and indices that are traded on three European exchanges: Paris (FR), Amsterdam (NL), and Frankfurt/Xetra (ETR). Among all data events, we only consider the data events containing the attributes that are relevant for the Grand Challenge queries. The relevant attributes include the unique identifier for a symbol and its respective exchange, security type (i.e., equity or index), last trade price, trading time, and trading date.

The challenge involves two tasks. The first task is to calculate two exponential moving averages, EMA(38) and EMA(100), for each symbol. The exponential moving average is an essential quantitative indicator used in technical analysis to identify trends. The second task is to identify breakout patterns for each symbol by tracking its two exponential moving averages over different intervals. There are two breakout patterns: a bullish breakout indicates a buying opportunity, and a bearish breakout generates sell advice. As data events arrive in batches, a framework that can efficiently process incoming batches of data is required to determine the solutions to the two tasks.

2 BACKGROUND

Apache Flink [1] is an open-source framework and distributed processing engine for stateful computations over unbounded data streams. Flink's design enables the system to perform operations at in-memory speed and at any scale. Every application in Flink can be logically described in the form of a Directed Acyclic Graph (DAG) where each node of the graph is an operator with a specific function. Operators in Flink can be of two types, stateful and stateless. The stateless function does not store any data whereas stateful functions maintain a state and store data to process future events. Flink guarantees exactly-once semantics for both kinds of operators. In other words, Flink can ensure that each incoming event affects the internal state exactly once under failures. For fault tolerance and recovery purposes, the state is stored in persistent storage at regular intervals by checkpointing.

Every program in Flink has an execution environment that provides functions to control the execution of a job (such as parallelism, checkpointing, logging, etc.) and also to interact with the outer world, i.e. incoming streams of data. The context in which a streaming program is executed is called a *StreamExecutionEnvironment.*

Flink programs ingest data from various sources using source functions which act as the entry point for data streams. A *RichSourceFunction* is the base class for implementing a parallel data

[*]All authors contributed equally to this research.

DEBS '22, June 27–30, 2022, Copenhagen, Denmark
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539648

source that has access to the context information and additional life-cycle methods.

The *KeyedProcessFunction* is a stream processing operation that provides the basic building blocks such as events, state, and timer to all acyclic streaming applications. The *KeyedProcessFunction* has access to the keyed state and timers. It handles events by getting invoked every time a new event is received from a keyed data stream. This function is used to transform the input streams into results by the program logic. For fault tolerance, the function gives access to the keyed state through the *RuntimeContext.*

Stateful operations rely on a state to remember information across multiple events, and Flink provides different state backends that specify how and where the state is stored. Keyed state, in particular, is maintained in an embedded KV store. The state is partitioned and distributed strictly together with the streams read by the stateful operators. The keyed state provides access to different types of states that are all scoped to the current event's key, and thus can only be used on keyed streams. The alignment of streams' keys and state ensures that all state updates are local and guarantees consistency without adding transaction overhead. It also allows Flink to redistribute the state and adjust the stream partitioning transparently.

The results obtained in the Flink program can be published by adding them to a sink. The *invoke()* function is executed each time an event is encountered by the sink, through this function, the result is published to the chosen destination. The *RichSinkFunction* is a kind of sink function which not only provides basic sink functionality but also includes other utility methods such as initialization, tear-down, state management functions, etc.

3 DESIGN

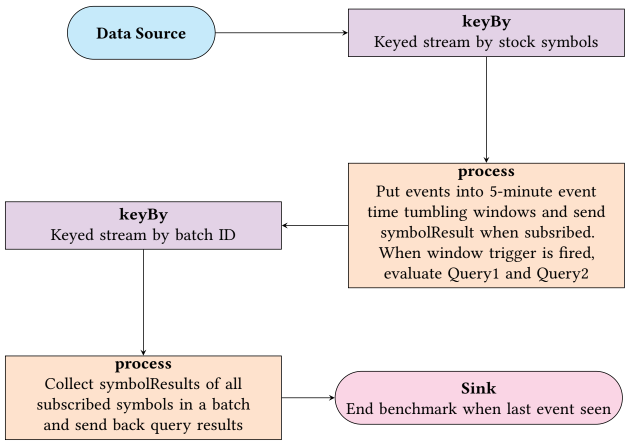

Data Source
keyBy
Keyed stream by stock symbols
process
Put events into 5-minute event time tumbling windows and send symbolResult when subsribed. When window trigger is fired, evaluate Query1 and Query2
keyBy
Keyed stream by batch ID
process
Collect symbolResults of all subscribed symbols in a batch and send back query results
Sink
End benchmark when last event seen

Figure 1: Application Design

In this section, we will discuss the design of our solution. We will talk about our implementation in detail in the next section.

We have chosen the data-stream approach for our solution, meaning that we model the incoming batch of data as a stream of data. Data is retrieved from the given competition platform in batches. Each batch contains a list of Event objects defined by the competition.

Next, we put those objects into the data stream. Using the event's symbol name as the key, we keyed the stream using the keyBy operator and then pushed them through a custom window process operator. Here, both queries are evaluated for each symbol every 5 minutes of event time. The output of this operator is then keyed by the event's batch ID and forwarded to another processing operator. Here, the results of those queries for each batch are sent back to the competition platform for evaluation. Finally, the events will be forwarded to the sink.

We also have a functional graphic user interface that utilizes the output of our stream processing application to display the results, much similar to a trading platform.

4 IMPLEMENTATION

In this section, we will discuss how we implemented our design for the solution. First, we will talk about how data is retrieved, then how that data will be used in evaluating the two queries given by the challenger. We will also be going through the technical aspects of the implementation of the two queries.

4.1 Data Retrieval

The *challenger.proto* file provided generates the sources that enable communication to the evaluation platform to fetch the data. We create a channel and a new benchmark that uniquely specifies the evaluation instance we are running. A Remote Procedure Call (gRPC) is used to maintain communication with the server. The data source, *GrpcClient* extends the *RichSourceFunction* where we start the benchmark. The data is received in batches. Each batch contains a list of events, a list of lookup symbols, the batch sequence ID, and a boolean value that indicates whether the batch is the last batch of the evaluation run. The lookup symbol list indicates the symbols (a symbol refers to an identifier consisting of a unique string together with the exchange code of the exchange that the instrument is being traded on) for which we have to send the query results after the batch has been processed. Each batch has a different set of lookup symbols. We iterate over the event list and wrap each with a *SymbolEvent* object and put it into the data stream. After we have placed all the events in a batch into the data stream, we put a dummy event for each symbol present in the symbol lookup list for the batch onto the data stream to act as a marker indicating the end of the batch.

```
DataStream<SymbolEvent> events = env
  .addSource(grpc)
  .name("API")
  .rebalance()
  .assignTimestampsAndWatermarks(
    WatermarkStrategy.forMonotonousTimestamps();
```

4.2 Data Processing

We key the data stream by the symbol since we have to generate the query result for each symbol. The *SymbolQueryProcess* extends the *KeyedProcessFunction* processes each event keyed by the symbol value. The *KeyedProcessFunction*, an extension of the *ProcessFunction*, gives access to the key of timers in its *onTimer()* method. For each event, we place the event within a 5-minute event time tumbling window, register the *TimerService* for the window trigger timestamp and update the last trading price for the window. Since the *TimerService* deduplicates timers per key and timestamp, i.e., there is at most one timer per key and timestamp. If multiple timers are registered for the same timestamp, the *onTimer()* method will be called just once.

```
DataStream<SymbolResult> symbolResultDataStream =
  events
    .keyBy(symbolEvent -> symbolEvent.getSymbol())
    .process(new SymbolQueryProcess(Time.minutes(5)));
```

At the end of a 5-minute event time window for a symbol, Flink invokes the *onTimer* method, where we calculate the EMA values and crossover events for the symbol. The EMA value is calculated using the window's last trading price and the corresponding EMA values of the previous 5-minute tumbling window for the symbol. The crossover values are calculated from the newly evaluated EMA values and the previous window's EMA values. We will update the last crossover event if there is a crossover of corresponding previous and new EMA values. We have to keep track of the last three crossover events, so we shift the crossover events back for each new one.

```
if (ema38Prev <= ema100Prev && ema38New > ema100New) {
  if (secondLastCrossover.value() != null)
      thirdLastCrossover.update(Tuple2.of(
      secondLastCrossover.value().f0,
      secondLastCrossover.value().f1));
  if (lastCrossover.value() != null)
      secondLastCrossover.update(Tuple2.of(
      lastCrossover.value().f0,
      lastCrossover.value().f1));
  lastCrossover.update(Tuple2.of(timestamp, 0L));
} else if (ema38Prev >= ema100Prev &&
         ema38New < ema100New) {
  if (secondLastCrossover != null)
      thirdLastCrossover.update(Tuple2.of(
      secondLastCrossover.value().f0,
      secondLastCrossover.value().f1));
  if (lastCrossover != null)
      secondLastCrossover.update(Tuple2.of(
      lastCrossover.value().f0,
      lastCrossover.value().f1));
  lastCrossover.update(Tuple2.of(timestamp, 1L));
}
```

Now consider the scenario when we have to send the symbol EMA and crossover values if the symbol has been subscribed for in a batch. In the source, we set dummy events for each symbol in the lookup symbol list of a batch. When the *KeyedProcessFunction* receives this dummy event, it will collect the current keyed state EMA and crossover event values in a *SymbolResult* object and some batch details and send them forward to the next operator.

4.3 Sending Results

We then key the resultant *SymbolResult* data stream by the batch ID since we have to send the query results for each batch. We send along from the source in the dummy event the number of subscribed symbols for the batch. This value is used in the *BatchResultProcess* to keep track of the number of *SymbolResults* it has received. When the number of *SymbolResult* received for the batch reaches the lookup symbol count, it accumulates all the *SymbolResult* values into Query 1 and Query 2 result responses. The gRPC client sends these values back to the evaluation platform.

```
DataStream<Tuple2<Long, Boolean>>
  batchResultDataStream = symbolResultDataStream
   .keyBy(symbolResult -> symbolResult.getBatchId())
   .process(new BatchResultProcess());
```

Finally, we use the sink to keep track of the number of batches evaluated so far and end the benchmark when we have processed all the batches. Since Flink is a framework for continuous data processing and streaming programs usually run indefinitely, there is no standard method to stop the application gracefully. The last batch of an evaluation run contains an indicator to know when we can stop the source connection. We pass this indicator to the downstream operators in the dummy event. When the last batch has been processed and reaches the sink, we store the corresponding batch ID of the last batch. Now since we parallelize processing, it is possible that the last batch received is not necessarily the last batch processed. So we have to keep count of the number of batches processed, but since the batch IDs are sequential, we know for sure we can receive at most the last batch ID number of batches. Even if the last batch were to be processed earlier or is the last to be processed, when it is received at the sink, we would update the number of batches we are expecting and check if the processed batch count is equal to the last batch ID. If so, it will end the benchmark or keep waiting until all batches are processed, and the condition is met.

```
batchResultDataStream.addSink(new BlackHole(benchmark));
```

5 GRAPHICAL USER INTERFACE

5.1 Design

Designing the frontend required that it can be run separately from the main application and it did not impact the performance of the main application. To decouple the main application from the frontend, we use a message queue. As the main application is uploading its query 1 and query 2 results, it also publishes these results as messages to a message broker. This is seen in (step 1) Figure 2.

Concurrently, we create a consumer process that is off the critical path of the main program. The consumer consumes messages (step 2) and writes them to a database (step 3). To see the results, we simply create a frontend that queries the database at some time interval and visualizes the collected data (step 4).

The reason we use a message queue, instead of writing to a database directly, is because databases take much longer to write to, and using a message queue allows us to do some further processing to format the data of the main application for writing to the database.

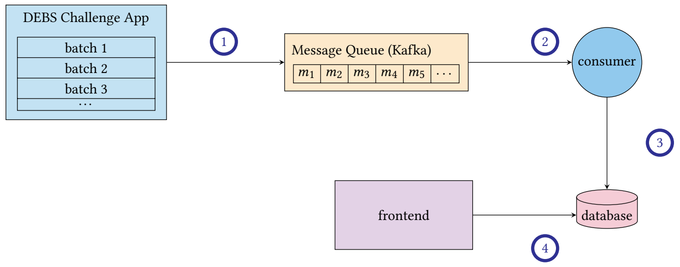
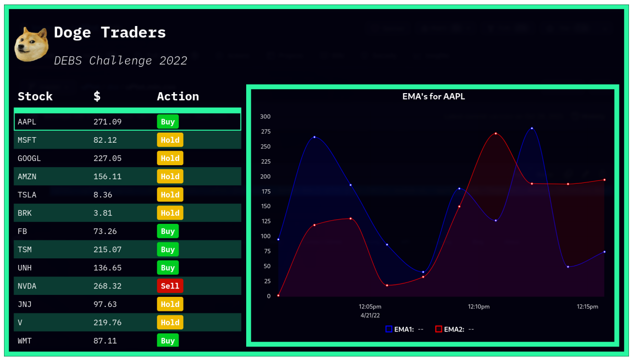


Figure 2: GUI



Doge Traders

DEBS Challenge 2022

Stock	$	Action
AAPL	271.09	Buy
MSFT	82.12	Hold
GOOGL	227.05	Hold
AMZN	156.11	Hold
TSLA	8.36	Hold
BRK	3.81	Hold
FB	73.26	Buy
TSM	215.07	Buy
UNH	136.65	Buy
NVDA	268.32	Sell
JNJ	97.63	Hold
V	219.76	Hold
WMT	87.11	Buy

Figure 3: Desktop Screenshot

It is also not feasible to stream data directly from the main application to the frontend because there may be many instances of the frontend, slowing down the main application.

The database is not necessary, and the frontend itself can act as a consumer and show the data as it comes in, but accessing data that is older would not be possible.

5.2 Implementation

The main application has to be started with the SEND_TO_KAFKA environment variable set. When this is set, it will create producers for the results of queries 1 and 2 and publish the data on each batch completion. The main application produces query 1 results to topic 'query1' and query 2 results to topic 'query2'.

We chose to use Kafka as our message queue because it is fault-tolerant, highly available, and can prevent data loss. It allows for complex publish-subscribe systems and was perfect for our use case.

The consumer is a simple Node.js process that subscribes to both of the query topics and for each message writes it to a time-series database.

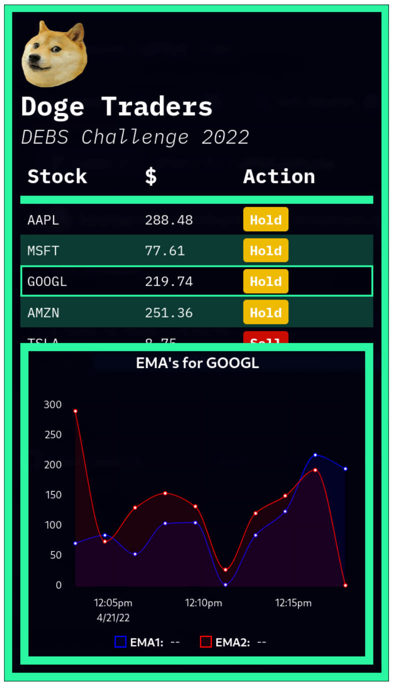
Time-series databases are simply databases that have been optimized for use with time-series data (e.g. query 1 and query 2). We chose to use QuestDB as opposed to a popular alternative InfluxDB, as it self-reports to be able to ingest at 6.4 times the rate at 1.4 million records per second.

For the frontend, we build the graphics using React, a frontend framework that automatically handles changes in data and updates the UI accordingly. It is modular, extendable, and easily maintainable. For serving the webpage and making requests to the database, we use the Express library with Node.js.

Launching all of this machinery is tedious done manually, running the application is easier done with Docker Compose. It containerizes Kafka, the consumer processes, the database, and the frontend. By containerizing everything, it is feasible to scale the frontend servers and consumer processes easily.

5.3 Results

The frontend is responsive, polls the database every one second, and clicking a symbol outputs a graph of the last 15 minutes of data.



Doge Traders

DEBS Challenge 2022

Stock	$	Action
AAPL	288.48	Hold
MSFT	77.61	Hold
GOOGL	219.74	Hold
AMZN	251.36	Hold

Figure 4: Mobile Screenshot

6 RESULTS

6.1 Setup

The application was built using Java and Apache Flink version 1.14.4 on a Ubuntu installation. The application was deployed on a cluster of 3 virtual machines (VMs) each having 4 cores and 8GB of main memory. The configuration at the time of deployment comprised of default parallelism set to 1 (as the parallelism of each operator was specified in the code), the job manager memory set to 1700 MB, and the task manager memory set to 7680 MB. The memory allocated for the task manager included all memory usage within the task manager process including the JVM metaspace and other overheads. The number of task slots was set to 4 as 4 cores were available for processing. To maximize the throughput and minimize latency, logging was disabled for the application. In-memory state backend was used for operators with the state as it has 10X faster transfer time when compared to the other state backend in Flink, i.e. RocksDB.

6.2 Deployment

It was observed that the application produced 3X better results when deployed on a cluster of 3 VMs as compared to a single machine with 12 cores keeping the memory and operating system the same. During deployment, different configurations with varying memory distributions between job manager and task manager were compared to get the best performance. The configuration stated above resulted in the best performance.

6.3 Performance

The application processed 5940 batches, each having 10,000 events. The total time taken to process these 59 million events was around 111 seconds (≈2 minutes). The application produces results for 100 percent of the given batches for both query 1 and query 2. The average latency (50th percentile) for query 1 was 94.896 milliseconds and the 90th percentile latency was 128.712 milliseconds. The average latency for query 2 was 96.403 milliseconds and the 90th percentile latency was 130.482 milliseconds. The throughput for both query 1 and query 2 was 53.107 batches per second. This amounts to the Flink application processing about half a million events per second. The performance of the application in terms of latency can be better understood through the cumulative distribution function (CDF) graph given below.

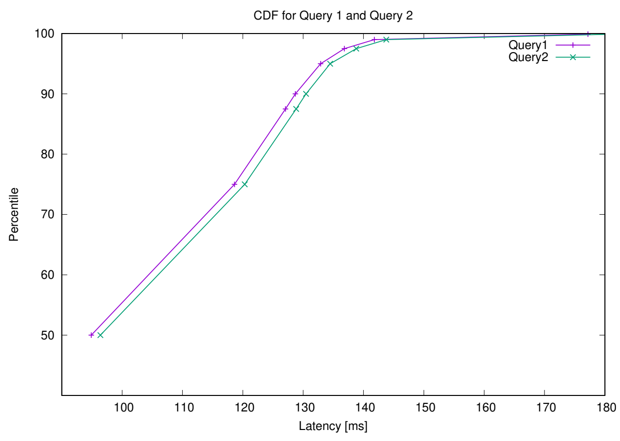


Figure 5: CDF for Query 1 and Query 2

7 CONCLUSION

In this paper, we presented the design and implementation of an Apache Flink-based application to solve the DEBS 2022 Grand Challenge: Detecting Trading Trends in Financial Tick Data. Our primary focus was to develop an end-to-end, functional system that receives data from the gRPC source and correctly as well as efficiently processes the required queries. The application is capable of handling very high volumes of real-time data and provides 'buy' or 'sell' advice for financial instruments of interest to traders. We have also developed a graphical user interface for the system which allows traders to see the information in a human-readable format. The graphs in the GUI help in making quick decisions for the trading of financial instruments.

As we wanted the best performance metrics for our system, we do not currently have very robust fault tolerance policies. Since most of the fault tolerance techniques generally used today either have high run-time overhead or a high recovery overhead, they are not suitable for financial market applications such as ours. Further work can be done in the direction of making the application more robust in terms of fault tolerance and recovery by integration of systems like LineageStash[7] and DS2[5].

8 ACKNOWLEDGMENTS

We would like to thank professor Vasiliki Kalavri for her mentorship and support throughout this project.

REFERENCES

[1] 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 Data Eng. Bull.* 38 (2015), 28–38.

[2] Sebastian Frischbier, Mario Paic, Alexander Echler, and Christian Roth. 2020. Managing the Complexity of Processing Financial Data at Scale - An Experience Report. In *Complex Systems Design & Management*, Guy André Boy, Alan Guegan, Daniel Krob, and Vincent Vion (Eds.). Springer International Publishing, Cham, 14–26.

[3] Sebastian Frischbier, Jawad Tahir, Christoph Doblander, Arne Hormann, Ruben Mayer, and Hans-Arno Jacobsen. 2022. 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).

[4] Sebastian Frischbier, Jawad Tahir, Christoph Doblander, Arne Hormann, Ruben Mayer, and Hans-Arno Jacobsen. 2022. *2022.* DEBS 2022 Grand Challenge Data Set: Trading Data.

[5] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* USENIX Association, Carlsbad, CA, 783–798. https://www.usenix.org/conference/osdi18/presentation/kalavri

[6] Team Doge Traders. 2022. Github - DEBS Challenge 2022 Solution. https://github.com/ryterichard/debs-challenge-2022.

[7] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage Stash: Fault Tolerance off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19).* Association for Computing Machinery, New York, NY, USA, 338–352. https://doi.org/10.1145/3341301.3359653