Efficient Processing of High-Volume Tick Data with Apache Flink for the DEBS 2022 Grand Challenge

Stefanos Kalogerakis, Antonis Papaioannou and Kostas Magoutis
{skaloger,papaioan,magoutis}@ics.forth.gr
Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH)
Computer Science Department, University of Crete
Heraklion, Greece

ABSTRACT

The DEBS 2022 Grand Challenge (GC) focuses on real-time complex event processing of real-world high-volume tick data. The goal of the challenge is to efficiently compute specific trend indicators and detect patterns resembling those used by real-life traders to decide on buying or selling on the financial markets. Motivated by the exciting nature of the 2022 GC topic, we undertook the design and implementation of a system that addresses it. Our design features a custom windowing mechanism that leverages event semantics. This paper reports on our team's solution to the 2022 GC and reports on the performance we observed in the evaluation testbed.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

data stream processing, high-volume processing of financial data

ACM Reference Format:
Stefanos Kalogerakis, Antonis Papaioannou and Kostas Magoutis. 2022. Efficient Processing of High-Volume Tick Data with Apache Flink for the DEBS 2022 Grand Challenge. 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/10.1145/3524860.3539649

1 INTRODUCTION

The 2022 DEBS Grand Challenge (GC) [3] supported by Infront Financial Technology[1] focuses on real-time complex event processing of high-volume tick data. In the real-world data set provided [2], about 5000+ financial instruments are being traded on three major exchanges over the course of a week. The goal of the challenge is to efficiently compute specific trend indicators and detect patterns that resemble those used by real-life traders to decide on buying or selling on financial markets. The 2022 DEBS GC requires developers to implement a basic trading strategy aiming at (a) identifying trends in price movements for individual equities using event aggregation over tumbling windows (Query 1) and (b) triggering buy/sell advises using complex event processing upon detecting specific patterns (Query 2). The first query implements the exponential moving average (EMA) [4], an indicator per symbol used in technical analysis to identify trends. Q2 uses the quantitative indicators of query 1, tracking two EMAs (with different smoothing factors) per symbol computed over different intervals to identify breakout (indication of market turning to bullish or bearish) patterns.

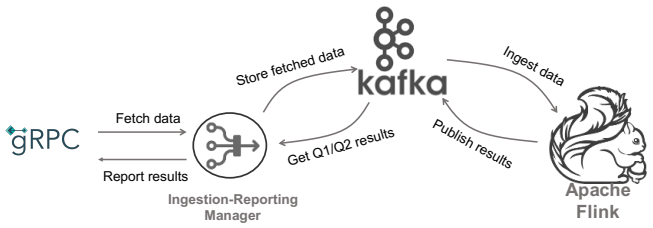

Figure 1: Data analysis pipeline

The evaluation dataset [2] is provided by the GC platform via a gRPC-based API in a continuous stream of event batches, B_i, $i = 0, 1, 2, \ldots$ Each batch B_i includes a list of events, each event comprising a symbol (identified by unique ID and exchange), type (equity or index), last trade price, date of last trade, and time of last update (bid/ask/trade). Each batch also specifies *lookup symbols* that the evaluation platform *subscribes to* for this batch. The analytics pipeline must report answers to Query 1 and 2 for the subscribed symbols for each batch B_i back to the GC platform. Performance is evaluated based on average throughput and mean (for the two queries) of the 90th-percentile latency for each batch. The reporting mechanism is also based on the supported gRPC API.

We decided to use the Apache Flink [1] framework as our data analysis platform to leverage the scalability and operational reliability afforded by the base Flink platform, customizing the application logic to solve the DEBS 2022 GC in an accurate manner and avoiding loss of information. Apart from Flink, our complete software stack includes a data ingestion and reporting service, fetching data from the GC platform via the gRPC-based API [6], and Apache Kafka [5] as a messaging and persistence service (Fig. 1) that decouples ingestion from data analysis, simplifying their integration.

In designing our solution to the DEBS 2022 GC, we identified the handling of late (out-of-order) events and the mapping between batches of events and window-closings that contribute to them as major correctness challenges. To address them we designed a custom window operator that leverages event semantics to correctly

[1] https://www.infrontfinance.com/

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

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539649

order events and to map event-batches to window-closings. While tuning the performance of our solution, we identified the need to rate-control the data ingestion process (which pulls event-batches off of the GC platform) to ensure a suitable latency-throughput operating point. Addressing this as well, we achieved a solid response to the 2022 GC objectives. While our code parallelizes most operators (including the custom window logic), it maintains a single instance of the batch-unpack logic (as parallelizing this introduces further correctness considerations), eventually limiting the achievable parallelism. We have developed an all-parallel version of our code that allows partitioning the batch-unpack phase as well, but did not manage to performance-tune it in time for submission to the 2022 GC. We nevertheless describe its workings in this paper as an additional design point, part of our ongoing and future work.

2 DESIGN AND IMPLEMENTATION

The proposed architecture of our data analysis pipeline comprises three major components (Fig. 1): (1) the Data Ingestion-Reporting Manager component, a tailor-made Java process that acts as an interface to ingest data from and report query results to the evaluation platform; (2) the Apache Kafka [5] component that is used to decouple the data ingestion/reporting phase from data analysis, and; (3) the stream analytics engine built on top of Flink. Here we describe the design and implementation of each component.

2.1 Data Ingestion-Reporting Manager (DIRM)

The Data Ingestion-Reporting Manager (DIRM) component is a Java process specifically designed to act as an interface with the DEBS'22 GC evaluation platform. It can ingest data and report the query results using the GC-supported gRPC-based API. It is also responsible to report query results back to the GC platform. The GC platform makes data available in batches, identified by an ID assigned by the gRPC service.

The GC platform evaluates the latency of our solution by monitoring the response time of each batch by the time we fetch it through the gRPC API until we report query results for the batch back to the platform. Having data go through an additional process, the DIRM, can increase latency. However, we opt for this design, as decoupling the data ingestion/reporting phase from the data analysis improves portability and interoperability of the solution. The ingestion/reporting component can be extended to fetch data from different data sources (e.g. files) and/or reporting services without affecting the implementation of the rest of the pipeline.

The ingestion and reporting tasks are implemented as separate threads. The ingestion thread fetches and stores data on a Kafka topic, while the reporting thread subscribes to the topics that the analytics task publishes query results (more in Section 2.2).

The data ingestion rate from the GC platform should match the results-reporting rate. Fetching data at a high rate leads to batch queue-up within DIRM, waiting for subsequent analysis. While this could improve throughput as the data analytics job will never be idle waiting for data, an unnecessarily-high fetch rate may overly penalize latency. To handle the latency vs. throughput trade-off, we implemented a rate controller in the data ingestion task of this component. The controller throttles the ingestion rate according to the rate results are generated and reported. We empirically (through repeated measurements and informed parameter settings) achieved a balance between latency and throughput of data analysis in our evaluation runs (more in Section 4.1).

The reporter can also be configured to report results for query 1 or query 2 or both (see Section 3). Finally, the DIRM component keeps track of the total number of ingested batches and the reported queries. When all queries have been reported back to the evaluation platform, the reporting component signals benchmark completion using the GC-supported gRPC API.

2.2 Use of Kafka for asynchronous messaging

Kafka [5] is used to simplify the integration of the ingestion and data-analysis components. It maintains the ingested data before the subsequent analysis, and the query results before the reporter component reports them back to the GC-evaluation platform. The use of Kafka allows us to leverage an existing Kafka connector that is already well integrated with Flink (data ingestion, checkpointing) rather than having to implement such a connector from scratch.

DIRM uses Kafka producers to publish data to a Kafka topic. We opt for the synchronous Kafka producer API (instead of the more performant asynchronous mode) to reduce the window of uncertainty as to the status of published batches in case of DIRM or Kafka crash. Finally, we built our own custom (de)serializers to transfer data objects from and to Kafka topics in binary format.

2.3 Data Processing

The data analytics job comprises a stream-processing graph with multiple operators depicted in Figure 2. These operators a) consume the ingested data; b) unpack events included in batches; c) implement custom window logic to determine the last observed price per symbol in 5-minutes time-frames[2], guaranteeing there will not be dropped late events and window-closing will be correctly associated with event batches; d) calculate the EMA; e) discover crossover events; and e) gather and report the query results on all lookup symbols per batch. Next we describe the design choices and implementation details of each operator.

Source operator. This operator consumes data from Kafka by subscribing to the corresponding topic. We use the Flink-supported Kafka connector[3] as our data source operator. We assume a single instance of this (and the Unpack) operator, processing batches in batch-ID order. We will discuss the impact of this choice in this section and the implications of relaxing it in Section 5.

Unpack operator. Each batch fetched consists of a list of events (trade actions for symbols) and a list of symbols of interest, *lookup symbols*, that a user subscribes to (requests query results for that symbols). The *Unpack* operator extracts and emits events from a batch and also injects metadata on each output tuple (Listing 1). The metadata include the (1) batch ID the event is extracted from; (2) a flag per event symbol that marks if it is included in the lookup list; (3) the number of lookup symbols in the batch; and (4) a flag that indicates if the event is the last occurrence of the symbol in the batch. The metadata are necessary for the subsequent analysis on downstream operators.

[2]We use the term *time-frame* rather than *window* to refer to the different time intervals/ranges whose state may be simultaneously maintained by the window operator
[3]https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/kafka

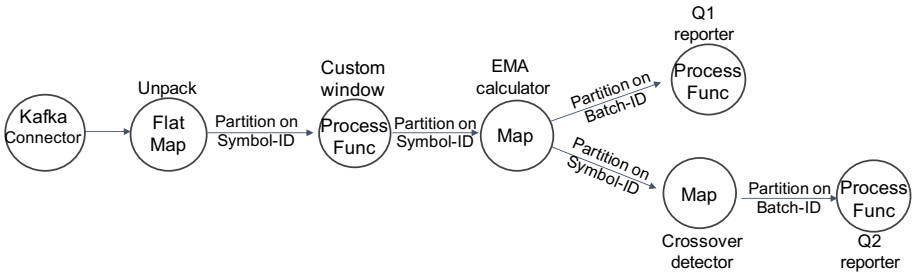

Figure 2: Stream-processing job

```
case class EventUnpackSchema (
    symbol: String,
    securityType: SecurityType,
    Price: Double,
    timestamp: Long,
    batchID: Long,
    isSymbolsLastOccurence: Boolean,
    lookupSymbolBool: Boolean,
    lookupSize: Int,
    isLastBatch: Boolean )
```

Listing 1: Emitted output of unpack operator

Window operator. Following the unpack operator is a window operator that emits the last observed price per symbol in 5-minute time-frames. A major challenge is to handle out-of-order late events, i.e., events that arrive after a window has closed. These events are typically dropped and as a results this could result in correctness issues in the subsequent trend analysis. Flink's built-in window operators support *allowedLateness* option that can accept late events for the specified amount of time when a window closes. However, it is still challenging to predict an appropriate *allowedLateness* value; in the general case, it is not possible to achieve a guarantee that there will not be events that arrive later than the specified setting.

Our goal was to design an application that will not sacrifice correctness over performance. We thus decided to build our own custom window operator that closes a time-frame when, based on event semantics, it determines that there are no events left out that belong to the corresponding 5-minute time-frame. For each symbol our window operator maintains a table of 5-minutes time-frames. Upon the arrival of an event, the window operator has to decide in which time-frame the event is to be assigned according to the event time. Our mechanism performs event grouping and alignment using Equation 1:

$$f(event\_ts) = \lfloor (event\_ts/win\_interval) * win\_interval \rfloor \quad (1)$$

The input to Equation 1 is the timestamp of each processed event. The window interval is set to 5 minutes. The function returns the starting time of the 5-minute time-frame that the event belongs to. For example, event timestamps 14:00:00.001, 14:00:02.421, 14:00:04.343 all belong to the time-frame starting at 14:00:00.000.

For every 5-minute time-frame, the operator maintains the last-price seen for the symbol along with its timestamp. Our custom operator applies incremental processing logic, i.e., it updates the last-price seen of the symbol upon processing an incoming event by comparing if the new event's timestamp is later than the previous stored last-price. Thus we maintain minimal state per time-frame, avoiding buffering of all events within the same time-frame.

The operator also maintains a list of all batches seen and keeps track of the progress of processing each batch, namely whether the operator has processed all events of a batch according to the metadata emitted by the upstream unpack operator. The operator also identifies the 5-minutes time-frames affected by each batch. Specifically, we *link* each batch with the **last time-frame** *affected by the batch*. To do this we use the timestamp of the last occurence of the symbol within the batch (the metadata flag *isSymbolsLastOccurance* in Listing 1). In Figure 3, the last timestamp of symbol ABC in batch B_5 is 12:28, affecting *up to* time-frame 12:25-12:30.

As events are aggregated from multiple sources we cannot assume events for different symbols are timestamp-ordered. However, we assume that events for the same symbol (always ingested from a single source) have monotonically increasing timestamps across batches, a fact that we have validated in the GC data set [2]. Thus, for a given symbol the timestamp of its first event in batch B_i is later than the last timestamp of that symbol in batch B_{i-1}. Based on these ordering properties, for a given symbol, the batch B_i cannot affect a time-frame preceding the time-frame linked to B_{i-1}. For instance, in the example of Fig. 3 for symbol ABC, B_4 cannot affect a time-frame before the one linked to B_3.

When a window operator fully processes a batch, i.e., all events of the batch have been processed, the operator checks if there are *safe-to-close* time-frames to emit the symbol's last price observed in such time-frames. A time-frame is considered as safe-to-close when all batches linked to it and, *at least the first batch linked to the next time-frame*, are completely processed. In the example of Fig. 4, batches B_3 and B_4 are linked with time-frame 12:20-12:25. However, the time-frame is not considered as safe-to-close after the processing of these batches as we are not sure if the next batch contains events that contribute to it. As soon as we have processed the first batch of the next time-frame, i.e., batch B_5, we are sure that the subsequent batches cannot contain events for symbol ABC affecting time-frames preceding the one that B_5 is linked to.

Another challenge for our custom window operator is *empty batches*, i.e., batches that do not contain events for a given symbol (e.g., batch B_1 in Fig. 3 does not contain events for ABC). As the events of each symbol are timestamp-ordered and we use a single instance of the source and unpack operator forwarding events to the corresponding window operator instance, we derive the following

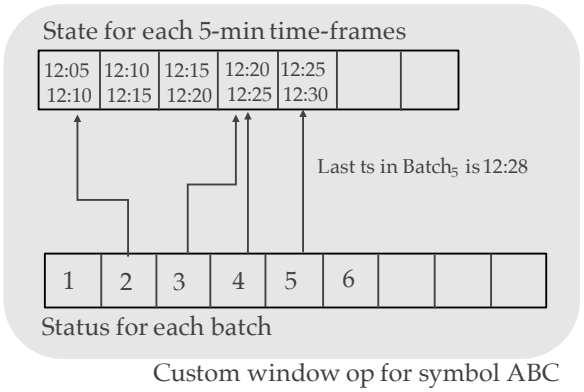

Figure 3: Custom window-operator state for symbol ABC. Each batch B_i, $i = 1, 2, \ldots$ points to the time-frame affected by the last occurrence (last ts) of symbol ABC in that batch

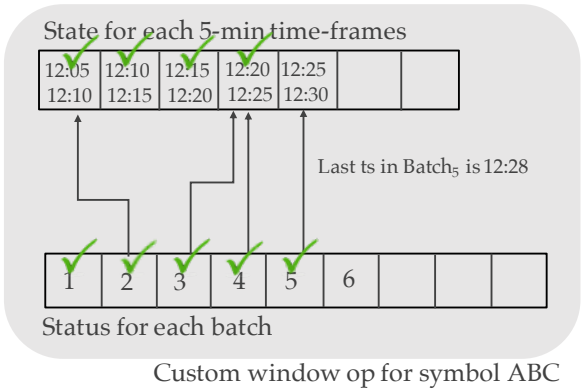

Figure 4: Window closing example: The checkmarks indicate that batches B_1-B_5 have been fully processed and four 5-minute time-frames are safely considered fully closed

property: a batch B_i for which a window operator for symbol ABC has seen no events from, while having seen events from B_{i+1} or later, means that B_i is empty for ABC (e.g. B_1 in Fig. 4).

When a batch is complete, the window operator identifies the time-frame it is linked to and checks if there are pending time-frames that can now be marked as safe-to-close. The example in Fig. 4 illustrates the aforementioned scenario for symbol ABC: The last occurrence of ABC in batch B_2 has its timestamp within 12:05-12:10 (the time-frame is still not considered as safe-to-close when B_2 is fully processed). Batch B_3 is linked to the time-frame 12:20-12:25 (i.e., is the last occurrence of events regarding ABC fall into this time-frame). When B_3 is fully processed we can mark time-frames 12:05-12:10, 12:10-12:15 and 12:15-12:20 as safe-to-close. This is because all events for ABC in subsequent batches are expected to have timestamp later than the last occurrence of ABC in B_3.

The operator emits to its output the closing price of the symbol for each safe-to-close time-frame and purges its state. If there are no events for the symbol associated with a time-frame (e.g. time-frames 12:10-12:15, 12:15-12:20 for symbol ABC in Fig. 4), the operator ignores the time-frame and purges its state. However, if the symbol is in the lookup-symbols list, the operator emits a special-crafted tuple to indicate to the downstream operators that there is no closing price for the 5-minute time-window and thus, they should report the previous EMA (see EMA calculator described next).

When B_4 is completely processed, there are no new safe-to-close time-frames. In this case the custom window operator emits a specially-crafted output tuple for the lookup symbols that indicate that the processing of the batch has completed. These specially-crafted tuples indicate to the downstream operator (the EMA calculator described below) that it can rely on the last computed EMA corresponding the last closed time-window. In the example of Fig. 4 assuming ABC is a lookup symbol, when batch B_4 closes, the time-frame 12:20-12:25 is not safe-to-close, hence it emits a tuple signaling batch completion. That specific time-frame will be marked as safe-to-close only when B_5 is fully processed.

EMA calculator. The last observed price for a 5-minute time-frame emitted by the window operator is necessary for the EMA calculation. There is a separate instance of the EMA calculator per symbol. The EMA calculator operator computes the EMA according to Equation 2.

$$EMA^j_{w_i} = [Close_{w_i} * (\frac{2}{1+j})] + EMA^j_{w_{i-1}} * [1 - (\frac{2}{1+j})] \quad (2)$$

w_i: the 5-minute time-frame
j: the smoothing factor for EMA with $j \in \{38, 100\}$
$Close_{w_i}$: the last price observed within time-frame w_i

The operator computes EMA for a given time-frame for two different j values specified by the user (see Table 1). When a tuple indicates that a batch has completed but no new time-frames have closed, we just fetch the latest EMA for this symbol and pass it to the next operator. Otherwise, on entries indicating that new time-frame(s) have closed, we calculate first the newest EMA(s) and emit the newest results.

Q1 reporter. For the first query of the challenge we have to report the EMAs for all the lookup symbols in a batch. The Q1 reporter operator gathers all computed EMAs for a batch and reports the query result. The output of the EMA calculator is partitioned on the batch ID. The metadata included on each event indicating the total number of lookup symbols in a batch indicated when the Q1 reporter has gathered all the requires EMAs for that batch. When a lookup symbol emits multiple *safe-to-close* time-frames in a batch and therefore multiple results, an indicator points to the latest time-frame result for the reporter to expect.

Crossover calculator. The second query of the GC requires to identify breakout patterns that indicate the start of a trend in the development of a symbol's price. This process is based on the computed EMAs for a symbol over different intervals (i.e., EMA j paramater). The Crossover calculator operator consumes the output of the EMA calculator and discovers crossover events (breakout patterns) as described in the GC. The operator maintains the three most recent breakout events per symbol as required by query 2. Upon detecting a new crossover event, the operator updates its state and discards outdated state.

Q2 reporter. Similar to Q1 reporter, this operator gathers all crossover events for all the lookup symbols in the batch before it

Parameter	Description	Default
p	Parallelism of Flink Application	1
i	Parameter to Calculate EMA	38
j	Parameter to Calculate EMA	100
c	Checkpointing interval (mins)	None
q	Specify the required queries for reporting. 1 for Q1, 2 for Q2	Both

Table 1: Configuration Options

reports the query 2 results. The input stream of the operator is partitioned on the batch ID.

Both Q1 and Q2 reporters also act as sink operator, using a Kafka producer to publish the results to the corresponding Kafka topic.

3 AUTOMATION SCRIPT

Our code repository[4] ships with a configurable deployment and execution management script. The script makes the installation and deployment process of the dependent software components easy. A simple command is enough to install the necessary library dependencies and the software stack (Java runtime, Apache Flink and Apache Kafka).

```
$> ./manage.sh install
```

The management script can also be used to build the submitted software components (DIRM, Analytics application) from source with the following command:

```
$> ./manage.sh build
```

Finally, the execution of the whole analytics pipeline can be invoked using the management script:

```
$> ./manage.sh start
```

However, running the application also supports user defined configuration settings (Table 1) including different smoothing factors for the EMA calculation (parameters *i* and *j* in Table 1) and the reported queries (parameter *q*). Our analytics application also supports scalable deployments (parameter *p*). We also support operation reliability using the Flink checkpointing mechanism (using the checkpointing interval option *c*).

4 EVALUATION

In this section we provide a summary of our experience with how our code performs in the GC evaluation platform. The results of the following section are averages over at least 4 runs with negligible standard deviation. While there is a multitude of possible evaluation dimensions, here we showcase key aspects affecting performance of our implementation.

4.1 Effect of ingestion rate-control (throttling)

As analyzed in Section 2.1 (DIRM) we created a rate-control mechanism as a way to increase throughput via pre-fetching of batches from the gRPC service, and to effectively balance the tradeoff between latency and throughput. Tuning this mechanism demanded extensive evaluation of different parameters. Table 2 shows the impact of different degrees of throttling (number of batches the DIRM reads-ahead from the gRPC service) tested with 5GB of memory and 1 slot (i.e., parallelism is set to 1 for all Flink operators). We observe the tradeoff between latency and throughput in the results. One may choose throttling settings based on specific goals (such as rankings in this GC), and during our evaluation we chose 15 as this seemed to provide the best outcome versus competition. Throttle 10 may have been another good choice as it leads to significantly lower latency with a small impact on throughput. Based on our experience during evaluation trials and a focus on throughput at the time, we have narrowly opted for throttle 15 in our code and use it to conduct the rest of our evaluation tests.

Throttle	Latency (ms)	Throughput (batches/sec)
5	287	27.1
10	328	38.3
15	484	38.8
20	602	39.0

Table 2: Varying degrees of throttle (1 slot, 5GB mem)

Mem (GB)	Latency (ms)	Throughput (batches/sec)
4	503	36.8
5	484	38.8
6	485	38.9

Table 3: Varying memory size (1 slot, throttle 15)

# slots	Latency (ms)	Throughput (batches/sec)
1	484	38.8
2	404	47.3
3	401	46.2

Table 4: Varying parallelism (throttle 15, 5GB mem)

4.2 Effect of memory allocated to Flink

Choosing the most efficient memory to allocate in the Flink component (TaskManager setting) is a challenge when building new applications. Our choice of using 5GB memory in the experiments of Section 4.1 was made based on early experience. The choice is supported by the systematic evaluation shown in Table 3. Setting Flink memory at 5GB achieves the best performance in both latency and throughput compared to 4GB or 6GB (performance with 6GB is practically indistinguishable from that of 5GB). Note that had cost-effectiveness rather than sheer performance been the key criterion here, 4GB would have been a better choice as it leads to a better performance *per GB* ratio in both latency and throughput.

4.3 Effect of parallelism on single TaskManager

After experimenting with the throttling mechanism and different memory configurations, we also tested different parallelism options to obtain the best performance results possible. Our Flink setup currently operates in standalone mode with multiple task slots enabling parallelism of a Flink job within one machine.

For this set of experiments, we utilized the best configuration from the throttling section (throttling 15) and the most effective memory configuration (5GB of memory). Table 4 showcases results for different slot options.

[4] https://github.com/skalogerakis/DEBS_2022_GrandChallenge

Our application performed best when we assigned two task slots to it. Increasing the number of slots beyond that does not yield any additional improvement. This is due to the fact that our source operator is serial, so scaling the rest of the application even more does not improve overall throughput/latency results. Parallelizing the source operator in the way described in Section 5 is expected to further increase performance benefits from parallelism.

5 ONGOING AND FUTURE WORK

After completing and fine-tuning our current solution, we investigated ways to improve even further our implementation while preserving correctness guarantees. As noted in Section 2.3 our designed solution relies on batches arriving in order, a condition made possible by the centralized source operator. An apparent improvement is to parallelize the source operator so as to spread its load over multiple tasks, which should further improve scalability. The differences between the presented solution and a fully parallel implementation are concentrated on two operators, the unpack and the custom window operator.

The main challenge with parallelizing the source operator is that batches will be processed by different instances of the source operator and thus events from them may now arrive downstream out of order. Thus a window operator for symbol ABC may see all events from batch B_i before seeing any event from batch B_{i-1}, due to the fact that batches i and $i-1$ may be processed by different partitions of the source operator and thus events from batch $i-1$ may be delayed. Our custom window operator relies on the fact that time-frames close only when all batches that may be contributing to them have been processed, so delayed arrival of a batch means that certain time-frame(s) will remain open waiting for it.

In our current (serial source operator) implementation, seeing all events from batch B_i before seeing any from B_{i-1} means that there are no instances of symbol ABC in batch B_{i-1}, thus no need to wait for B_{i-1}. In a parallel implementation, this condition may also hold because of a delay in batch B_{i-1}, which may follow a different path (different source/unpack operator partition) compared to B_i. We thus need to determine whether we should wait for events from batch B_{i-1} or no such events exist and we should not expect to account of them in time-frames maintained by the operator.

This can be decided by an efficient set-membership test such as a Bloom Filter (BF) to let operators ask queries such as "is symbol ABC a member of the set of symbols appearing in batch B_i?". As background, BF[5] is a probabilistic data structure used to evaluate whether an element is member of a set. We implemented such a solution in conjunction with an external Redis Database for maintaining BF state.

Every instance of the Unpack operator creates a new bloom filter upon processing a batch. The BF contains all the distinct symbols in that batch and finally the BF is stored in Redis under the current batch Id. We were also careful in the creation of BFs, to minimize the probability of false positives that would lead to erroneous behavior. Every instance of our custom window operator fetches the bloom filter for the missing batches and checks whether it contains the symbol processed by that instance of the window operator. If the BF does not contain the symbol, the operator can safely mark the batch as complete as we don't expect to arrive any event regarding that specific symbol extracted from that batch. Otherwise, the window operator marks the batch as pending and expects to process the late events. In the case that a requested BF does not exist in Redis, which means that it has not been created or stored yet, we handle the batch like it is missing and will be checked again in the future, i.e., when a next batch is fully processed by the operator.

Our current parallel implementation is code complete but could not be thoroughly optimized for performance in time for submission to the 2022 GC. This is a focus of our ongoing and future research work.

Another focus of future work is to further improve the rate-control mechanism (Sections 2.1 and 4.1) by implementing a rate controller that can adapt its ingest rate for maximum throughput at the lowest latency, as the data analytics pipeline scales up or down.

6 CONCLUSIONS

In this paper we report on our team's (Group 14) response to the DEBS 2022 Grand Challenge. We present the design and implementation of a data-analysis pipeline that addresses the GC by leveraging event semantics to correctly handle the mapping between event-batches and window-closings as well as to handle late (out of order) events. We report on the performance we observed in the evaluation testbed along with the impact of key parameters (degree of throttling at the source, memory and task slots allocated to a task manager). Beyond performance, our solution concretely addresses configurability and operational resilience through easy-to-use management scripts and the robustness afforded by the mature data-analysis platforms underlying our solution. We also outline the design and implementation of a fully-parallel version of our data-analysis pipeline, subject of an ongoing evaluation, hoping to soon be able to demonstrate its additional scalability benefits.

ACKNOWLEDGMENTS

We thankfully acknowledge funding by the Hellenic Foundation for Research and Innovation through the STREAMSTORE faculty grant (Grant ID HFRI-FM17-1998) that made this research possible. We would also like to thank the DEBS 2022 Grand Challenge organizers [3] for proposing this exciting challenge and for their support through our implementation and evaluation effort.

REFERENCES

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[2] S. Frischbier, J. Tahir, C. Doblander, A. Hormann, R. Mayer, and H.-A. Jacobsen. 2022. DEBS 2022 Grand Challenge Data Set: Trading Data. https://doi.org/10.5281/zenodo.6382482.

[3] S. Frischbier, J. Tahir, C. Doblander, A. Hormann, R. Mayer, and H.-A. Jacobsen. 2022. The DEBS 2022 Grand Challenge: Detecting Trading Trends in Financial Tick Data. In *Proc. of the 16th ACM Int. Conference on Distributed and Event-Based Systems* (Copenhagen, Denmark) *(DEBS '22)*. 6 pages.

[4] P. J. Kaufman. 2013. *Trading Systems and Methods* (5th ed.). Wiley Publishing.

[5] Jay Kreps, Neha Narkhede, and Jun Rao. [n.d.]. Kafka: a Distributed Messaging System for Log Processing. In *Proc. of 6th International Workshop on Networking Meets Databases (NetDB 2011)* (Athens, Greece, June 12, 2011).

[6] J. Tahir, C. Doblander, R. Mayer, S. Frischbier, and H.-A. Jacobsen. 2021. The DEBS 2021 Grand Challenge: Analyzing Environmental Impact of Worldwide Lockdowns. In *Proc. of the 15th ACM Int. Conf. on Distributed and Event-Based Systems (DEBS '21)*. https://doi.org/10.1145/3465480.3467836

[5] https://en.wikipedia.org/wiki/Bloom_filter