RStream: Simple and Efficient Batch and Stream Processing at Scale

Alessio Fino
Politecnico di Milano
alessio.fino@mail.polimi.it

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

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

Marco Donadoni
Politecnico di Milano
marco.donadoni@mail.polimi.it

Edoardo Morassutto
Politecnico di Milano
edoardo.morassutto@mail.polimi.it

***Abstract*—Distributed data processing platforms aim to provide a balance between ease of use and performance. The question is: do they succeed? Systems like Apache Spark or Apache Flink offer a high-level programming model that results in simple and concise definition of the processing tasks, abstracting away most of the concerns associated to concurrency and distribution but at the cost of a large performance gap with custom programs that use low-level primitives to control distribution and resource usage. May we fill this gap? May alternative design choices yield better performance without sacrificing simplicity?**

This paper answers the above questions by introducing RStream, a novel data processing platform written in Rust. RStream provides a high-level programming model similar to that of mainstream data processing systems, which supports batch and stream processing, data transformations, grouping, aggregation, iterative computations, and time-based analytics, incurring in a much lower overhead, closer to that of custom, low-level code. In numerical terms, our evaluation shows that RStream programs present nearly identical complexity as similar programs written in Flink, delivering from $2\times$ to $20\times$ the throughput of Flink, rivaling custom MPI implementations.

***Index Terms*—batch processing, stream processing, dataflow**

I. Introduction

As data guides the decision making process of increasingly many activities, analytics over static (batch) and dynamic (streaming) data becomes a core component in the software stack of many companies. The tension between simplicity in the definition of the processing tasks, and efficiency and scalability of their execution, guided the advent of a programming model that promotes distributed processing while hiding most of the burdens it brings. Introduced with MapReduce [1], 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. This dataflow approach offers task parallelism by enabling different operators to run simultaneously on the same or different machines. It also offers data parallelism by launching parallel instances of operators, each one working on an independent partition of the input data. The resulting job definitions are very concise: developers focus on the behavior of operators and how the input data is partitioned among parallel instances, while the runtime automates deployment, scheduling, synchronization, and communication. Over the years, various systems perfected the model, for instance introducing iterative computations and better exploiting memory management [2]. Remarkably, they proved the programming model capable of unifying batch and stream processing under a common abstraction [3], [4], [5].

Despite these significant achievements, current state-of-the-art data processing systems cannot provide a level of performance that is comparable to custom programs optimized for the specific problem at hand. As recognized in recent literature [6], [7], custom implementations using low-level programming primitives, such as MPI, 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.

This state of things leads to the research questions we address in this paper. *Is the existing performance gap with custom low-level code only the inevitable result of the programming model employed by modern data processing systems? Or do other design choices play a role?*

We answer these questions by presenting RStream, a lightweight distributed data processing platform. RStream provides a unifying model for batch and stream processing, with primitives for data transformation, partitioning, aggregation, and support for iterative computation. This model is similar to that provided by state-of-the-art distributed processing systems. In fact, a direct comparison with programs written for Flink [5] or Spark [2] shows nearly identical complexity. At the same time, RStream delivers performance closer to custom MPI implementations. In our experiments we measured from $2\times$ to $20\times$ higher throughput than Flink.

This better compromise between performance and ease-of-use is the result of various design choices. First, RStream abandons the JVM-based languages, typically adopted by mainstream competitors, in favor of Rust [8], a compiled programming language that offers high-level abstractions at virtually no cost, with a trait system that statically generates

978-1-6654-3902-2/21/$31.00 ©2021 IEEE

custom versions of each abstraction for different data types and avoids dynamic dispatching. Second, RStream adopts a lightweight approach to resource management, which leverages the services offered by the operating system as much as possible. For example, RStream co-locates operators that perform different steps of a processing job on the same machines, letting them compete for CPU time, based on their dynamic requirements, while it leverages the mechanisms embedded into TCP to implement backpressure.

The paper is organized as follows. Section II provides background on distributed data processing systems and Rust. Section III and Section IV present the programming model and the design of RStream, and Section V evaluates its performance and scalability, comparing them with Flink and custom MPI programs. Section VI discusses related work and Section VII draws conclusive remarks.

II. Background

This section presents the programming model of distributed data processing platforms and the key features of Rust that RStream exploits to attain simplicity and efficiency.

A. Distributed data processing

Modern platforms for distributed data processing rely on a dataflow programming model first introduced by Google's MapReduce [1]. Computation is organized into a directed graph of operators, whose edges represent the flow of data from operator to operator. Since operators do not share any state, the model promotes distribution and parallelism by deploying operators in multiple instances, each processing an independent partition of the input data and running in parallel with the others, on the same or on different machines.

The famous example used to illustrate the model is "word count", a program to count the number of occurrences of each word in a large set of documents. It can be expressed using two operators, the first operates in parallel on various partitions of the input documents splitting them in words and emitting partial counts for each word. These partial results are then regrouped by word and passed to the second operator that sums the occurrences of each word. Developers need only to express how to operate on an individual document (first operator) and how to integrate partial results for each word (second operator). The runtime takes care of operator deployment, synchronization, scheduling, and data communication: the most complex and critical aspects in distributed applications.

The dataflow model accommodates stream processing computations with only minor adjustments. Due to the unbounded nature of streams, developers need to specify when certain computations are triggered and what is their scope, which is typically expressed using *windows*. For instance, developers could implement a streaming word count computation over a window of one hour that advances every ten minutes, meaning that the count occurs every ten minutes and considers only documents produced in the last hour.

Data processing systems implemented the dataflow model using two orthogonal execution strategies. Systems such as Hadoop [9] and Apache Spark [2] dynamically schedule operator instances over the nodes of the compute infrastructure. Communication between operators occurs by saving intermediate results on some shared storage, with operators deployed as close as possible to the input data they consume. Other systems such as Apache Flink [5], and Google Dataflow [4] deploy all operators instances before starting the computation. Communication takes place as message passing among instances. RStream adopts the second strategy, which enables lower latency for streaming computations, as it does not incur the overhead of operator scheduling at runtime.

B. Rust

RStream heavily relies on some key features of the Rust programming language to offer a high-level API with limited performance overhead.

1) Generics and static dispatch: In Rust, developers can express data structures and functions that are *generic* over one or more types. For instance, all RStream operators consume and produce a generic `Stream<T>`, which represents a bounded or unbounded dataset of a generic type `T`. This high-level construct is implemented at virtually no cost by Rust, which adopts static dispatching. The compiler generates a separate version of each generic structure or function for each different way in which it is instantiated in the program, while invocations to generic functions are translated into direct calls to the correct version [10].

2) Memory management: Rust provides automatic and safe deallocation of memory without the overhead of garbage collection. It achieves this goal through an *ownership and borrowing* model [11], which certainly represents Rust's most distinctive feature. In Rust, every value has an *owning scope* (for instance, a function), and passing or returning a value transfers its ownership to a new scope. When a scope ends, all its owned values are automatically destroyed. A scope can lend out a value to the functions it calls: the Rust compiler checks that a lease does not outlive the borrowed object. All together, this model allows Rust to fully check safety of memory accesses at compile time, also avoiding the need for (costly) runtime garbage collection.

3) Iterators and closures: The iterator pattern is heavily used in idiomatic Rust code and enables chaining operations over a collection of items without manually implementing the logic to traverse the collection. Operations are implemented as *iterator adapters* that take in input an iterator and produce a new iterator. Rust provides *closures*, which are anonymous functions that can capture the enclosing environment. Iterator adapters are often higher-order functions that accept closures defining their behavior as parameters. The iterator pattern strongly resembles the dataflow model discussed above. For this reason, we used iterators as the blueprint for RStream's model and implementation, making its API intuitive both for Rust developers and for users of data processing platforms.

4) Traits and serialization: Traits represent a collection of functionalities (methods) that any data type implementing that trait should offer. Traits are widely used in Rust to bound

generics, for instance to restrict the use of a generic function only to parameters that implement certain traits. RStream leverages traits to transparently implement parameter passing among distributed instances of operators. More specifically, RStream requires all data types to implement the `Serialize` and `Deserialize` traits.

III. Programming Interface

RStream offers a high-level programming interface that hides most of the complexities related to data distribution, communication, serialization, and synchronization.

A. Streams

Streams are the core programming abstraction of RStream. A generic `Stream<T>` represents a dataset of elements of type `T`, which can be of any type that implements the `Serialize` and `Deserialize` traits. Since these traits can be automatically derived at compile time by the Serde library [12], developers can use their custom data types without manually implementing the serialization logic. Streams model both static (bounded) datasets (e.g., a file) and dynamic (unbounded) datasets, where new elements get continuously appended (e.g., a TCP link). Streams are created by *sources*, processed by *operators* that produce output streams from input streams by applying functional transformations, and collected by *sinks*. Finally, Streams can be partitioned, enabling those partitions to be processed in parallel.

1) Creating and consuming streams: Streams are built from a source of data elements. RStream provides two methods to build a source. The `Stream::new` method takes an iterator and creates a source that produces all elements returned by the iterator: in the example below, all integers from 0 to 100.

```
let stream1 = Stream::new(0..100);
```

Similarly, the `Stream::new_parallel` method creates a source that consists of multiple instances producing elements in parallel. It takes a closure with two input parameters: `r` is the rank of the current source instance, while `size` is the total number of instances to create. The rank is an integer identifier assigned to each instance, going from 0 to `size-1`. The closure must return an iterator for each rank, to be associated with the corresponding instance. In the example below, each instance produces 10 integers, the first one starting from 0, the second one starting from 10, and so on. Source instances will be deployed on different processes (see Section IV).

```
let stream2 = Stream::new_parallel(|r, size| {
   let iter = (10*r)..(10*(r+1));
   Box::new(iter) });
```

The use of iterators is widespread in Rust libraries, for instance it is used by the Apache Kafka API for Rust, easing the job of RStream programmers.

Job execution starts when a sink consumes a stream. For instance, the two sources above would not produce any element until a sink demands so. RStream provides four sinks: `for_each` and `parallel_for_each` apply a function to each and every element in the stream, either sequentially or in parallel (in multiple processes), `collect_vec` gathers all elements in a vector, `reduce` applies a global reduction over all elements of the stream. For example, the following code snippet prints all elements in the stream:

```
Stream::new(0..100).for_each(|i:u32| {
   println!("{}",i); });
```

Likewise, the code snippet below sums all elements in the stream, resulting in a single number.

```
Stream::new(0..100).reduce(|i: u32,j: u32| i + j);
```

2) Transforming streams with operators: Operators transform a stream into a new stream. Developers encode operators' behavior in closures. Examples of operators are `map`, `flat_map`, and `filter`. A `map` transforms each element of the input stream into one element of the output stream. For instance, the following code snippet transforms a stream of integers doubling each element to produce the output stream.

```
stream.map(|i: u32| i * 2);
```

A `flat_map` operator is similar to a map, but can generate zero, one, or more elements in the output stream for each element in the input stream. For instance, for each integer `i` in the input stream, the following code outputs three integers: `i`, `i` multiplied by 2, and `i` multiplied by 3. The developer packs the output elements produced when processing an input element into a vector and RStream automatically integrates all results into the output stream.

```
stream.flat_map(|i: u32| vec![i, i * 2, i * 3]);
```

A `filter` operator takes a predicate and retains only the input elements that satisfy it. For instance, the code snippet below retains only the even numbers from the input stream.

```
stream.filter(|i: u32| v % 2 == 0);
```

3) Partitioning and parallelism: RStream assumes that operators process each element in their input stream independently from the others. Under this assumption, stream elements can be processed in parallel by different operator instances that are executed, at runtime, by separate processes, possibly launched on different machines (see Section IV).

However, in some cases developers may want to retain some common state for a group of elements in a stream, for instance to count the number of elements in each group. To support these scenarios, RStream allows developers to explicitly control stream partitioning with two operators: `key_by` and `group_by`. Both take in input a closure that computes a *key* for each element in the stream and repartition the stream to guarantee that all elements having the same key will belong to the same partition. This way, developers can retain the state associated to a given key in a single operator instance, with the guarantee that this instance will receive all elements with that key. Keys can be of any type that implements the `Hash` and `Eq` traits. The two operators differ in that `key_by` admits any downstream operator, whereas `group_by` allows only aggregation operators (e.g., `reduce`) that will work independently on each key. For instance, the code below

organizes an input stream of integers in two groups, even and odd, by associating each element with a key that is 0 for even numbers and 1 for odd numbers. Then, it sums all elements in each group. The result will be a stream of two elements, representing the sum of all even and all odd numbers in the original stream.

```
stream.group_by(|i: u32| i % 2).reduce(|x, y| x + y);
```

Even though RStream shares with data processing platforms the philosophy of hiding the details about concrete deployment and execution as much as possible, it also provides few constructs to control parallelism. The `max_parallelism` operator sets the maximum number of partitions allowed for the output stream. The `shuffle` operator evenly redistributes elements across partitions: for instance, after a `key_by` and a chain of transformations partitioned by key, it might be beneficial to reshuffle data when subsequent operators do not need to preserve key partitioning anymore.

B. Iterations

Several algorithms for data analytics are iterative in nature. For instance, many machine learning algorithms iteratively refine a solution until certain quality criteria are met. RStream supports iterative computations with two operators.

The `iterate` operator repeats a chain of operators until a terminating condition is met or a maximum number of iterations is reached. In the first iteration, the chain consumes elements from the input stream, while at each subsequent iteration, the chain operates on the results of the previous iteration. The `iterate` operator accepts three input parameters: a closure to reduce all elements in the dataset, a closure that evaluates the result of the reduction and returns a Boolean to indicate if a new iteration is needed, and the maximum number of iterations. For instance, the following code snippet repeats the `map` operator until the sum of all the elements in the stream overcomes 1000 or after 10 iterations.

```
stream.map(|i| i + 1).iterate(
   |x: i32, y: i32| x + y), // Reduce
   |ris| ris > 1000,        // Stopping criterion
   10);                     // Max num of iterations
```

The `cycle` operator supports algorithms that need to iteratively update some mutable state. As the `iterate` operator, `cycle` repeats a chain of operators until a terminating conditions is met. However, each iteration receives the same input stream and the current value of a state variable. At the end of each iteration, `cycle` updates the value of the state variable: this update is performed on a single instance, which broadcasts the new value before starting a new iteration. The following snippet exemplifies the use of the operator.

```
let state = State::from(0u32); // Initialize state
Stream::new_cycle(
   source,         // Source: any iterator
   state.clone(), // State
   0               // Maximum number of iterations
).map(|i| {        // Any chain of operators
  println!("Current state: {}", state.borrow());
  i                // Only prints the current state
}).cycle(
   |s: &mut u32, d: u32| {
      *s += d;     // Update state s with element d
   },
   |s: &mut u32| *s < 20); // Stopping criterion
```

The `State::from` function initializes the state variable. In this case, the state is a single unsigned integer, but any serializable data structure can be used. The cycle starts with the `new_cycle` that takes in input the source of elements, a copy of the state, and the maximum number of iterations, where 0 indicates no threshold. The operators in the cycle (only `map` in the example) can read the current value of the state by borrowing it. At the end of each iteration, the `cycle` operator updates the state using the first closure (in the example, it adds all elements to the state) and restarts the iteration if the second closure returns true (in the example, if the state is smaller than 20). In the two closures, state is a (mutable) reference, so developers need to use a dereference $\star$ operator.

C. Windows and time

Windows identify finite portions of unbounded datasets [13]. As common in stream processing systems [14], RStream defines windows with two parameters: *size* determines how many elements they include and *slide* determines how frequently they are evaluated. RStream offers both *count* windows, where size and slide are expressed in terms of number of elements, and *time* windows, where size and slide are expressed in terms of time. Windows take in input a closure that specifies how to reduce the content of a window to a single element. For instance, the code below uses a count window to compute, every 2 elements, the sum of the last 5 elements received.

```
stream.sliding_count_reduce(
   |x, y| x + y, // Reduce closure
   5, 2);        // Size and slide
```

If a window operator is applied after a `group_by` operator, a separate window is considered for each group. For instance, the code below applies one window to the group of even numbers and one to the group of odd numbers. The example uses time windows that are evaluated every 20ms over all the elements received for that group in the last 100ms.

```
stream.group_by(|v| v % 2).sliding_p_time_reduce(
   |x, y| x + y,                 // Reduce closure
   Duration::from_millis(100),  // Size
   Duration::from_millis(20));  // Slide
```

When dealing with time windows, RStream supports two definitions of time: *processing* and *event* time. Processing time is the wall clock time of the machine computing the window. For instance, when executing the code snippet above, the process responsible for the group of even numbers computes the sum of elements received in the last 100ms according to the clock of the machine hosting that process. However, many scenarios need to decouple application time from execution time [4] to guarantee consistent results even in the case of delays or when processing historical data. To handle these cases, RStream supports event time semantics, where sources associate a logical timestamp to each element. Specifically,

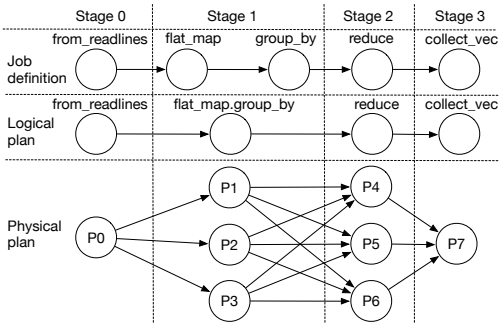


Fig. 1: Deployment of the word count job.

the `sliding_e_time_reduce` has the same syntax and behavior as the `sliding_p_time_reduce`, but uses the timestamps inside elements to measure time progress.

IV. Design and Implementation choices

RStream is implemented as a Rust framework that offers the API discussed in Section III. To run a data processing job on a set of *hosts*, developers: (i) write a Rust program that defines the job using RStream API; (ii) compile the program and copy the executable on all hosts; (iii) provide a configuration file that specifies the list of hosts and their computational resources (number of cores); (iv) start the computation using the `streamrunner` command-line tool.

The `streamrunner` tool uses `ssh` to connect to each host, spawns compute processes that connect to each other, and starts the computation. This workflow is inspired by MPI, the standard for compute-intensive tasks [15].

A. Operators deployment

Starting from a job definition, RStream creates and launches a set of processes, each executing a portion of the computation. We illustrate the translation from a job to a set of executing processes with the classic word count example. The following code snippet shows its implementation in RStream.

```
Stream::from_readlines(&file_path)
   .flat_map(|line| tokenizer.tokenize(line))
   .group_by(|(w, _c)| w.clone())
   .reduce(|(w1, c1), (_w2, c2)| (w1, c1 + c2))
   .collect_vec();
```

The `from_readlines` helper function instantiates a new source reads a file and produces a dataset of lines; `flat_map` extracts pairs of words with their associated counts; `group_by` and `reduce` group the pairs by word and reduce each group by summing all counts; `collect_vec` gathers the final results into a vector. The translation from a high-level job to a set of executing processes takes place in two steps, as exemplified in Fig. 1.

1) Logical plan: RStream first defines a *logical plan* by splitting the job into *stages*. A stage consists of a sequence of operators that do not alter data partitioning. Data partitioning changes when data is regrouped or reshuffled using the `key_by`, `group_by`, and `shuffle` operators, and when using operators that impose a single partition, such as single sources and sinks. For instance, the job in Fig. 1 consists of four stages: stage 0 contains the source, stage 1 contains the `flat_map` and `group_by` operators. Since the `group_by` operators changes the way in which data is partitioned, the subsequent `reduce` operator is associated to a new stage 2. Finally, the sink `collect_vec` defines stage 3.

2) Physical plan: Stages define the boundaries of *task parallelism*. Operators part of the same stage are chained together, each consuming exactly the same data produced by the previous one, such that it is not worth processing them in parallel: they represent an unbreakable sequence of operations. Conversely, from a stage to the next one, data can be reshuffled and potentially reordered, such that running stages in parallel may improve throughput. The benefits of task parallelism become more evident when we add *data parallelism* to the picture: elements flowing from a stage to the next one can be partitioned to run multiple instances of the same stage in parallel, one per partition.

When translating the logical plan into a *physical plan*, RStream instantiates at least one process for each stage (task parallelism). When multiple processes are instantiated for a single stage, they work in parallel on different partitions of the input stream (data parallelism). The number of processes (and consequently the number of partitions) per stage is defined in a configuration file read by the `streamrunner` tool before deploying and executing a job. For instance, in Fig. 1 a single process `P0` implements the source, three processes `P1`, `P2`, and `P3`, implement stage 1, working in parallel on different partitions of the input stream. The same happens for stage 2, run by processes `P4`, `P5`, and `P6`. Finally, a single process `P7` collects the results in stage 3.

3) Deployment and resource allocation: While developers are free to choose the number of processes running each stage based on the computing resources available, our experiments show that RStream delivers the best performance when we instantiate one process per stage per available CPU core. This approach integrates well with a key design decision we adopt in RStream: we use processes to provide task and data parallelism and we fully delegate the scheduling of such processes to the operating system, alleviating the application layer from such system-level concerns.

Since stages might have heterogeneous computational demands, reserving a CPU core to each stage might result in under-utilizing the CPU core associated to light stages. Similarly, reserving a CPU core to a single partition would be sub-optimal when the computational demands of different partitions are not evenly distributed. By running one partition of each stage per CPU core we avoid both situations, giving the operating system full freedom to schedule execution of stages and their partitions at the cost of a relatively large number of processes to be managed, a cost that proved to be minimal for the scheduler of current operating systems. Other stream processing platforms such as Flink and Kafka Streams allocate an equivalent number of parallel tasks but as threads running within JVMs, increasing the architectural layers.

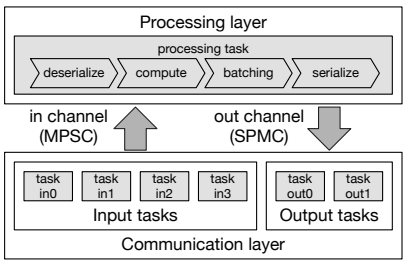


Fig. 2: Communication layer: architecture and interactions.

B. Communication

In RStream, processes communicate using direct TCP channels. Since stages are separated by reshuffling boundaries, process `P` running stage `S` receives data from every process running the upstream stage and delivers data to every process running the downstream stage. For instance, process `P1` in Fig. 1 received data from process `P0` using a single TCP channel and delivers data to processes `P4`, `P5`, and `P6` using three TCP channels.

1) Communication layer: We paid great attention to the design and implementation of the communication layer of RStream, leading to the solution illustrated in Fig. 2. We initially used MPI point-to-point primitives [15], but they employ a polling mechanism to receive data that is not suitable to overlap computation and communication. Similarly, using a separate thread for each input and output channel may lead to an excessive number of threads that compete for resources with the compute task. To avoid these problems, we implement RStream communication layer using the Tokio library [16] that provides cooperative task scheduling. We allocate a Tokio task for each input and output channel and let Tokio run all these tasks cooperatively. When a task is blocked waiting for an I/O operation to complete, it yields to the Tokio scheduler that resumes another task ready for execution.

Fig. 2 exemplifies the architecture of the communication layer for a process with four input channels, handled by tasks `in0`, `in1`, `in2` `in3`, and two output channels, handled by tasks `out0` and `out1`. The communication layer interacts with the processing layer that implements the operator logic and handles data serialization and deserialization in an additional Tokio task. We use binary serialization using `bincode` [17]: the communication layer delivers binary data to the processing layer through a multiple-producers-single-consumer channel, and receives binary data through a single-producer-multiple-consumers channel.

2) Buffering: RStream supports buffering of messages (similar to Kafka micro batches) to reduce communication overhead. Buffering is implemented in the processing layer, where outgoing messages are stored into one output queue for each outgoing channel before serialization (see Fig. 2). We support two policies: *fixed* and *adaptive*. Fixed buffering waits for a fixed number of outgoing messages for a given channel before serializing and sending them. As fixed buffering may increase latency by waiting until the desired number of messages is available, adaptive buffering flushes an output queue when a timeout elapses.

3) Timestamped streams and watermarks: When using event time, sources associate a *timestamp* metadata to each element in the stream they generate, and the runtime needs to preserve timestamp order during processing. However, in the presence of data parallelism, each partition in a stage simultaneously receives data from all the partitions in the previous stage, which does not guarantee that timestamp order is preserved. We solve this problem with a standard mechanism in stream processing platforms: *watermarks*[5]. Watermarks are special elements periodically emitted by sources that contain a single timestamp T indicating that no elements with timestamp lower than T will be produced in the future. Under event time semantics, processes buffer data before processing. When a process p in a stage s receives a watermark greater than T from all incoming channels, it can be sure that it will not receive any more data with timestamp lower than or equal to T. At that point, it processes all elements up to timestamp T from the buffer, and propagates a watermark T downstream.

V. Evaluation

Our evaluation has two goals: (1) determine if and to what extent RStream delivers better performance than today's data analytics platforms and closes the gap with custom low-level programs; (2) verify that improved performance does not come with increased program complexity.

To achieve these goals, we implement the benchmarks described in Section V-A in RStream, C++/MPI, and Apache Flink. C++/MPI sets the level of performance achievable with ad-hoc optimized implementations. Flink represents the state-of-the-art in data processing platforms, with comparable or better performance and scalability than competing solutions [18]. We compare the implementations in terms of code complexity (Section V-C), absolute performance and scalability (Section V-D). Whenever suitable, we explore different workload characteristics and configuration parameters to shed light on the key aspects that contribute to the performance of RStream.

A. Benchmarks

Our suite of benchmarks covers batch and stream processing scenarios, including iterative computations.

Word count (`wc`) counts the number of occurrences of each word in a plain text file. We use an input file of 4 GB generated from the project Gutenberg dataset of books [19].

Vehicle collisions (`coll`) performs three queries over a public dataset of vehicle collisions [20]. The dataset consists of a CSV file with about one million collision entries occurring in five years. The queries compute: (1) the number of lethal accidents per week; (2) the number of accidents and percentage of lethal accidents per contributing factor; (3) the number of accidents and average number of lethal accidents per week per borough.

K-means (`k-means`) is an iterative clustering algorithm that divides a dataset of points into `k` non-overlapping groups (clusters), aiming to maximize the distances between the centers of

Benchmark	RStream	Flink	C++/MPI	Rust
wc	35	38	160	30
coll	129	148	650	147
k-means	157	184	316	138
win-wc	39	47	216	81

TABLE I: Lines of code used to implement each benchmark.

each cluster (*centroid*). We randomly generate datasets with up to 10 million bi-dimensional points (200 MB of input data).

Windowed word count (`win-wc`) performs the word count computation over a sliding window rather than on the entire dataset. Computing aggregations over windows is a typical stream processing task.

B. Experiment setup

Unless otherwise specified, we run the experiments on an AWS cluster composed of c5.2xlarge instances, equipped with 4-cores/8-threads processors and 16 GB of RAM each, running Ubuntu server 20.04, residing in the us-east-2 zone, and communicating through the internal AWS network with an average ping time of 0.1 ms. RStream jobs are compiled with rustc 1.47.0 in release mode with all the optimizations active. We use Flink 1.11.2 executed on the OpenJDK 14.0.2, with 12 GB of RAM allocated to TaskManagers. As RStream does not currently implement fault-tolerance mechanisms, to offer a fair comparison we disabled them also in Flink: in particular, we configured Flink to save state as Java objects in memory and to never checkpoint state to durable storage. We compile C++/MPI programs with gcc 9.3.0 using OpenMPI 4.0.3 and OpenMP 4.5 and maximum optimization level. For both batch and stream processing tasks, we use a finite input dataset and we measure the overall processing time. When measuring latency, we use a single source and a single sink, deployed on the same host, and we use its local clock to compute latency. We run each experiment 6 times and compute the average value, discarding the results of the first execution to let the kernel load the dataset in the Page Cache memory.

C. Programming model

Evaluating code complexity is hard and subjective. We approach the matter by (1) counting the lines of code required to implement the benchmarks in the systems under analysis (Table I); (2) highlighting the main differences in the implementations that contribute to the count. Since the use of different programming languages affects code verbosity, we also report the lines of code of a sequential implementation of the benchmarks in Rust.

Comparing RStream and Flink, we observe a similar number of lines in all benchmarks. The slightly higher number of lines for Flink are mainly due to the Java language and the need to define custom sub-classes to encode the behavior of some Flink operators. In summary, RStream and Flink present similar programming models and very similar complexity. Interestingly, in iterative algorithms (`k-means`), we observe that RStream better hides distribution concerns in handling mutable state across iterations: while Flink requires a special construct (i.e., broadcast variables), RStream can use a standard Rust

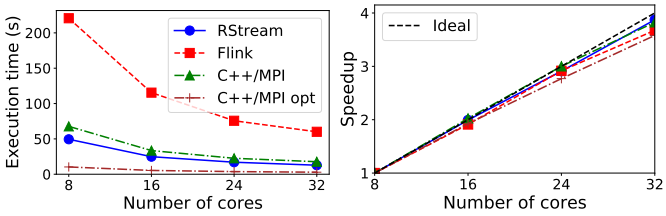


Fig. 3: `wc`: execution time and scalability.

closure, with the enclosed variable being a copy of the state at the current iteration in each process.

RStream implementations also present a level of complexity close to sequential Rust. Indeed, the use of a data abstraction that closely resembles iterators makes RStream programs appear like paradigmatic Rust code for processing collections. In `win-wc`, the sequential Rust implementation is complicated by the need of implementing by hand the windowing logic that RStream provides in its API.

In comparison, C++/MPI requires much more coding effort. This reflects in a higher number of lines of code: 2× more in `k-means`, 4× more in `wc` and 5× more in `coll` and `win-wc`. Most significantly, the additional lines of code reflect the necessity to handle low-level concerns that RStream and Flink abstract away, greatly increasing implementation complexity. First, developers need to select the data structures that encode input data and intermediate results, and define the serialization and deserialization format and strategy: for instance, if and how to overlap communication, serialization, and processing. While this offers a high degree of flexibility, it requires fine tuning and might lead to sub-optimal choices, as also discussed in Section V-D. Second, C++ memory model requires considering when data structures are deallocated and might lead to runtime errors such as memory leaks or invalid references. The ownership model of Rust automates memory deallocation and enables the compiler to statically check safety, while the programming model of RStream hides management of all the data structures used for networking and I/O. Third, MPI exposes low-level primitives for communication and synchronization. For instance, in the case of streaming computations, as in `win-wc`, developers need to manually encode buffering strategies for inter-process communication. Again, the increased freedom enables for fine tuning, but can also expose to wrong communication patterns such as deadlocks. Finally, our C++/MPI implementations combine process-level parallelism with thread-level parallelism (using OpenMP). This yields some additional performance improvements but comes at the cost of additional code complexity.

D. Performance

We now focus on absolute performance and study the scalability of each implementation by moving from one host (8 virtual cores) to 4 hosts (32 virtual cores).

1) Word count (`wc`): Fig. 3 shows the execution time and scalability for `wc`. In absolute terms, RStream completes the task about 4.7× faster than Flink with 4 hosts, despite a similar

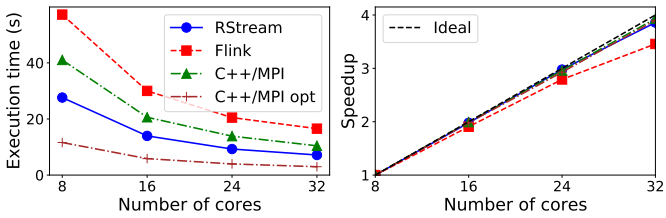


Fig. 4: `coll`: execution time and scalability.

programming model, which shows the efficiency of its design and implementation choices. We optimized the C++/MPI code in many ways. In the reduction phase, RStream and Flink partition the dataset by word and perform the reduction in parallel, before collecting all the results in a single process and saving the results. Given the limited size of the partial results, in the C++/MPI implementation, we skip the intermediate phase and collect all partial results directly in a single process, saving one communication step. We further limit communication by using only one process per host and exploiting thread-level parallelism inside each process, which brings an improvement of about 8% in processing time with 4 hosts. Despite these custom-made optimizations, C++/MPI is still about 35% slower than RStream. A detailed analysis showed a bottleneck when reading data from file using functions from the standard C++ library and parsing using regular expressions (the same we do in Flink and RStream). Thus, we implemented an additional version with ad-hoc file reading (by mapping the file in memory with `mmap`) and a simplified parser that only considers 7bit, ASCII files instead of UTF-8 encoded text. This version is labeled "C++/MPI opt" in Fig. 3 and is about 4.4× faster than RStream with 4 hosts, but at the cost of additional code complexity and reduced generality and reusability. Some of the improvements in IO and parsing can also be implemented in RStream, with an improvement of more than 30% in execution time. However, we do not consider them in the executions shown in Fig. 3 as they lead to non-paradigmatic Rust code.

All three implementations achieve near linear scalability: when moving from 1 to 4 hosts we measure a speedup of 3.87× for RStream, 3.83× for C++/MPI, 3.67× for Flink, and 3.58× for the optimized version of C++/MPI. Indeed, the most expensive operations, namely reading and parsing the file, and performing a partial count, are executed in parallel without synchronization across processes.

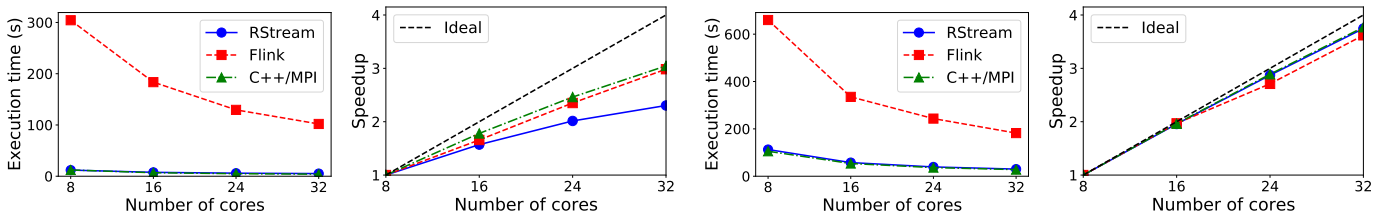
2) Vehicles collisions (`coll`): Fig. 4 shows the execution time and scalability for the `coll` benchmark. We consider the overall execution time to answer the three queries, which involves starting three jobs in Flink, and reading the input data three times in RStream and in C++/MPI. Also in this case, RStream outperforms Flink with more than 2.3× speedup with 32 cores and is about 30% faster than C++/MPI when using standard IO and parsing. A C++/MPI version using custom IO and parsing is up to 2.4× faster than RStream. RStream and C++/MPI implementations show close-to-ideal scalability, while Flink has a slightly lower speedup of 3.45× when moving from 1 to 4 hosts.

3) K-means (`k-means`): Fig. 5 shows the execution time and scalability for `k-means`, using an input dataset of 200 MB (10 million bi-dimensional points) and a fixed number of 30 iterations. Fig. 5a shows the results for 50 centroids and Fig. 5b shows the results for 750 centroids. A higher number of centroids requires more computation at each iteration. RStream achieves a level of performance that is comparable to a custom C++/MPI implementation. Indeed, most of the execution time is spent inside the iterations to compute the distances between points and centroids, which reduces the benefits of optimized input reading and (de)serialization.

With 50 centroids (Fig. 5a) the overhead of communication at the end of each iteration limits the speedup of RStream to 2.3× when moving from 1 to 4 hosts. However, the cost of iterations are even more evident in Flink, and RStream achieves 20–25× better performance. With 750 centroids (Fig. 5b) each iteration becomes computationally more expensive, reducing the relative contribution of communication, and leading to 3.75× speedup when moving from 1 to 4 hosts. In this scenario RStream remains up to 6× faster than Flink.

To better understand how the computational complexity of each iteration affects performance, we repeat the same experiment (10 million points, 30 iterations, 4 hosts) while changing the number of centroids. Fig. 6 shows the result we measured: the execution time from 1 to 20 centroids grows faster in Flink than in RStream and C++/MPI. After this threshold, the three systems exhibit a linear behavior, and the relative advantage of RStream slowly drops from about 25× to about 6×. C++/MPI is more efficient than RStream with very few centroids (up to twice as fast with less than 10 centroids), when the execution time is below 10 s. As the time spent in the computation increases, the relative weight of custom IO optimizations in C++/MPI decreases, and the execution time of the two systems becomes comparable.

4) Windowed word count (`win-wc`): As a representative of stream processing computations, we use `win-wc` over a count window of size 10 and slide 5. The limited size of the window forces continuous communication between the first stage that counts words and the subsequent stage that aggregates partial counts. Fig. 7 shows the results we measured: due to frequent communication, RStream and C++/MPI present a higher execution time and lower scalability than in `wc`. We suspect that the computational efficiency of these two implementations makes inter-process communication and synchronization become a bottleneck when increasing the number of processes. This is particularly visible in the C++/MPI implementation, that does not scale beyond 3 hosts. We experimented several C++/MPI variants, manually implementing message batching strategies to improve execution time. We achieved the best performance (reported in Fig. 7) with half of the processes reading from files and half of the processes performing windowed reduction. Flink performance drops less, indicating that reading from input and processing still represent a dominant part of execution time. In comparison, RStream is still about twice as fast as Flink.



(a) 50 centroids (b) 750 centroids

Fig. 5: `k-means`: execution time and scalability (10M points, 30 iter).

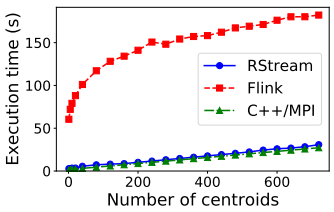


Fig. 6: `k-means`: num of centroids (10M points, 30 iter).

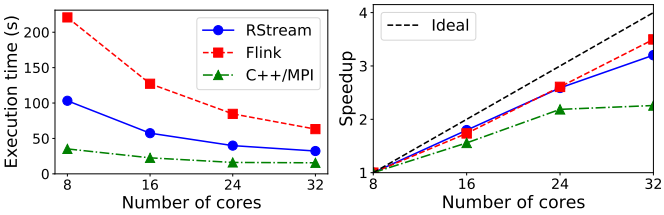
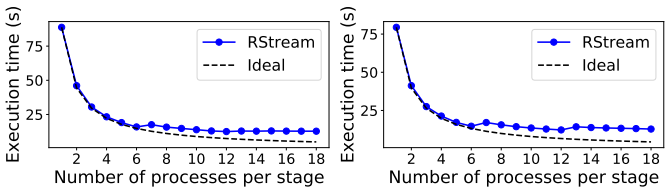


Fig. 7: `win-wc`: execution time and scalability.

5) Scalability on a single host: As discussed in Section IV, we aim to maximize the use of resources by instantiating one process for each stage for each available CPU core. To motivate this choice, we study how RStream scales on a single host when increasing the number of processes. We perform this experiment on a single machine equipped with a 6-core/12-threads Intel Core i7-8700 CPU and 64 GB of RAM. Since the goal of the experiment is to study scalability with the number of processes, we disable Turbo Boost and lock the CPU frequency at 3.2 GHz.

Fig. 8 shows the results we obtain with the `wc` and `k-means` algorithms. For `wc`, we use a dataset of 1.2 GB. For `k-means`, we use a dataset of 5 million points, 70 centroids, and 30 iterations. Both benchmarks include 2 processing stages: with 6 processes per stage we instantiate 12 processes, the same as the number of CPU threads. Up to this point, both benchmarks show nearly perfect scalability. After this threshold, we observe an initial performance drop when starting to overcommit physical resources and then the curve flattens. In both cases, me measure the lowest execution time with 12 processes per stage, that is, one process for each stage for each (logical) CPU core. This shows that delegating resource management to the operating system, which can prioritize stages and partitions based on their computational demands, is indeed effective.



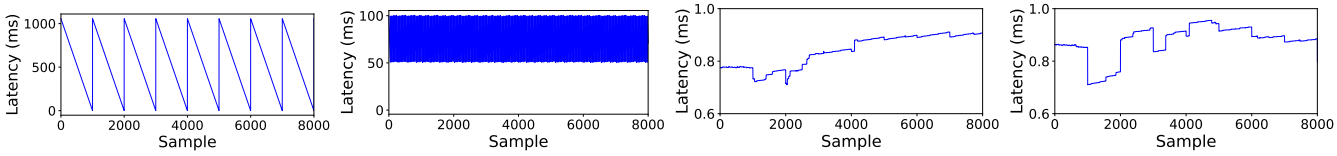
(a) `wc` (b) `k-means`

Fig. 8: Scalability with the number of processes.

6) Buffering and latency: Low latency is often a requirement in stream processing, but solutions that optimize latency negatively affect throughput. We test the effect of buffering strategies on execution time and latency by running a benchmark on 3 hosts (24 cores). The benchmark consists of a single source that generates integer numbers and a parallel `map` that simply copies input to output, toward a single sink. By avoiding computationally-expensive operations, we better measure the contributions of inter-process communication.

We study how buffering strategies affect throughput, measured in terms of overall execution time for a fixed input of 4 GB. When using a fixed buffer size of 1k elements, performance increases by more than 60× with respect to immediately sending individual data elements. Our experience with RStream showed us that execution time rapidly improves when increasing the buffer size, but after a threshold the improvements become negligible. Although the precise threshold depends on the specific workload and network infrastructure, a buffer size of 1k is sufficient in all the scenarios we tested. In fact, we used a buffer size of 1k in all experiments previously discussed. An adaptive buffering that delivers elements to the next stage every 50 ms even if fewer than 1k elements are available increases execution time by less than 15%. We used this approach for the `win-wc` benchmark, as it guaranteed an upper bound for latency with limited overhead.

Fig. 9 shows the effect of the buffering strategies on latency. We consider both a light load scenario, where the source produces one element every millisecond, and a heavy load scenario, where a source feeds input data at maximum rate. Under a light load, the fixed buffering strategy accumulates 1k elements before sending them to the next stage, so the first element in a buffer waits for about 1 second before being delivered, while the last element in the buffer is delivered immediately. This result in the behavior of Fig. 9a. When



(a) Light load - fixed 1k (b) Light load - adaptive 1k, 50 ms (c) Heavy load - fixed 1k (d) Heavy load - adaptive 1k, 50 ms

Fig. 9: Buffering strategies: effect on end-to-end latency.

applying adaptive buffering (Fig. 9b), we observe a maximum latency close to 100 ms, as the job contains two communication steps from source to sink. Under a heavy load, the input rate dominates and we observe similar behaviors for the fixed strategy (Fig. 9c) and for the adaptive strategy (Fig. 9d).

These experiments confirm that an adaptive strategy can limit latency in scenarios where some buffers might take time to fill, with no impact under heavy load.

VI. Related Work

Our work focuses on programming models and platforms for distributed data processing. In this context, the dataflow model we consider in this paper has attracted increasing attention over the last several years, and many platforms have been implemented by researchers and practitioners [21], [22], [4], [23]. The Flink system we use for our evaluation is a mature commercial product, representative of these platforms, and often cited for its good level of performance [18].

To simplify the implementation of complex algorithms, most platforms also offer higher-level libraries for specific domains. Prominent examples are the libraries to process structured data [24], which convert declarative queries from SQL-like languages to dataflow programs, often providing unified abstractions for batch and stream processing of structured data [25], [26]. The conversion enables automated query optimizations, which are common in database systems. Other examples of libraries range from machine learning [27] to graph processing [28], to pattern recognition in streams of events [29]. As all these libraries generate dataflow programs, we could implement similar abstractions on top of RStream. Given the performance advantages of RStream especially in iterative computations, we believe that it has the potential to bring significant improvements in domains like machine learning and graph processing.

Some research works propose alternative programming models or extensions to the dataflow model. Naiad [30] and its timely dataflow model enrich dataflow computations with explicit timestamps that enable implementing efficient coordination mechanisms. These characteristics make it more expressive than the dataflow model used in distributed processing frameworks and adopted in our work, but they come at the cost of a lower-level programming paradigm that exposes communication and synchronization concerns. Interestingly, the timely dataflow model has been also implemented in Rust [31]: as future work, we plan to study this implementation to gather deeper insight on how alternative programming models and implementation choices can influence the balance between performance and ease of use.

Fernandez et al. [32] introduce an imperative programming model with explicit mutable state and annotations for data partitioning and replication. TSpoon [33] extends the dataflow abstraction with additional features and guarantees, such as transactional semantics. These efforts are orthogonal to our work, which mostly targets efficient system design and implementation, rather than investigating new models.

Some systems optimize the use of resources on a single machine. For instance, StreamBox targets multi-core machines [34], while SABER considers heterogeneous hardware platforms consisting of multi-core CPUs and GPUs, which are increasingly available in modern heterogeneous servers [35]. By building on a compiled language, RStream simplifies the access to hardware resources with respect to JVM-based systems. In fact, we already experimented with OpenCL-based implementations of operators that exploit GPUs, and we plan to further explore this line of research in future work.

As a final note, all modern data processing systems provide fault-tolerance mechanisms to recover from software and hardware failures. As the current version of RStream does not offer fault-tolerance mechanisms, we disabled them in all the systems used in our evaluation (see Section V) for a fair comparison. We plan to implement fault-tolerance in future releases by building on consolidated approaches such as asynchronous snapshots [36], which bring negligible runtime overhead, since they do not block normal processing.

VII. Conclusions

In this paper, we show that *it is possible to reduce the performance gap between distributed data processing platforms and custom, low-level implementations, without relinquishing abstraction*. To do so, we introduce RStream, a novel data processing framework written in Rust. RStream provides all core features of state-of-the-art data processing platforms – unified batch and stream processing, iterative computations, windowing, time-based data analytics – within the same, high-level processing model. At the same time, its design and implementation choices – compiled language, efficient memory and communication management, task allocation that maximizes the use of processing resources – yields up to more than an order of magnitude improvements in throughput with respect to existing data processing systems, rivaling custom MPI solutions in some workloads.

We believe that these findings will foster investigations to build a new breed of data processing platforms, offering a bet-

ter trade off between simplicity and performance than possible today. We will continue to contribute to this research line by investigating new features in the programming model, such as joins of streams and nested iterations, and new optimizations to the processing engine, such as multiplexing of network communication and exploitation of hardware accelerators.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013, pp. 423–438.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "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.*, vol. 8, no. 12, pp. 1792–1803, 2015.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, 2015.

[6] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, vol. 53, pp. 121–130, 2015, iNNS Conference on Big Data.

[7] P. González, X. C. Pardo, D. R. Penas, D. Teijeiro, J. R. Banga, and R. Doallo, "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*, ser. CCGrid '17. IEEE Press, 2017, pp. 797–806.

[8] S. Klabnik and C. Nichols, "The rust programming language," 2018.

[9] T. White, *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[10] A. Turon, "Abstraction without overhead: traits in rust," 2015. [Online]. Available: https://blog.rust-lang.org/2015/05/11/traits.html

[11] ——, "Fearless concurrency with rust," 2015. [Online]. Available: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[12] "Serde framework." [Online]. Available: https://serde.rs

[13] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, p. 121–142, 2006.

[14] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "Secret: A model for analysis of the execution semantics of stream processing systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, p. 232–243, 2010.

[15] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced Features of the Message Passing Interface*, 2nd ed. MIT Press, 1999.

[16] "Tokio asynchronous runtime." [Online]. Available: https://tokio.rs

[17] "Bincode crate." [Online]. Available: https://docs.rs/bincode/

[18] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Perez-Hernandez, "Spark vs flink: Understanding performance in big data analytics frameworks," in *International Conference on Cluster Computing*, ser. CLUSTER '16. IEEE, 2016, pp. 433–442.

[19] "Project gutenberg." [Online]. Available: https://www.gutenberg.org

[20] "Nyc open data – motor vehicle collisions dataset." [Online]. Available: https://data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Crashes/h9gi-nx95

[21] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endow.*, vol. 6, no. 11, p. 1033–1044, 2013.

[22] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '14. ACM, 2014, pp. 147–156.

[23] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, p. 239–250.

[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, p. 1383–1394.

[25] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, ser. BIRTE '18. ACM, 2018.

[26] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. Knowles, "One sql to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '19. ACM, 2019, p. 1757–1772.

[27] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 1, p. 1235–1241, 2016.

[28] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *USENIX Symposium on Operating Systems Design and Implementation*, ser. SOSP '14. USENIX Association, 2014, pp. 599–613.

[29] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis, "Complex event recognition in the big data era: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 313–352, 2020.

[30] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013, p. 439–455.

[31] "Timely dataflow." [Online]. Available: https://github.com/TimelyDataflow/timely-dataflow

[32] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC'14. USENIX Association, 2014, p. 49–60.

[33] L. Affetti, A. Margara, and G. Cugola, "Tspoon: Transactions on a stream processor," *Journal of Parallel and Distributed Computing*, vol. 140, pp. 65–79, 2020.

[34] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox: Modern stream processing on a multicore machine," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '17. USENIX Association, 2017, p. 617–629.

[35] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '16. ACM, 2016, p. 555–569.

[36] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015.