DEBS Grand Challenge: Analysis of Market Data with Noir

Luca De Martini
luca.demartini@mail.polimi.it
Politecnico di Milano
Italy

Alessandro Margara
alessandro.margara@polimi.it
Politecnico di Milano
Italy

Gianpaolo Cugola
gianpaolo.cugola@polimi.it
Politecnico di Milano
Italy

ABSTRACT

The 2022 DEBS Grand Challenge targets the analysis of real-world market data to define a trading strategy that triggers buy/sell advice based on specific temporal patterns. This paper presents a solution to this problem based on Noir, a distributed data processing framework developed at Politecnico di Milano that aims to provide high-level programming abstractions with minimal overhead. Noir abstracts all concerns related to deployment, concurrency, and synchronization, allowing developers to concentrate on the logic of the processing task. It simplifies the definition of a solution and minimizes the time to develop it and make it operational, while at the same time it does not sacrifice absolute performance. Our experiments show that Noir can analyze millions of events per second and deliver answers with a latency of few milliseconds.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; • **Information systems** → **Data analytics**; • **Applied computing**;

KEYWORDS

DEBS Grand Challenge, Noir, dataflow, Rust, data processing, stream processing, big data

1 INTRODUCTION

The 2022 DEBS Grand Challenge [6] targets the analysis of real-world market data provided by Infront Financial Technology[1].

The data set includes one week of trading events on three major exchanges [5], and the challenge uses more than 50 million events in total. Events report the trading price of over five thousand financial instruments. Participants are asked to implement a basic trading strategy that: (i) identifies temporal trends in the price of

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

individual equities by analyzing data in non-overlapping time windows; (ii) triggers a buy/sell advice when the identified temporal trends exhibit specific patterns.

In designing and implementing a solution for this task, and in general a solution for any data-intensive problem [10], there is always an intrinsic tension between generality and absolute performance.

Data processing platforms that offer simple and general programming abstractions to encode a wide range of problems are of great value in industrial settings, as they reduce the time to develop and deploy processing jobs to answer new business needs. This is also acknowledged by the organizers of the DEBS Grand Challenge, who explicitly encourage participants to provide reusable and extensible solutions. Following this need, the research community proposed the dataflow programming model, initially incarnated in the MapReduce framework [3], which rapidly become the standard approach to build general-purpose, distributed data processing platforms [1, 2, 13, 14]. This model expresses jobs as directed graphs of operators, each applying a functional transformation on the input data and feeding downstream operators with its output. It enables data parallelism, as the same task can be executed in parallel on different partitions of the input data, and task parallelism, as tasks may run simultaneously on the same or different machines. The resulting definition of data processing jobs is very concise: developers focus on the behavior of operators and how the input data is partitioned among parallel instances, while the run time automates deployment, scheduling, synchronization, and communication.

On the other hand, current state-of-the-art data processing platforms cannot provide a level of performance that is comparable to custom programs optimized for the specific problem at hand. As recognized in recent literature [7, 12], custom implementations using low-level programming primitives can yield more than one order of magnitude performance improvements. But this comes with a much higher difficulty in software validation, debugging, and maintenance, as programmers are exposed to concerns related to memory management, data serialization, communication, and synchronization [4].

Our research group at Politecnico di Milano is trying to address this apparent dualism between generality, ease of use, and performance by developing Noir [11], a distributed data processing platform that aims to provide the benefits of the dataflow model with very limited overhead. Noir is written in Rust [9], a compiled programming language that offers high-level abstractions at virtually no cost, with a trait system that statically generates custom versions of each abstraction for different data types and avoids dynamic dispatching. In previous work, we built dataflow-based prototypes that could compete with custom MPI programs and deliver up to more than 20× higher throughput than widely adopted open source data processing platforms such as Apache Flink [4].

Noir is the evolution of these prototypes and brings higher expressiveness (e.g., support for nested iterations) and better exploitation of resources (e.g., support for both thread-level and process-level parallelism).

In this paper, we present our experience in using Noir to solve the 2022 DEBS Grand Challenge. We show that Noir enables for a very compact definition of the processing jobs, abstracting all concerns related to deployment, concurrency, and synchronization. At the same time, it delivers a high level of performance, as it can analyze millions of events per second, deliver answers with a latency of few milliseconds, and scale to many cores/machines without any change in job definition.

The paper is organized as follows: Section 2 introduces the programming interface and architecture of Noir; Section 3 presents the 2022 DEBS Grand Challenge problem and the solution we implemented in Noir; Section 4 evaluates our solution; Section 5 draws concluding remarks.

2 NOIR IN A NUTSHELL

This section provides a high-level view of Noir, introducing its programming interface (Section 2.1) and the main implementation strategies that contribute to its performance (Section 2.2).

2.1 Programming interface

The core data structure in Noir is the stream, `Stream<T>`, representing a (bounded or unbounded) data set of elements of type `T`. Streams are created from *sources*, for instance a file or a TCP connection; they are processed by *operators* that take in input one or more streams and apply functional transformations to produce one or more output streams; they are collected by *sinks*, for instance a file or a database, which store the final results of a sequence of transformations. Streams can be split into independent *partitions* that the platform can process in parallel if enough computational resources are available.

The example below showcases the programming interface using the classic *word count* example, which counts the number of occurrences of each word in a large document.

```
let result = Stream::from_readlines(&file)
  .flat_map(|line| tokenizer.tokenize(line))
  .map(|word| (word, 1u32))
  .group_by(|(word, _count)| word.clone())
  .reduce(|(_word1, count1), (word2, count2)| (word2, count1+count2))
  .collect_vec();
```

First, `from_readlines` creates a source that reads from the file passed as parameter and produces a `Stream` of strings, where each element is a line in the file; `flat_map` transforms the input stream of lines into an output stream of words, that is, it splits each line into words using the function passed as parameter; `map` converts each word into a pair (word, count), where count is the number of occurrences of that word and is initially set to one; `group_by` groups the data set by word, meaning that the subsequent operators get applied to each group (that is, to each word); `reduce` sums all the counts for each word; `collect_vec` is a sink that stores all results in memory (in a vector).

Despite its simplicity, the example well illustrates the high level of abstraction provided by the Noir programming interface, which

completely hides details about serialization, communication, concurrency, and synchronization. In fact, the example can be executed in parallel exploiting the processing cores of a single or multiple machines by simply changing few lines in a configuration file.

Noir includes a rich library of operators, including stateful operators that accumulate an internal state across invocations, split and join operators to create parallel branches of computation and merge them. It supports (nested) iterations, timestamps, several types of windows and aggregation functions. Moreover, it offers a simple API to create additional, custom operators at need.

2.2 Architecture

Internally, Noir converts each job into a dataflow graph of operators and organizes them into *stages*. A stage includes contiguous operators that do not alter the way in which data is partitioned.
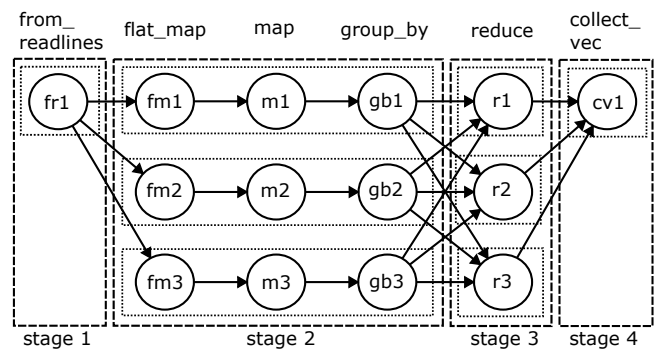

Figure 1: Dataflow graph for the word count example.

Figure 1 depicts the dataflow graph and the division in stages for the word count example presented in Section 2.1. As Figure 1 shows, the graph has four stages, represented as dashed blocks. The first one includes the source, which reads sequentially from a file. The second stage starts from the `flat_map` operator, which consists of multiple instances processing lines in parallel, and terminates with the `group_by` operator, which partitions data elements by word. The third stage includes the `reduce` operator. The last stage includes the final sink, which collects all data into a single location.

Each stage consists of one or more parallel instances. In the example in Figure 1, blocks are represented as dotted boxes: the first and the last stages consist of a single block, while the second and the third stages consist of three blocks.

Blocks are the unit of deployment. By default, Noir instantiates a single process on each machine available in the computing infrastructure, and allocates one block of each stage to each processing core (on the same or on different machines). This way, blocks of different stages compete for processing resources and their scheduling is handled by the operating system with no additional overhead.

Operators within each block are executed sequentially. The communication between stages takes place through in-memory Rust channels[2] (if the communicating blocks are on the same machine) or through TCP channels (if the communicating blocks are on multiple machines).

---

[2]https://doc.rust-lang.org/book/ch16-02-message-passing.html

Developers can configure the *batching* policy for each communication channel. Larger batches accumulate more data elements before transmitting them over the channel: this choice tends to increase throughput at the cost of a higher latency. If latency is important, developers can reduce the batch size or set a maximum timeout per channel.

3 IMPLEMENTING THE CHALLENGE

This section presents the problem proposed in the DEBS Grand Challenge (Section 3.1) and the solution we implemented in Noir (Section 3.2).

3.1 Problem definition

The 2022 DEBS Grand Challenge requires analyzing financial data to implement a simple trading strategy [6]. The data set used for the evaluation derives from real world data provided by Infront Financial Technology [5] and consists of almost 60 million trading events collected from three major exchanges in a week. Events include five attributes: (1) symbol is a unique string identifying a financial product and the respective exchange; (2) sec is the security type, which can be either equity or index; (3) price is the last trade price; (4) date is the date of the last trade; (5) time is the time of the last trade. The date and time attributes taken together form the timestamp of the event.

The challenge defines two queries. The first one computes quantitative indicators for each symbol, which are used in financial analysis to identify trends. Specifically, the query computes *exponential moving averages* (EMAs), which are defined (recursively) for each symbol as follows:

$$EMA^j_{w_i} = close_{w_i} \cdot \frac{2}{1+j} + EMA^j_{w_{i-1}} \cdot (1 - \frac{2}{1+j})$$

Where w_i identifies the ith time window: in the challenge, windows are five minutes long and non-overlapping (tumbling); $close_{w_i}$ is the closing price for the symbol within window w_i, i.e., the price of the last event received within w_i; j is the smoothing factor: the query in the challenge computes *EMAs* with $j = 38$ (EMA^{38}) and with $j = 100$ (EMA^{100}).

The second query looks for patterns in the EMA^{38} and EMA^{100} indicators to generate buy/sell advice. In particular, the query triggers a buy advice for a symbol when EMA^{38} overtakes EMA^{100} for that symbol, that is when:

$$EMA^{38}_{w_i} > EMA^{100}_{w_i} \wedge EMA^{38}_{w_{i-1}} \leq EMA^{100}_{w_{i-1}}$$

Symmetrically, the query triggers a sell advice for a symbol when EMA^{100} overtakes EMA^{38} for that symbol, that is when:

$$EMA^{38}_{w_i} < EMA^{100}_{w_i} \wedge EMA^{38}_{w_{i-1}} \geq EMA^{100}_{w_{i-1}}$$

The challenge provides an evaluation platform that exposes a gRPC-based API to receive input events and to submit the query results computed.

Input events are organized in *batches*. Each batch contains a sequence of events from the input data set and a set of symbols the evaluation platform subscribes to.

As response to the first query, the evaluation platform expects, for each batch and for each symbol it is subscribed to, the latest EMA^{38} and EMA^{100} indicators.

As response to the second query, the evaluation platform expects, for each batch and for each symbol it is subscribed to, the last three sell/buy advice.
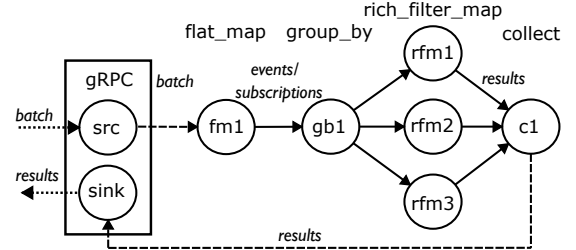
3.2 Solution in Noir

Figure 2: Implementation of the 2022 DEBS Grand Challenge in Noir: architecture of the solution.

Figure 2 shows the architecture of the solution we implemented in Noir. A gRPC module handles the communication between the evaluation platform and the main Noir engine. It consists of a gRPC source (src in Figure 2) that fetches input batches from the evaluation platform and a gRPC sink that submits query results to the evaluation platform. Figure 2 shows gRPC communication channels as dotted lines. The gRPC module communicates with Noir via flume asynchronous channels[3] (represented as dashed lines in Figure 2).

The core of the dataflow graph used in Noir to solve the challenge consists of four operators. A flat_map splits each batch b into its constituting parts and propagates downstream all events in b, followed by all subscriptions in b. Events and subscriptions are partitioned by symbol (group_by), which enables the subsequent rich_filter_map to process different symbols in parallel (Figure 2 shows three parallel instances as an example).

A rich_filter_map is a default operator in Noir that accumulates state (hence the name rich) and produces either zero (filter) or one (map) output elements for each input element. We use the state in the operator to store (i) the information required to compute the EMA^{38} and EMA^{100} indicators for each symbol: the value of EMA^{38} and EMA^{100} in the previous window and the last price in the current window; (ii) the last buy/sell advice per symbol. Upon receiving an event, the operator updates its internal state without producing any result. Upon receiving a subscription, the operator emits the current results for the two queries.

Finally, a collect operator collects the results produced for the various symbols and submits them back to the gRPC sink. As the gRPS src and sink are part of the same module, they know the number of results to expect for each batch (that is, the number of subscriptions within that batch), so the sink can determine when all results for a batch have been received and can be submitted to the evaluation platform.

All components except the gRPC module were already present in the standard Noir library, and the gRPC module can be easily reused. In fact, when approaching the challenge, we observed that the

---

[3]https://github.com/zesterer/flume

communication with the evaluation platform could easily become a bottleneck. Hence, we payed particular attention in engineering the gRPC module. In the final solution, both the gRPC `src` and the gRPC `sink` open parallel connections with the communication platform and handle the communication over these connections using asynchronous tasks. Both the number of connections and the number of threads processing the tasks in parallel can be set as configuration parameters: in the evaluation platform, we used 6 input connections and 4 threads.

Order. Input events may be received out of (timestamp) order, as events in the data set were originally produced from multiple sources not guaranteed to be synchronized with each other. The evaluation platform does not produce any watermark, i.e., we can never know if further elements with a lower timestamp will ever be produced. Our implementation closes a window w for a given symbol s upon receiving for the first time an event with symbol s and timestamp larger then the closing time of w. For each window, it computes the EMA^{38} and EMA^{100} indicators with the latest (in timestamp order) price received before closing the window.

Visualization. A bonus request for the challenge was a smart visualization of the results. We implemented it as an optional feature using standard components for data management, analysis, and visualization.

When the feature is enabled, the parallel `rich_filter_map` operators submit the EMA^{38} and EMA^{100} indicators to a Redis[4] in-memory database, using the RedisTimeSeries module[5]. The communication with Redis happens over TCP channels, so Redis can be installed on any machine. The communication is handled as an asynchronous task: in Section 4 we measure the overhead of adding this feature.
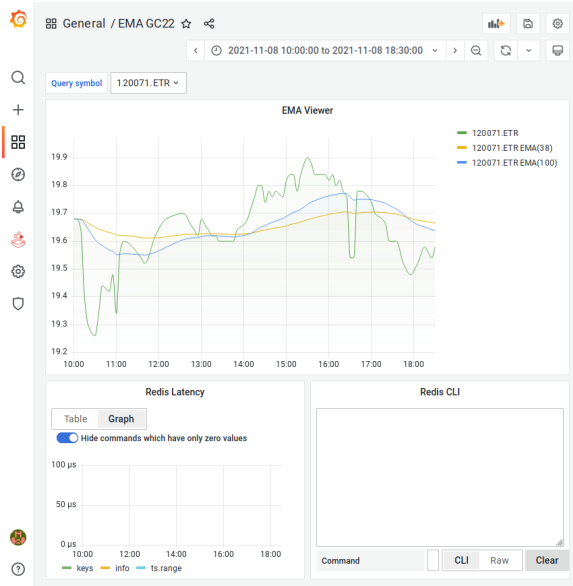

Figure 3: Smart visualization feature using Grafana.

---

[4]https://redis.io
[5]https://redis.io/docs/stack/timeseries/

The actual visualization is implemented as a dashboard in Grafana[6], which shows the temporal evolution of the EMA^{38} and EMA^{100} indicators for each symbol within one graph. Figure 3 shows a screenshot of the dashboard.

4 EVALUATION

Participants to the 2022 DEBS Grand Challenge had the opportunity to deploy their solution onto one or more machines located in the same cluster as the evaluation platform. We conducted several experiments on this platform, and we report our main findings in Section 4.1. To better assess the scalability of our solution, we also deployed it on alternative infrastructures and tested more financial indicators, as we discuss in Section 4.2. Finally, Section 4.3 studies the overhead of smart visualization.

4.1 Grand Challenge evaluation platform

To assess the performance of our solution, we measure the total execution time and the latency per batch. The total execution time indicates how long a given configuration takes to fully process all the batches submitted by the evaluation platform: the input data set consists of 5940 batches, each containing 10 k events (for a total of about 59.4 million events) and subscriptions to 15 different symbols. We measure the total execution time from the moment when we submit the first gRPC request to the moment when we submit the query results for the last batch. The throughput of the system can be derived by dividing the total execution time by the overall number of events.

The latency per batch is the difference between the time when we receive a batch and the time when we submit the query results for that batch. For each experiment, we report the 10th percentile, the median, and the 90th percentile latency per batch. This measure differs from the latency observed by the benchmarking platform used in the Grand Challenge, as it does not include network latency.

Figure 4 shows how the performance of our solution changes when increasing the amount of parallelism, i.e., the number of blocks Noir allocates for the operators that compute the indicators (operator `rich_filter_map` in Figure 2).

The machine provided in the platform had an Intel Haswell CPU with 4 cores, so we changed the level of parallelism from 1 to 4. As Figure 4 shows, neither the overall execution time (Figure 4a) nor the latency per batch (Figure 4b) change significantly when increasing the parallelism. We see two motivations for this: (i) reading data from the network represents the main bottleneck; (ii) the queries are not computationally expensive, as they only require storing the latest value for each window and performing a simple arithmetic computation when a window closes.

We investigate the second hypothesis in Section 4.2 by using alternative indicators and deployment infrastructures. To validate our first hypothesis, we study the performance of our solution while changing the number of parallel gRPC connections used to fetch input data (with a fixed parallelism of 4). As Figure 5a shows, the execution time decreases when we increase the number of connections, and becomes stable with about 5 connections: at this point, Noir is processing data at about 2.5 Gbit/s and this appears to be the maximum we can obtain regardless of other configuration parameters.

---

[6]https://grafana.com
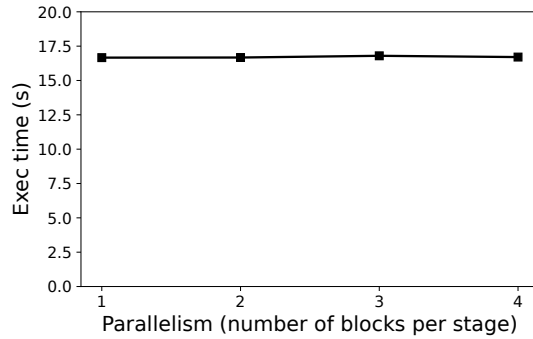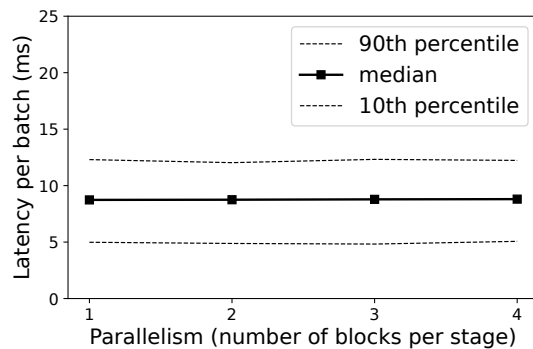
(a) Execution time

(b) Latency

Figure 4: Effect of parallelism.

(a) Execution time

(b) Latency

Figure 5: Effect of number of gRPC connections.

We suspect we reached the maximum bandwidth of the network interface. While the execution time decreases, Figure 5b shows that the latency per batch increases with the number of connections: this is because multiple batches are processed in parallel and the time to collect all query results for each individual batch increases. The final solution we submitted for the DEBS Grand Challenge adopts 6 connections, which delivers the maximum throughput with a limited cost in terms of latency.
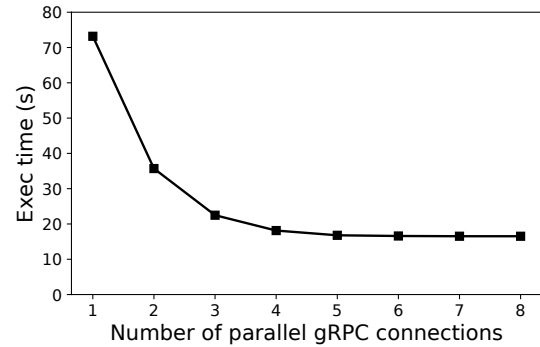
4.2 Scalability

To study the scalability of our solution, we (i) implemented additional, more computationally expensive financial indicators; (ii) deployed Noir on alternative infrastructures that offer a higher number of processing cores. We report the results we measured in Figure 6: in these experiments, we removed the bottleneck of network communication and serialization, simulating ingestion by artificially generating input data[7]

As additional indicators we implemented Moving Average Convergence Divergence (MACD)[8] and an indicator based on three simple moving averages (SMA)[9]. Figure 6 presents both the results
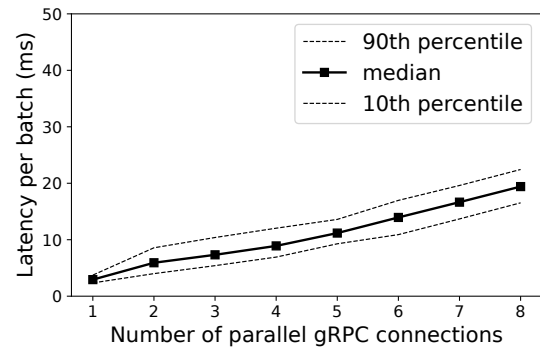
---

[7]We used the same symbols, batch size, and number of subscriptions as in the DEBS Grand Challenge evaluation platform, but generated data randomly, so the results may differ from those of the DEBS Grand Challenge data set.
[8]https://www.investopedia.com/terms/m/macd.asp
[9]https://www.investopedia.com/terms/s/sma.asp

when using only the DEBS Grand Challenge indicators (`GC-22 indicators`) and the results when using also the additional indicators (`Extended indicators`).

We run our experiments on the Amazon EC2 cloud infrastructure, using a M5n instance offering 16 virtual CPU cores and 64 GB or RAM (Figure 6a) and on a local laptop equipped with a M1 Max CPU with 10 cores and 32 GB of RAM.

In the experiments, we change the level of parallelism, i.e., the number of blocks used to process events in parallel, and we measure the average execution time per batch by dividing the total execution time by the number of input batches. We run each experiment for at least 50 s and we discard the first 5 s to make sure that the system is in a steady state.

On both deployments, the execution time per batch decreases when moving from a parallelism of 1 to a parallelism of 4, but then it stabilizes or even slightly increases when using a higher parallelism. In both cases, the use of additional indicators adds computational complexity, but Noir handles this complexity by using the available resources: as a results, it delivers the same performance with the Grand Challenge indicators and with the extended indicators with a parallelism of 4 or higher.

In absolute terms, as each batch contains 10 k events, Noir is processing over 10 million events per second in the Amazon EC2
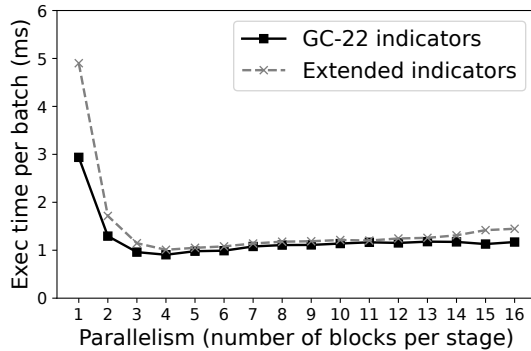
(a) Amazon EC2 (M5n instance, 16 vCPUs)
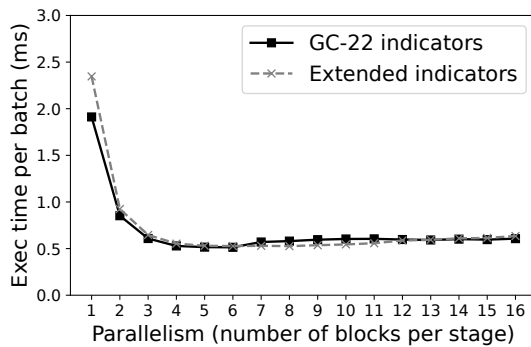
(b) Local (M1 Max, 10 cores)

Figure 6: Scalability with different workloads and deployment infrastructures.

deployment and almost 20 million events per second in the local deployment.

4.3 Overhead of smart visualization

The results presented so far do not include smart visualization. We implemented this feature by storing indicators as time series in the Redis in-memory database, so its cost is ultimately determined by the performance of Redis, and in particular by the rate at which it accepts new values.

We repeated the experiments on Amazon EC2, deploying Redis on the same instance as Noir. In this setting, the processing time was largely dominated by the time required to ingest new data in Redis, leading to an average execution time per batch of about 19 ms (about 500 k events per second) with smart visualization.

This means that the overall throughput decreases by almost 20× as writing into the database become the bottleneck. Indeed, the data set contains one week of events and two indices need to be updated every five minutes, leading to over 4 k updates per symbol. As there are more than 5 k symbols, we need to write about 20 million values to the database: although Redis can sustain over 100 k updates per second, it still remains the main bottleneck of the system. We plan to investigate alternative solutions to store time series that may sustain a higher ingestion rate [8].

5 CONCLUSIONS

This paper presented a solution to the 2022 DEBS Grand Challenge based on Noir, the distributed batch and stream processing platform we are developing at Politecnico di Milano [4, 11].

Noir aims to offer a high level of abstraction and ease of use, comparable to that of state-of-the-art data processing systems, while reducing as much as possible the overhead with respect to custom solutions. It is implemented as a library in Rust that developers can use to compile highly optimized parallel and distributed programs for data analysis.

In this paper, we provided a high-level view of Noir and we discussed how we implemented the 2022 DEBS Grand Challenge on top of it. We analyzed the performance of our solution using different deployment infrastructures, configuration parameters, and workloads. Noir could process over ten million events per second on a single machine, processing one week of financial data coming from three major exchanges within few seconds.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proceedings of VLDB Endow.* 8, 12 (2015), 1792–1803.

[2] 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 Engineering Bulletin* 38, 4 (2015), 28–38.

[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[4] Alessio Fino, Alessandro Margara, Gianpaolo Cugola, Marco Donadoni, and Edoardo Morassutto. 2021. RStream: Simple and Efficient Batch and Stream Processing at Scale. In *2021 IEEE International Conference on Big Data (Big Data '21)*. IEEE, 2764–2774. https://doi.org/10.1109/BigData52589.2021.9671932

[5] Sebastian Frischbier, Jawad Tahir, Jawad 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

[6] Sebastian Frischbier, Jawad Tahir, Jawad 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.

[7] Patricia González, Xoán C. Pardo, David R. Penas, Diego Teijeiro, Julio R. Banga, and Ramón Doallo. 2017. Using the Cloud for Parameter Estimation Problems: Comparing Spark vs MPI with a Case-Study. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*. IEEE Press, 797–806.

[8] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.

[9] Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language.

[10] Alessandro Margara, Gianpaolo Cugola, Nicoló Felicioni, and Stefano Cilloni. 2022. A Model and Survey of Distributed Data-Intensive Systems. https://doi.org/10.48550/ARXIV.2203.10836

[11] Edoardo Morassutto and Marco Donadoni. 2021. Noir : design, implementation and evaluation of a streaming and batch processing framework. http://hdl.handle.net/10589/180143

[12] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. 2015. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science* 53 (2015), 121–130. INNS Conference on Big Data.

[13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '13)*. ACM, 423–438.

[14] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.