FlameStream: Model and Runtime for Distributed Analytical Stream Processing

Igor E. Kuralenok
JetBrains Research
St. Petersburg, Russia
ikuralenok@gmail.com

Artem Trofimov
JetBrains Research
St. Petersburg, Russia
trofimov9artem@gmail.com

Nikita Marshalkin
JetBrains Research
St. Petersburg, Russia
marnikitta@gmail.com

Boris Novikov
JetBrains Research
St. Petersburg, Russia
borisnov@acm.org

ABSTRACT

In recent years, there has been a growth in research and industrial solutions in the field of distributed stream processing. However, even state-of-the-art stream processing systems are inconvenient to work with in some ways. Firstly, existing solutions suppose that the state of operations should be managed directly by a user, and that can be confusingly for inexperienced users of analytical processing systems. Secondly, these systems do not provide for deterministic processing by default. Furthermore, the most common approach for this feature is buffering. Its main problem is the extra cost for blocking before each order-sensitive operation. The goal of this paper is to propose a model, which addresses both recognized issues. On the one hand, our model stateless from the business logic perspective. Such behavior is obtained by the technique that allows a state to be a part of a stream. On the other hand, it is deterministic by design. We introduce an optimistic method that avoids buffering before each operation to achieve determinism with low overhead. The experiments show that prototype of our system requires a low extra cost for supporting proposed features and is able to outperform alternative industrial solutions in certain conditions.

1 INTRODUCTION

Nowadays, a lot of real-life applications use stream processing for network monitoring, financial analytics, training machine learning models, etc. State-of-the-art industrial stream processing systems, such as Flink [7], Samza [14], Storm [3], are able to provide low-latency and high-throughput in distributed environment for this kind of problems. However, users of these systems still experience a variety of difficulties during the process of their adaptation to custom tasks. Firstly, implementation of stateful operations requires explicit state handling [10, 17]. Such paradigm is flexible but can complicate business logic that in turn can lead to sophisticated bugs.

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

Secondly, the current generation of stream processing systems requires high overhead for the property of determinism [20]. This property implies that processing results are repeatable up to input order. Most of the modern systems assume that events are fed to the system with monotonically increasing timestamps or with minor inversions [19]. Often, such timestamps can be assigned at system's entry. Nevertheless, even if input items arrive monotonically, they can be reordered because of the subsequent parallel and asynchronous processing that leads to losing the determinism property. The lack of determinism causes complex processes of workflow verification and limits the applicability of such systems [16, 20].

In this work, we introduce FlameStream - stream processing model that addresses both mentioned issues. We define *drifting state* technique to make computations stateless from the business-logic point of view. This approach is based on the idea that state can go along the stream as an ordinary element.

The deterministic processing is achieved in our model using strong ordering. The typical way to perform in-order processing is to set up a special buffer in front of operation [12]. This buffer collects all input items until some user-provided condition is satisfied. Then the contents of the buffer are sorted and fed to the operation. The main disadvantage of such technique is latency growth [20]. This issue becomes even more significant if the processing pipeline contains several operations that require ordered input. We introduce an optimistic approach to handle out-of-order items to avoid this issue. Our evaluation demonstrates that our method has low overhead and can outperform alternative industrial solution under normal load conditions.

Therefore, the contributions of this paper are the following:

- Definition of stateful computational model that does not require state handling from user
- Introduce the optimistic schema for deterministic processing and demonstrate its performance competitiveness

The rest of the paper is structured as follows: in section 2 we introduce the basics of the proposed model, state management is detailed in section 3, optimistic schema for deterministic processing is described in section 4, the implementation details of the prototype are discussed in 5 and its performance is demonstrated in 6, the main differences of our system from the existing are shown in 7, finally we discuss the results and our plans in 8.

2 COMPUTATIONAL MODEL

In this section we outline the model of our system. We focus on the core concepts and definitions. Such model is in line with common stream processing systems but has some differences: the strict ordering model and reduced operation set.

2.1 Data flow

The basic data flow abstraction is a *stream*. The stream is represented by an infinite heterogeneous sequence of data items. Data item is a *payload* and a *meta-information* associated with it.

$$DataItem := (Payload, Meta)$$

The payload is an arbitrary user-provided data. Meta is structured system-assigned information. The primary purpose of the meta-information is to impose the total order on data items.

Data payloads are got into the stream through *front* and got out through *barrier*. Particularly, front creates data items from input events by assigning them meta-information. Inside stream, data items can be dropped or their payloads and metas can be transformed. Barrier removes meta-information and outputs back pure payloads.

2.2 Computational flow

The stream between front and barrier is handled by a directed data flow graph. Each node of the graph represents a single operation, which can have multiple inputs and outputs. Edges indicate the order of these operations. Data items are processed one-by-one in a "streaming" manner. Figure 1 shows the example of data flow graph.

Our model allows cycles in the graph while data flow graphs are commonly assumed to be acyclic (DAGs) [6, 21]. Moreover, as we show further, there are cases when cycles are required, e.g., for MapReduce-based algorithms.

2.3 Physical deployment and partitioning

Data flow graph is distributed among computational units. Each unit runs a process called *worker*, and each of the workers executes complete data flow graph. An integer interval (hash range) is assigned to every worker. Intervals are not intersected and cover the range of 32-bit signed integer.

Each operation input has a user-provided hash function called *balancing function*. This function is applied to the payload of data items and determines partitioning before each operation. After that, the data items are sent to the worker, which is responsible for the associated hash range. Therefore, load balancing explicitly depends on the user-defined balancing functions. This allows the developer to determine optimal balancing which requires the knowledge of the payload distribution. The system optimizes the hash ranges assignment according to the processing statistics.

2.4 Ordering model

We assume that there is a total order on data items. Ordering is preserved when an item is going through the operations. More precisely, the order of output items is the same as the order of corresponding input items. If more than one item is generated, they

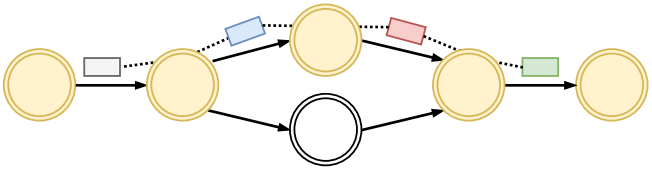

Figure 1: An example of the data flow graph

are inserted in output stream sequentially. Moreover, the output follows corresponding input but precedes the next item. Without diving into details, it should be noted that the order of items is maintained across different fronts.

The concept of ordering is shown in Figure 2. Data item with payload $1'$ is the derivative of the item with payload 1, according to operation F. The same is for items with payloads $2'$ and 2. After union operation, the order between 1 and 2 is preserved. Furthermore, $1'$ follows 1, and $2'$ follows 2.

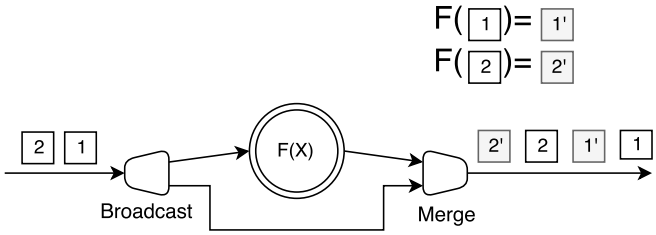

Figure 2: The concept of ordering model

We assume that input items of the operations are strictly ordered.

2.5 Operations

The list of available operations includes:

Map applies a user-defined function to the payload of an input item. This function returns a sequence of data items with transformed payloads. An output sequence can be empty.

Broadcast replicates an input item to the specified number of operations.

Merge operation is initialized with the specified number of input nodes. It sends all incoming data to the output.

Grouping has a *window size* parameter. Grouping stores input items into distinct buckets by the value of the input balancing function applied to the payload. When the next item arrives at the grouping, it is appended to the corresponding bucket. Each time the grouping outputs window-sized *tuple item*, which consists of the most recent (in terms of the meta-information) items of this bucket. If the size of the bucket is less than the window, all items of the bucket are taken. Grouping is the only operation that has a state.

The following example illustrates the semantics of the operation. The grouping accepts items with payload represented as natural numbers: 1,2,3, etc. The hash function returns 1 if the number is even and 0 otherwise. If the window is set to 3, the output is:

(1), (2), (1|3), (2|4), (1|3|5), (2|4|6), (3|5|7), (4|6|8)...

There are two important properties of the grouping operation: the output tuple is identified by its last element, the results among items with different values of a hash function are independent.

2.6 User-defined parameters

A user can set up the following parameters:

(1) Computational flow
(2) Balancing functions of the inputs
(3) Map functions
(4) Grouping windows

These parameters can produce more than one graph, which can yield equivalent results. Choosing among them is a performance optimization problem that relies on the system. It is important to mention that there are no parameters for state-management. Therefore, business-logic is stateless. Nevertheless, the operations set is enough to implement any MapReduce transformation as shown in the next section.

3 DRIFTING STATE

Most of the computational pipelines require state carrying between consecutive data items. For example, moving average calculation needs partial sums to be stored. The state management in distributed systems is a complicated topic that requires delicate treatment.

Some systems forces that all state management is put in business-logic: making state persistent, replication, ensuring state consistency, etc. Others shifts some difficulties to the system's concern, e.g., Flink provides a state API that guarantees consistency in case of failures [6]. While it eases the heavy destiny of users, the core problem remains.

We propose a technique to simplify the development of stateful pipelines by eliminating the operation's state while remaining able to implement common transformations. We call this approach *drifting state*. The core idea is to make a state a part of the heterogeneous data flow. In order to do state updates, the drifting state is grouped with new elements, combined, and the new state is emitted. The limited number of such combines can be implemented with DAG by repetition of grouping and map operations. As stream processing is aimed to infinite sequences, there is a need for a cycle in the computational pipeline. The exact structure of such cycle is described in the following subsection with an example of typical stateful transformation, MapReduce.

The complex graph with multiple stateful operators can be constructed using a cycle for each stateful component. It is important to say that such tangled structures may not be exposed to the end-user and can be hidden by some high-level API.

3.1 MapReduce transformation

Map stage of MapReduce can be formulated directly in terms of our map operation. The algorithm 1 shows a generic reduce stage. The *accumulator* is an explicit state that should be kept between subsequent iterations.

To implement reduce stage we apply the drifting state technique and make the accumulator value a part of the stream. Figure 3 shows a generic graph for MapReduce transformation. Map and reduce stages are highlighted with a dashed line.

Algorithm 1 Generic reduce stage

```
function REDUCE(key, values)
    accumulator                                  ▷ reduce's state
    for all v ∈ values do
        COMBINE(v, accumulator)
    end for
    return MAP(accumulator)
end function
```

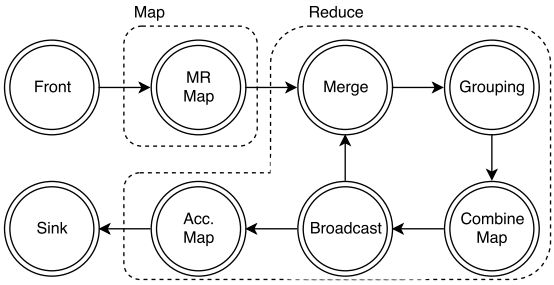

Figure 3: Logical graph for MapReduce transformations

There are four types of data items in this stream: *input*, *mapped*, *accumulator*, *and reduced* items. Mapped, accumulator and reduced items have the key-value structure of a payload. The operations of the stream have the following purposes:

- The first map operation accepts input items, and outputs mapped items according to map stage of MapReduce model
- The window of grouping is set to 2. It is used to group current accumulator with next data item to combine them further. The hash function is designed to return distinct values for payloads with distinct keys
- The second map implements the actual combining. It accepts inputs that have a form of: *(mapped item)* or *(accumulator item, mapped item)*. The first kind is transformed into some initial value. The second one is combined into the new accumulator item as specified by reduce stage of MapReduce. The tuples with structure *(mapped item, accumulator item)* are filtered out
- The third map is the accumulator map. It accepts accumulator items and applies the final map transformation to them

The fundamental idea is that ordering assumptions about data items guarantees that each accumulator item always arrives at the grouping right after previous mapped item and before a new one. Hence, each mapped item that has not been combined yet would be grouped with the right accumulator item. Additionally, when combine map accepts tuple *(mapped item, accumulator item)*, then it means that mapped item was generated before accumulator item, and therefore, it had been already combined. The cycle gives the ability for new accumulator items to get back in the grouping operation. The accumulator map transforms the accumulator item into the final reduced item right before sink. Thereby, the stream reacts to each input item by generating new reduced item, which contains the actual value of the reduce stage.

3.2 Example: word count

We illustrate the MapReduce algorithm with an example of word counting. Map stage of this algorithm transforms each input word into key-value pair where the word is a key, and the value is 1. Reduce stage sums all values into the final result for the specific key. In this case, the accumulator map is omitted, because the accumulator is the actual result of the reduce stage.

The example of input/output items, which are generated/ transformed by the part of the logical graph, is shown in Figure 4. According to our graph for MapReduce transformations, the item *m[dog, 1]* represents mapped item with key "dog" and value 1. The item *a[dog, 1]* describes accumulator item with key "dog" and value 1. Figure shows how the model reacts on two consequent input items containing word "dog". The meta-information of items is omitted for simplification. More precisely, there are four stages separated by dotted lines:

(1) New mapped item with key "dog" arrives at grouping with an empty state. Grouping outputs tuple with this single item. Combine map transforms it into the first accumulator item for key "dog" and value 1.
(2) The accumulator item arrives at grouping after it went through the cycle. It is grouped in the tuple with the mapped item that has been already in the state with key "dog". However, combine map drops this tuple, because of the order of items.
(3) New mapped item with key "dog" arrives at grouping. It is inserted right after the accumulator in the bucket for key "dog". The grouping operation outputs tuple containing the accumulator item and new mapped item. Map operation combines reduced and mapped items into new reduced items with key "dog" and value 2.
(4) New accumulator item arrives at grouping through the cycle, but new generated tuple is not accepted by combine map, as well as in step 2.

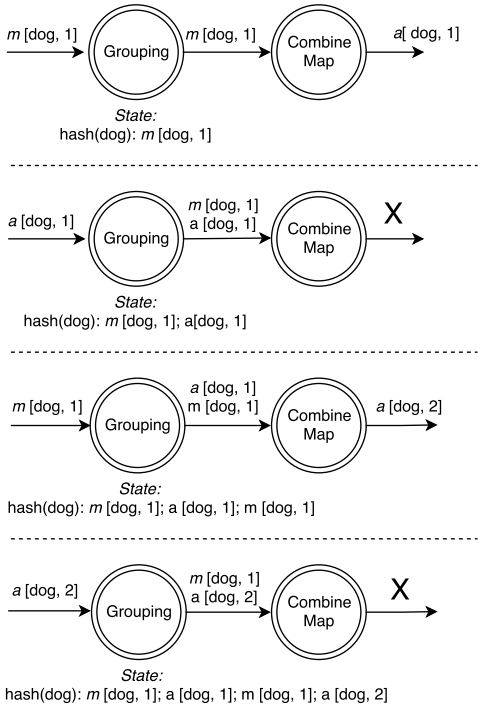

Figure 4: The example of input/output items, which are generated/ transformed by the part of the logical graph for word counting

4 DETERMINISTIC COMPUTATIONS

Deterministic execution is a desired property of any distributed system. If there are multiple possible outcomes, the developer needs to reason about which results are valid, which are equivalent and which are considered to be invalid.

To reduce outputs to only one possible result, we impose two restrictions on our model:

- We require map function to be pure: return value is only determined by its input values, without observable side effects
- We impose a strict ordering requirement on the grouping's input

While the first requirement can be easily satisfied by moderating the provided business-logic, the second one is foreign to the distributed systems. Because of asynchrony and the possible existence of multiple paths between two nodes, it is hard to deliver the right order.

In this section we review common approaches for the order enforcement, then we introduce our approach.

4.1 Existing solutions

There are two most common methods that are used to implement order-sensitive operators: in-order processing (IOP) [4, 9] and out-of-order processing (OOP) [12].

According to IOP approach, each operation must enforce the total order on output elements that can be violated due to asynchronous nature of execution. Buffering is usually used to fulfill this requirement. For example, the implementation of the merge operator, in the presence of skew between input streams, must buffer the earlier one to enforce order on the output.

OOP is an approach that does not require order maintenance if it is not needed. In the case of ordering requirements, OOP buffers input items until a special condition is satisfied. This condition is supported by progress indicators such as punctuations [18], low watermarks [1], or heartbeats [15]. They go through the stream as ordinal items, but do not trigger business-logic of the operations. Each progress indicator carries meta-information and promises that there are no elements with less meta-information. Therefore, indicators must be monotonic, but data items between two consecutive indicators can be arbitrarily reordered. Data sources periodically yield them.

While these methods are commonly adopted in practice, we find them to have unpredictable latencies. In our system, we employ an optimistic approach for handling out-of-order items.

4.2 Optimistic approach

As it was defined previously, only the grouping operation maintains a state and the state depends on the order of incoming items. Therefore, there is a need to enforce the right order to achieve deterministic processing.

As it was mentioned above, conservative methods for order enforcing can imply high latency overhead. Regarding our optimistic approach, we accept the fact that grouping can produce incorrect tuples. However, we guarantee that all correct tuples are eventually produced. The correctness of tuple means that this tuple would be generated if the order assumption was satisfied.

To eventually produce all correct tuples, we use an approach called *replay*. If an item arrives the grouping operation, according to the meta-information order, nothing is replayed, and only the most recent window is produced. However, if an item is out-of-order, it is inserted in the bucket at the correct location, and all tuples, which contain this element, are reproduced. Thereby, replay guarantees that eventually all correct tuples are generated. At the same time, for tuples, that has been produced but became invalid, *tombstones* are sent.

Tombstones are ordinal data items but with a special flag in its meta-information. This flag means that tuples with such meta-information are invalid, and they should not leave the system. Tombstones have the same payloads as invalid items in order to go along the same path in the computational pipeline.

The example of replay is shown in Figure 5, The green item is out-of-order. The output consists of the new valid items *(1, 2) and (2, 3)*, and the tombstone *(1, 3)* for the previously generated item.

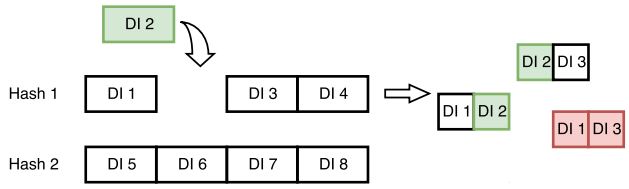

Figure 5: The replay in grouping with window = 2. The new items are generated on insertion

In the case of the right order of input items, there are no redundant items produced.

The barrier filters out invalid elements, when corresponding tombstones arrive. It is partially flushed for some meta-information interval when there is a guarantee that there are no any out-of-order items and tombstones further up the stream for this range. The exact technique for providing such guarantee is defined in Section 5.

4.3 Advantages and limitations

The proposed architecture's performance depends on how often reorderings are observed during the runtime. In the case when the order naturally preserved there is almost no overhead: when the watermark arrives, all computations are already done. The probability of reordering could be managed on a business-logic level and optimized by the developer. In experiments section it is shown that the computational nodes count is one of such parameters. Regarding the weaknesses, this method can generate additional items, which lead to extra network traffic and computations. Experiments, which are shown in the section 6 demonstrate that the number of extra items is low.

5 IMPLEMENTATION

5.1 Ordering model

The meta-information of data item is a tuple of a *global time*, a *trace*, *child ids* and a tombstone flag.

$$Meta := (GlobalTime, ChildIds[], Trace, IsTombstone)$$

Global time is assigned to data item once the item enters the system. It is a pair of milliseconds since the epoch start and the identifier of the front. The identifier is used to assign different global times to different items, even in case of wall-clock collisions.

$$GlobalTime := (FrontTs, FrontId)$$

Global times are compared by front timestamp if they coincide - by front id. It is important to notice that we do not rely on any clock synchronization between nodes, but we require strict monotonicity within the single front. The only implication of the clock skew is the system degradation regarding latency: 1ms of the nodes clock difference appends 1 ms to minimal latency.

Each map operation can produce multiple items from one. To differentiate them the ordinal number, *child id*, is stored in the meta information. The *ChildIds* is an array of child ids, that corresponds to the all visited map operations.

The global time and child ids are enough to uniquely identify data item within stream if all processing is done without replays. If there are any replays happened during processing, multiple items with the same global time and child ids exist in the stream. Multiple tombstones with the same global time and child ids can exist as well. They can take different paths in computational flow and travel within them with different speeds.

Despite this fact, an invalid element and the corresponding tombstone go along the same path, because they have the same payload and the balancing functions are deterministic. Moreover, the tombstone visits operations strictly after corresponding item, as links between operation are expected to be FIFO.

To match tombstone with proper item, there is *Trace* value stored in the meta-information. Trace encodes the path that item have traversed. The unique random 64-bit identifier is assigned to each physical operation. The trace is a xor of all operations' ids visited by item so far.

$$Trace := \bigoplus_{op \in \text{visited}} Id(op)$$

Metas are compared lexicographically: firstly global times are compared, then child ids, then traces. Notably, this order is in line with the FlameStream's ordering model.

Figure 6 shows the topology of each operation and how it affects the meta. Grouping and merge operations update trace of the data item by xoring initial trace with its physical id. Map and broadcast

apply the same update, but also append child id for each output item.

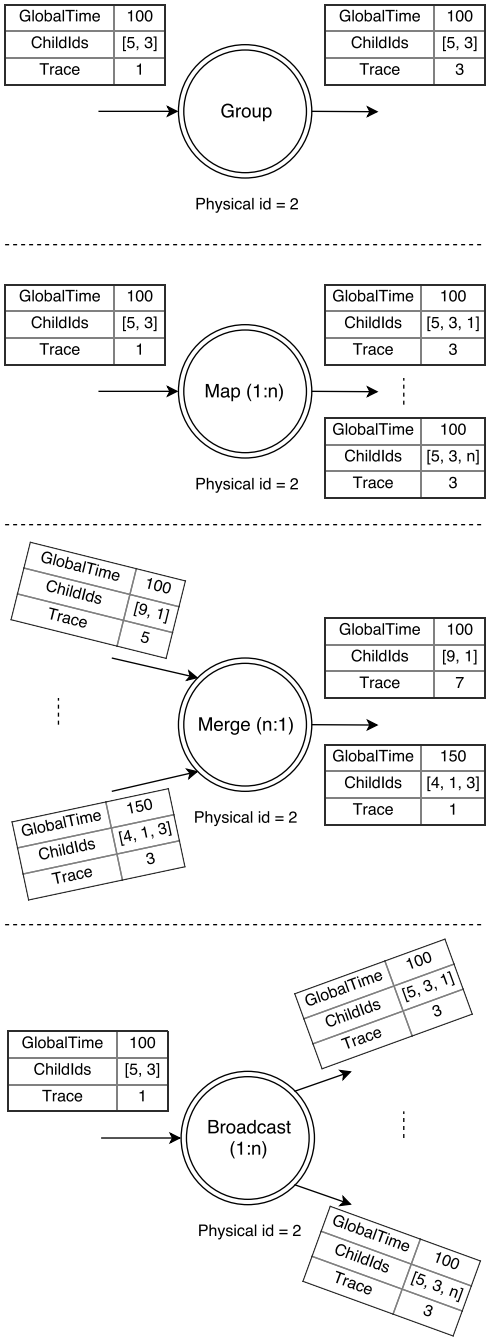

Figure 6: The topology of each operation and how it affects the meta

The structure of meta provides for tracking different relations between data items:

- Items with same global time and common prefixes are produced from the same item
- Item with lower meta is generated earlier, at front or in the stream
- If there are two items with the same global time and child ids, only one of them is the valid one

5.2 Minimal time within stream

To release the item from the barrier we need to ensure that there are no in-flight tombstones.

Lemma 1. If data item D has global time GT greater than the global time of each in-flight element, then all tombstones for that item had already arrived at the barrier.

Proof. Let D_{tomb} be an in-flight tombstone for D. According to the definition of the tombstone item, D_{tomb} and D has the same global time GT. We assumed that there are no in-flight element with the global time equal to GT. Contradiction.

New tombstones for D cannot be generated because items with global time greater than GT cannot trigger replay that affects D.

This implies that if the stream does not contain items with the global time less than or equal to GT, then all tombstones for D had already arrived at the barrier. □

Therefore, to output an item from the barrier, we should ensure that there are no items in the stream with the global time less than or equal to the global time of this item.

To track the global time of in-flight items we adopt an idea of *acker task* inspired by Apache Storm [3]. Acker tracks data items using a checksum hash. When the item is sent or received by an operation, its global time and checksum are sent to the acker. This message is called *ack*. Acker groups acks by a global time into the structure called *ack table*. Once acker receives an ack message with global time GT and XOR it updates GT entry in the table by xoring XOR with the current value. When an item is sent and later received by the next operation, xoring corresponding XORs would yield zero.

Acks are overlapped to nullify table's entry only when an item arrives at the barrier. That is, ack for receive is sent only after both processing and the ack sending for the transformed item, as illustrated in Figure 7. Different shapes of items mean different payloads. The ack for the sending of the triangular element is sent before the rectangular one. We expect the channel between the acker and each operation to be FIFO, so ack for the triangular item would be xored before the rectangular. So the two equal values are separated by distinct one.

This technique guarantees that the XOR for some global time is equal to zero only if there are no in-flight elements with such global time.

The minimal time within a stream is the minimal global time with non-zero XOR. On minimal time changes, acker broadcasts new minimal time to the barrier and operations. Therefore, the barrier can release elements with global time GT once it received notification from acker that the minimal time within the stream is greater than GT.

To ensure that no fronts can generate item with the specific timestamp, each front periodically sends to acker special message called *report*, which promises that front will not generate items

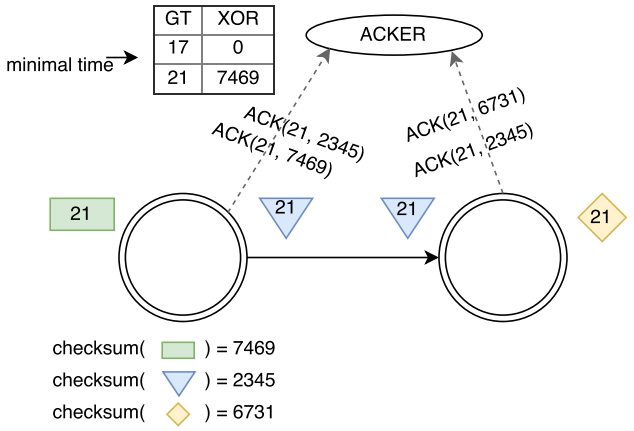

Figure 7: The example of tracking minimal time using acker

with a timestamp lower than the reported. The value in the ack table can become zero only after the corresponding report arrives.

The proposed mechanism could be isolated by hash range. This change allows us releasing from barriers on different workers independently. This feature is also known as early key availability.

5.3 Distributed runtime

FlameStream is implemented in Java, using Akka framework for messaging and Apache Zookeeper for cluster state management. The usage of Zookeeper mitigates the need for the dedicated master node.

From a user perspective, the system provides an API to define processing in terms of *ticks*. Tick is represented by a time interval and the graph that handles data items, which relate to this interval, according to the global time. To deploy a tick, the client writes it directly to Zookeeper, whereas Zookeeper notifies workers when the new tick appears. As soon as tick ends, the system starts to execute the next one if any was submitted to Zookeeper. Otherwise, processing is completely stopped.

The underlying purpose of ticks is to provide an ability to rebuild and redeploy graph periodically. We keep in mind that the possible client of FlameStream is a system that automatically builds a graph based on declarative language. However, such system cannot build it once and use without modifications, because the performance of graph execution can be influenced by the current load, the performance of operations, metrics, etc. Hence, frequent graph redeployment is crucial for our system.

5.4 Fault tolerance

Currently, fault tolerance mechanisms are not among the goals of our prototype. However, it is worth to share how they can be implemented in our system.

Typically, distributed systems take into consideration the following types of failures:

- Packet loss
- Node failure
- Network partitioning

Acker can determine the packet loss issue if it happens. Therefore, the part of the stream can be replayed by the fronts subsystem if it supports some reliable buffer.

Node failure also can be readily determined by the acker. The only difficulty is the loss of the grouping state. Hence, there is a need for grouping state replication.

Support of network partitioning tolerance is not planned because it leads to an unavoidable loss of data. We believe that in this case, stream processing does not make sense.

6 EXPERIMENTS

6.1 Setup

We performed the series of experiments to estimate the performance of our system's prototype. As a stream processing task, we apply building an inverted index. This task is chosen because it has the following properties:

(1) Task requires stateful operations. It allows us to check the performance of the proposed stateful pipeline
(2) Computational flow of the task contains network shuffle that can violate the ordering constraints of some operations. Therefore, inverted index task can verify the performance of our optimistic approach

It is worthwhile to mention that building inverted index can be considered as the halfway task between documents generation and searching. In the real-world, such scenario can be found in freshness-aware systems, e.g., news processing engines.

Building of inverted index is implemented in terms of MapReduce transformations. We start with page mapping into the pairs *(word; word positions within the page)*. After that, word positions are reduced by word into the single structure. We assume the output of the stream to be change records of the inverted index structure, i.e., each input page triggers the output of the corresponding updates.

Pairs *(word; word positions within the page)* must be ordered by page id and version before the update of inverted index state. Otherwise, it is possible to obtain the inconsistent index, if there are multiple versions of the same document.

In FlameStream this algorithm is implemented as the typical conversion of MapReduce transformation, which is shown in section 3. Inverted index structure plays the role of an accumulator, and the accumulator map produces the most recent changes of this structure if any.

By the notion of *latency* we assume the time between two events:

(1) Input page is taken into the stream
(2) All the change records for the page leave the stream

Our experiments were performed on the cluster of Amazon EC2 micro instances with 1GB RAM and 1 core CPU. We used 10000 Wikipedia articles as a dataset.

6.2 Overhead and scalability

The proposed method can bring an additional overhead. Thus, it is essential to analyze the system's behavior under different configurations.

We take the ratio of arrived at the barrier items count to the number of the valid items among them as a key metric for the

estimation of the overhead of our prototype. This value measures the extra cost of our approach.

The relation between the number of workers, the input document's average rate, and the proposed ratio is shown in Figure 8. As expected, the peak of the ratio is achieved when the document per second rate is high, and the number of the nodes is low. This behavior can be explained by the fact that a few workers cannot effectively deal with such intensive load. Nevertheless, the proportion of invalid items reduces with the increase of workers number. Under not very high pressure, the total overhead of the optimistic approach is under 10% for all considered number of workers. These results confirm that the ratio does not increase with the growth of the number of nodes.

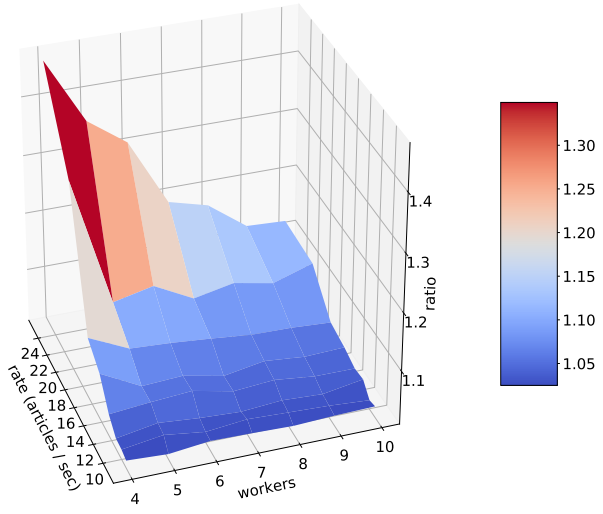

Figure 8: The relation between the number of workers, the input document's average rate and the replay ratio

The latencies of FlameStream across multiple workers for the fixed document rate of 70 ms are shown in Figure 10. This figure demonstrates that latency is not significantly increased with the growth of the number of workers.

Therefore, the most important conclusions of these experiments are: the proposed method is scalable, the overhead could be optimized by system setup.

6.3 Comparison against Apache Flink

One of the most important goals of the experiments is the performance comparison with an industrial solution regarding latency. Apache Flink is chosen for evaluation because it is state-of-the-art stream processing system that provides similar functionality and achieves low latency in the real-world scenarios [8].

For Apache Flink, the algorithm for building the inverted index is adopted by the usage of *FlatMapFunction* for map step and stateful *RichMapFunction* for reduce step and for producing the change records. Order enforcing before reduce is implemented using custom *ProcessFunction* that buffers all input until corresponding low

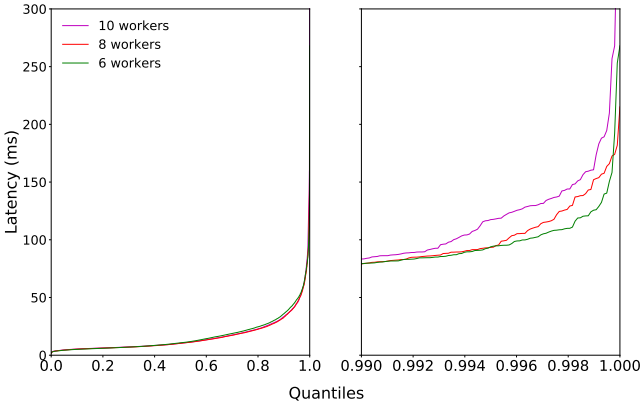

Figure 9: FlameStream latency distribution. Left - the whole distribution, right - high quantiles

watermark is received. Watermarks are sent after each document. The network buffer timeout is set to 0 to minimize latency.

In this paper, we compare 50^{th}, 75^{th}, 95^{th}, and 99^{th} percentile of distributions, which clearly represent the performance from the perspective of the users' experience.

The comparison of latencies between FlameStream and Flink within 10 nodes and distinct document rates is shown in Figure 10. In this case, FlameStream provides lower latency even under high load. These results confirm that optimistic approach for deterministic processing is able to provide less latency than conservative methods. Firstly, the reason for better performance can be the fact that Flink starts to update index only after the buffer before reduce stage is flushed. In contrast, FlameStream flushes its barrier right before data is sent to a user, according to its optimistic nature. At this moment, all corresponding computations have been already done. Secondly, low watermarks go along the stream and can be delayed by long-running operations, while acker processes ack messages independently. It is confirmed by Figure 11, which shows the comparison between waiting time in Flink's buffer and FlameStream's barrier.

However, there are conditions, which are not suitable for the optimistic approach. Figure 12 shows the comparison of latencies between FlameStream and Flink within 5 nodes and distinct document rates. Flink outperforms FlameStream under extreme load. Such behavior follows from the fact that FlameStream provides significant overhead under very high pressure within a few computational units. This result evidently corresponds with measurements of the overhead in Figure 8. Nevertheless, it should be noted that FlameStream demonstrates better latency under non-extreme load.

Thus, Flink can be more appropriate if there is a need to optimize computational resources under a fixed load, but the demands on latency are not very strict, or determinism is not required. FlameStream is more relevant for cases when low latency and determinism are strict requirements, but an allocation of additional resources is not a problem.

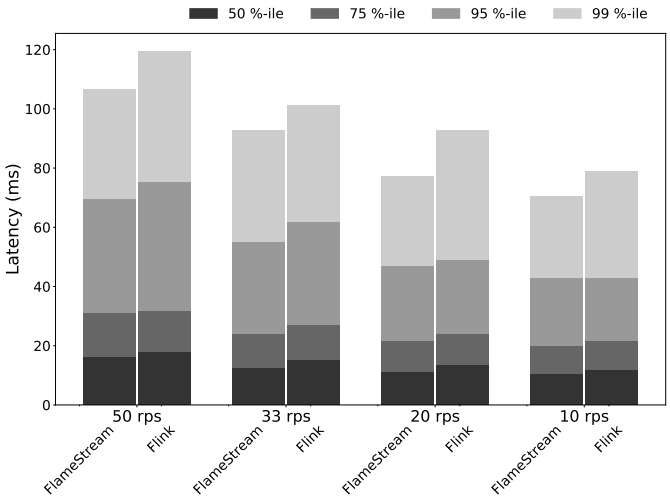

Figure 10: The comparison in latencies between FlameStream and Flink within 10 nodes and distinct document rates

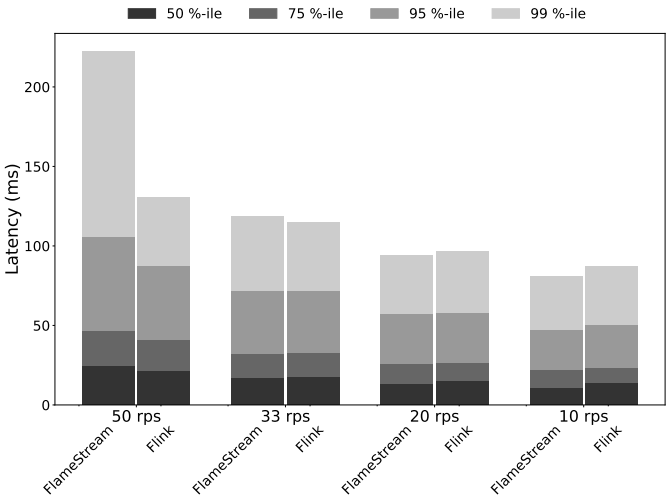

Figure 12: The comparison in latencies between FlameStream and Flink within 5 nodes and distinct document rates

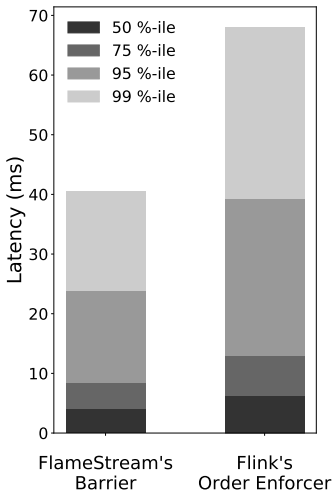

Figure 11: Comparison between waiting time in Flink's buffer and FlameStream's barrier

7 RELATED WORK

Data flow: One specific detail of our computational model is cyclic data flow graph support. Naiad [13] by Microsoft Research provides an implementation of this idea. Nevertheless, Naiad applies cycles only for iterative computations and allows for each operation to have its own state.

Another similar concept of Naiad is the usage of logical timestamps to monitor progress. However, to propagate the latest timestamp the pessimistic approach of notifications broadcasting is defined. Therefore, with the assumption of infrequent out-of-order items, our optimistic behavior is more relevant.

In our model, map and group operations are used as core processing primitives. Google Dataflow [2] provides the same idea. The primary distinction is that Google Dataflow has different state model which is not applicable to real-world MapReduce stream processing tasks. Additionally, this model provides different window types for grouping. FlameStream grouping is aligned with fixed-sized sliding window, but it is possible to implement other kinds of windows by using cycle and grouping with window-affiliation hash.

State: The common approach to state management is to give a user the ability to handle a state of almost any supported operations. Such behavior is implemented, for instance, in Apache Flink [7], Storm [3], Samza [14], Naiad [13]. To the best of our knowledge, FlameStream is the only open-source stream processing system that:

- Stateless in terms of business-logic
- Supports any MapReduce transformations

Deterministic processing and handling out-of-order items: Research works on this topic analyze different methods, but most of them are based on buffering.

K-slack technique can be applied, if network delay is predictable [5]. The key idea of the method is the assumption that an event can be delayed for at most K time units. Such assumption can reduce the size of the buffer. However, in the real-life applications, it is very uncommon to have any reliable predictions about the network delay.

IOP and OOP architectures are popular within research works and industrial applications. IOP architecture is applied in [4, 9]. OOP approach is introduced in [12] and it is widely used in the industrial stream processing systems, for instance, Flink [7] and Millwheel [1]. However, these methods require buffering all input items before each order-sensitive operation.

In [20], the mechanism to control the trade-off between determinism and low latency is proposed. However, such approach only

provides for relaxing determinism properties to achieve low latency if needed.

Regarding optimistic techniques, there is less scientific and industrial activity. In [19] so-called *aggressive* approach is proposed. They introduced an idea of deletion messages that is very similar to our tombstone items. However, authors describe their idea in an abstract way and do not provide any techniques to apply their method for the arbitrary operations. Another optimistic strategy is detailed in [11]. This method is probabilistic: it guarantees the right order with some probability. Besides, it supports only the limited number of query operators.

Tracking mechanisms within stream: One important task that FlameStream faces is handling of the minimal global time. In Apache Storm [3] acker is used for items tracking. We use acker to track the least global time of in-flight items and to detect package losses.

8 CONCLUSION AND FUTURE WORK

We recognized two issues that can increase the complexity of stream processing workflows:

- Direct state handling can distract from the main task and is error-prone
- The lack of determinism implies insufficient ability of verification and composition

In this paper, we presented stream processing model that resolves these problems. Particularly, it has the following properties:

- State management is implemented using drifting state technique, which allows the state to be the part of the stream in the form of an ordinary data item. Such approach relieves user from the direct state handling
- Novel optimistic approach for handling out-of-order items is designed to achieve determinism

Both features are able to significantly simplify the development of stream processing workflows because the system addresses problems, which were previously forwarded to a software developer.

Besides, we implemented the prototype of the proposed model and deeply analyzed its performance and imposed overhead. The series of benchmarks within different computational layouts demonstrated the scalability of the proposed framework. These experiments also showed that our system can outperform the alternative.

In the future, the following features are planned to be implemented:

- Fault tolerance, and, hence, at least once and exactly once guarantees
- Acker can be isolated by hash range. This change allows us releasing from barrier independently also known as early key availability

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[2] 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. *Proc. VLDB* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Apache Storm 2017. Apache Storm. (Oct. 2017). http://storm.apache.org/

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[5] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. 2004. Exploiting K-constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Trans. Database Syst.* 29, 3 (Sept. 2004), 545–580. https://doi.org/10.1145/1016028.1016032

[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink&Reg;: Consistent Stateful Distributed Stream Processing. *Proc. VLDB* 10, 12 (Aug. 2017), 1718–1729.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symp. Workshops (IPDPSW)*. 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[9] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 647–651. https://doi.org/10.1145/872757.872838

[10] flink-state 2018. Apache Flink documentation, Working with State. (Feb. 2018). https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/stream/state/state.html

[11] Chuan-Wen Li, Yu Gu, Ge Yu, and Bonghee Hong. 2011. Aggressive Complex Event Processing with Confidence over Out-of-Order Streams. *Journal of Computer Science and Technology* 26, 4 (01 Jul 2011), 685–696. https://doi.org/10.1007/s11390-011-1168-x

[12] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order Processing: A New Architecture for High-performance Stream Systems. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 274–288. https://doi.org/10.14778/1453856.1453890

[13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[14] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[15] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proc. PODS (PODS '04)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/1055558.1055596

[16] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. https://doi.org/10.1145/1107499.1107504

[17] storm-site 2018. Apache Storm documentation, Storm State Management. (Feb. 2018). http://storm.apache.org/releases/1.2.1/State-checkpointing.html

[18] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowl. and Data Eng.* 15, 3 (March 2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[19] Mingzhu Wei, Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal Claypool. 2009. Supporting a Spectrum of Out-of-order Event Processing Technologies: From Aggressive to Conservative Methodologies. In *Proc. of the 2009 ACM SIGMOD Intnl. Conf. on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 1031–1034. https://doi.org/10.1145/1559845.1559973

[20] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. 2017. Maximizing Determinism in Stream Processing Under Latency Constraints. In *Proc. of the 11th ACM Intnl. Conf. on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 112–123. https://doi.org/10.1145/3093742.3093921

[21] 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 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664