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

Current generation of scalable distributed systems designed for analytical processing of large volumes of data have addressed drawbacks of previous generation. However, several issues still remain. Particularly, existing solutions suppose that the state of operations should be managed directly by user.

FlameStream is a low-latency oriented distributed framework for executing analytical workflows. We offer a computational model that relieves user of explicit state handling and corresponding distributed environment that guarantees "exactly once" processing. The experiments show that prototype of our system is able to outperform alternative solutions.

1 INTRODUCTION

A need to process huge amounts of data (e.g. Internet scale) was addressed by distributed data processing systems such as Google's MapReduce [7] and Apache Hadoop [1]. These systems are able to run in a massively parallel fashion on clusters consisting of thousands of commodity computational units. The main advantages of this approach are fault-tolerance and practically unlimited scalability.

Several drawbacks of the initial models and architectures of this kind are deeply analyzed in [8]. Many of these drawbacks were addressed in the next generation of scalable distributed data processing architectures, e.g. Asterix [4], Spark [13], and Flink [6].

The goal of the FlameStream is to address special case of the data processing: real-time stream-based computing. This case emerges in short-term personalization, real-time analytics and other problems demanding both low update latency and flexibility of the computational subsystem. Specifically, the objectives are:

- Support map/reduce computations
- Provide "exactly once" semantics
- Be optimistic in terms of collision management
- Be able to optimize computational layout based on execution statistics
- Let the keys to get ready independently

EDBT 2018, March 26-29, 2018, Vienna, Austria

ACM ISBN 978-3-89318-078-3.
https://doi.org/

These nice properties we trade for in-memory processing: the total state volume must be less than the total available memory of the compute units. We analyzed where the states are coming from and reshaped the operations to optimize the state management while remaining map/reduce complete.

The optimistic collision management is done over collisions that already happen and it is often more lightweight then prevention mechanisms. This property is proided by the ability of the system to replay certain fragments of the stream. These replays demand the computation process to be deterministic. As many other systems, we use the graph representation of the computation and dynamically optimize this graph using statistics available.

"Exactly once" semantic is an arguable topic, we provide this guarantee inside our system but the overall contract could be violated because of source/sink flaws. To achieve this objective we introduce dynamic batching technique, based on Storm [2] ackers idea. The same mechanism allows us to get early key availability.

We follow these principles in our system architecture:

- The set of basic operations should not require state handling from the user. At the same time, it should provide for specification of arbitrary processing workflows, similar to those found in Spark and Flink.
- State is the system internal term and is available to the developer as a simple coupling abstraction.
- The processing is divided into ticks. Each tick could use different execution graph. These graphs must be equivalent in terms of the resulting stream, but can layout the computations differently.
- The computation is deterministic: repeated processing of the same data should yield the same output.
- Exactly once execution: each output item is valid and get out of the system exactly once even in case of partial system failures.

Enlisted principles are reflected in the following design. The computation to be performed with FlameStream is specified in terms of predefined operations that are parametrized with user-defined procedures (collectively called *business logic* in the jargon of developers community). Each operation accepts a stream or several streams of data items, and produces one or several output streams. The whole computational workflow is described in terms of a graph. The nodes of the graph represent operations, while edges set up logical routing of data items between operations.

The FlameStream runtime organizes the data items to be processed by operations into prioritized queues. The logical workflow graph is replicated to every compute unit in the cluster for scalable execution. The data items are partitioned between computational units based on user-defined hashing. This allows user to influence the computational layout and balance the load of the compute units if needed.

FlameStream provides the only operation that maintains state called *grouping*. Grouping does not require any state management by a user. Moreover, the structure of its state can be easily made persistent, which is useful for capturing snapshots.

In this paper, we introduce the basis for an implementation of the FlameStream and demonstrate advantages of the proposed system with the prototype implemented in Java on top of Akka platform. The contributions of this paper are the following:

- Definition of the MapReduce complete computational model without stateful user-defined procedures
- Development of the mechanisms that provide “exactly once” guarantees
- Introduce the optimistic collision management schema and demonstrate its performance competitiveness

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

2 COMPUTATIONAL MODEL

In this section we describe the computation structure of the proposed system. We will focus on operation set definition and computational layout. The guarantee mechanisms will be discussed in separate section 3.

2.1 Data flow

The top level of our data flow abstraction is a *stream*. Stream is represented by an ordered unlimited sequence of data items. Data item is a *payload* and a *meta-data* associated with it.

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

Payload is an arbitrary user-provided data. Meta-data is a structured system-assigned information. The primary purpose of the meta-data 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 payloads by assigning them meta-data. Inside stream, data items can be dropped or their payloads and metas can be transformed. Eventually, barrier removes meta-information and outputs back pure payloads. High-level view of our stream is shown on the figure 1.

2.2 Computaion 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. The figure 2 shows the example of data flow graph.

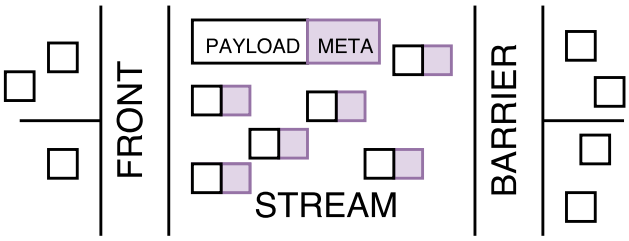

Figure 1: Data flow

Our model allows cycles in the graph while data flow graphs are commonly assumed to be acyclic (DAGs) [5, 13]. 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 computation units. Each computational unit runs a process called *worker*, and each of the workers executes complete data flow graph. Every worker is assigned by an integer interval (hash range). 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 assosiated 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. Though the hash ranges assignment is optimized by the system according to the processing statisticts. Possible partitioning of this graph onto two nodes is shown on the figure 3.

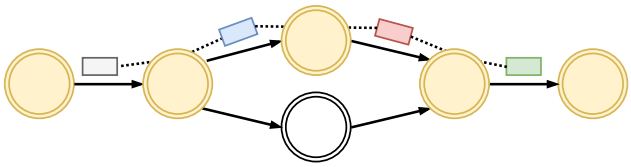

Figure 2: An example of the data flow graph

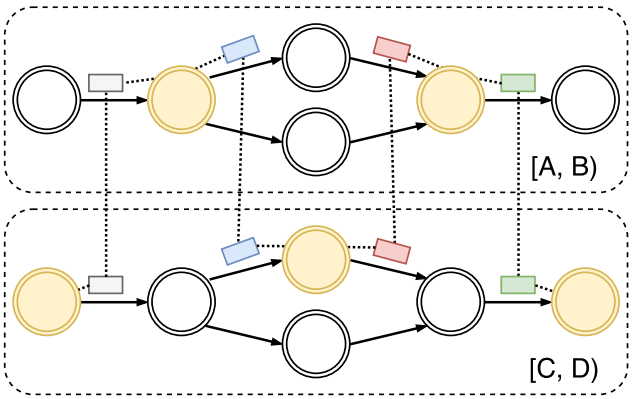

Figure 3: Possible partitioning of the data flow graph

2.4 Ordering model

As it was mentioned above, meta-information imposes an explicit total order on data items. Without diving into details, it should be noted that the order of items is maintained across different fronts.

Let *image* be an output data item of an operation and *preimage* is a corresponding input item. In this terms, the order of images is preserved, i.e. the order of images is the same as the order of preimages. Moreover, the image of the item follows its preimage but precedes the next item. The concept of ordering is shown on the figure 4

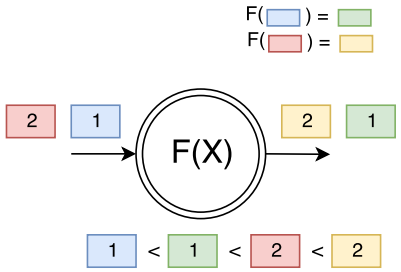

Figure 4: Ordering

We assume that input items of the operations are strictly ordered. After the meta-data is assigned to the data item at the front, the rest of computations become deterministic.

2.5 Operations

The set of available operations is limited by the following list.

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 or sinks.

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 according to the ordering model. 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 maintains state. To prevent collisions user can define equivalence relation to ensure that items with distinct payloads are got in the distinct buckets.

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:

$$(1), (2), (1|3), (2|4), (1|3|5), (2|4|6), (3|5|7), (4|6|8)\ldots$$

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 hash function are independent.

2.6 User-defined parameters

User is able to set up the following parameters:

(1) Computation 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. 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.

2.7 MapReduce transformations on stream

In this section, we demonstrate possible implementation of MapReduce-like transformation on a stream in order to show the soundness of our model. Map stage can be formulated in terms of our map operation. However, it is not obvious how to implement reduce stage, because it requires an explicit state handling. The algorithm 1 shows a generic reduce stage. The accumulator is an explicit state that should be maintained between subsequent iterations.

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
```

One possible way to implement reduce stage in our model is to make business logic state a part of the stream and work with it like with an ordinary data item. The figure 5 shows a generic graph for MapReduce computation. Map and reduce parts are highlighted with a dashed line.

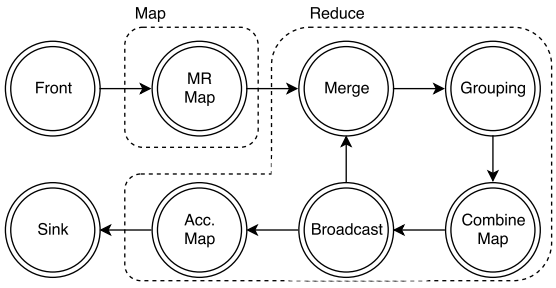

Figure 5: Logical graph for MapReduce transformations

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 in order to combine them together 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 key 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.

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. As the accumulator is the actual result of the reduce stage the accumulator map can be omitted.

The example of input/output items, which are generated/ transformed by the part of the logical graph, is shown on the figure 6. 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. The 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 4 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.

3 OPTIMISTIC COLLISION MANAGEMENT

As it was defined previously, only the grouping operation maintains a state and the state depends on the order of incoming items.

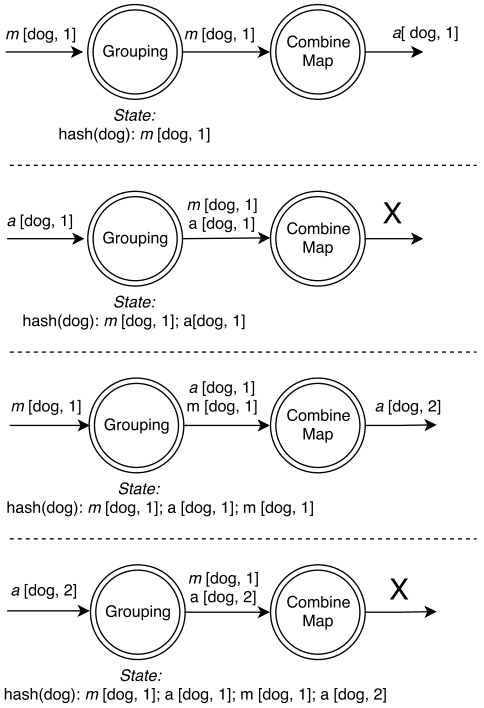

Figure 6: Part of the stream evalutaion for word counting

Because of asynchrony and the possible existence of multiple paths between two nodes it is hard to deliver the right order.

In order to address this issue, 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 the 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.

The example of replay is shown on the figure 7. The colored item is out of order and the window of grouping is 2.

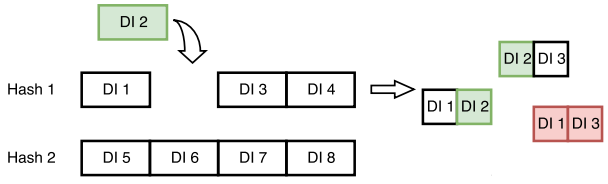

Figure 7: Replay in grouping

In the case of the right order of the input items, there are no redundant items produced. Nevertheless, replay can emit more relevant items, and former ones should not be sent to the outer world.

3.1 Adaptive microbatching

In order to filter out invalid elements that are produced on grouping replay, items are collected at barrier before leaving the system.

To differentiate a freshness of items we utilize the structure of meta-information that implements FlameStream ordering model, defined in section 2. Then, we introduce an invalidation order on meta that distinguishes correct elements from incorrect ones.

3.1.1 Meta information. The meta-information of data item is a pair of a *global time* and a *trace*.

$$Meta := (GlobalTime, Trace)$$

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 are not relying on any clock synchronization between nodes, but we require a strict monotonicity within the single front. The only implication of the clock skew is the system degradation in terms of latency: 1ms of the nodes clock difference appends 1ms to minimal latency.

Trace is an array of logical times of operations and the ordinal number of the output within single input which we call *child id*.

$$Trace := [LocalTime]$$

$$LocalTime := (LogicalTime, ChildId)$$

The logical time is a simple item counter within each operation. Therefore, child id is required as map and broadcast can generate multiple items from one. Items with the same child id are called *siblings*. When an item leaves an operation, its trace is appended with a new local time. The traces are compared lexicographically.

In spite of the fact that initially there are no items with the same global time, they can be generated by some operations. The trace is used to distinguish two elements with the same global time.

Our concept of meta-information is similar to vector clocks [9, 10]. However, unlike vector clocks, meta-information provides for the total order of data items due to the global time.

The figure 8 shows the topology of each operation and how it affects the trace of local times.

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

- Items with common prefixes are produced from the same item
- Item with lower meta is generated earlier, at front or in the stream

3.2 Invalidation mechanism

Replay in the grouping can generate multiple tuples with the same last item. Only one tuple from such set is correct. To find out which tuples are correct we introduce the invalidation rule on data items: if there are several tuples with the same last element, the most recent one is the correct one.

The item A is said to be invalidated by item B and B is called *invalidator* of A, or simply *invalidator* iff:

(1) They have the same global time
(2) The trace of A is lexicographically less than the trace of B
(3) The first difference in traces is in logical time

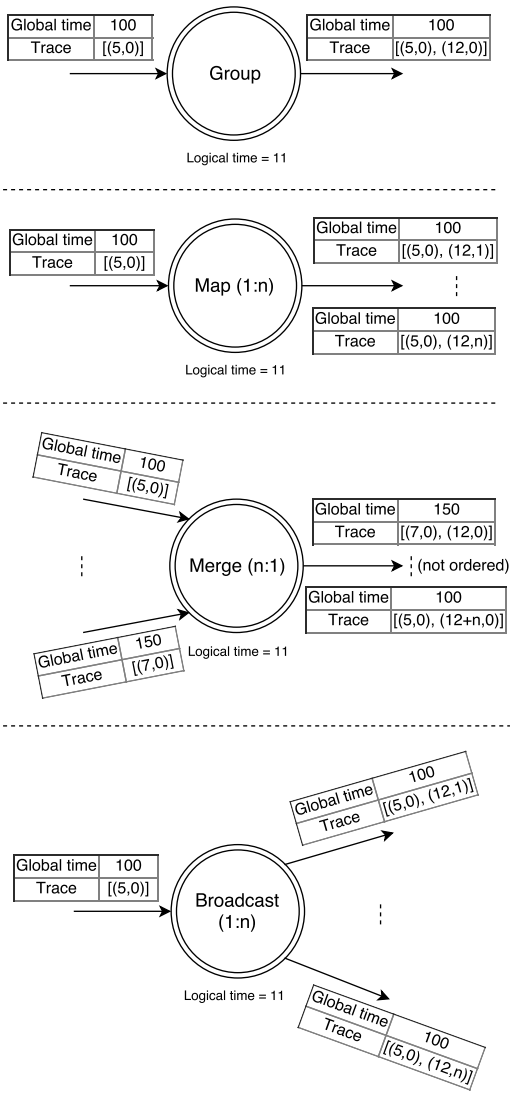

Figure 8: Meta-information handling

If the first difference is in child id, e.g. they were generated by the broadcast operation, there is no invalidation order between them. Hence, the invalidation order is a partial order.

Invalidation order is defined not only for the grouping output but for all items. Stateless operations cannot distinguish items thus act on them identically. However, the invalidation order will not be lost, when an item goes through operations because the trace of local times is append-only.

Barrier maintains a buffer of items. Once a new item arrives, it is inserted into the buffer and items that are invalidated by newcomer are removed. The pseudocode is shown in the alogorithm 2.

There are two complications with invalidation mechanism. The first one, barriers are deployed on multiple workers and are partitioned by the business-logic balancing function. Hence, the item and corresponding invalidator can arrive at distinct barriers. As defined earlier they have the same global time, so the partitioning of barriers by global time solves this problem. The second one, it is unclear when the items should be released from the barrier. The system should ensure that there are no in-flight invalidators.

Algorithm 2 Inserting element in buffer

```
function INSERT(a)
    for all item ∈ buffer do
        if item is invalidated by a then
            DEQUEUE(buffer, item)
        end if
    end for
    ENQUEUE(buffer, a)
end function
```

The figure 9 illustrates possible barrier issues. Triangular items with labels 4 and 7 got out from the system, despite the fact that they should be invalidated by corresponding rectangular items.

The solution of this problem is described in the next subsection.

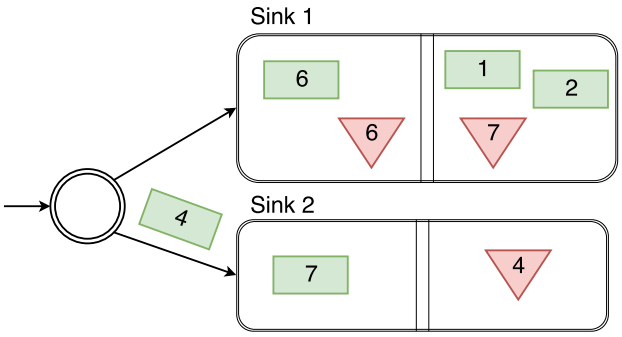

Figure 9: Possible barrier issues

3.3 Minimal time within stream

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

Lemma 1. If data item D has global time GT greater than the global time of the in-flight elements, then all items that could invalidate D had already arrived at the barrier.

Proof. Let E is in-flight element that invalidates D. According to the definition of invalidation order, E and D have the same global time GT, but the different trace. We assumed that there are no in-flight element with the global time equal to GT. Contradiction.

This implies that if the stream does not contain items with the global time less than or equal to GT, then all items which invalidate 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* borrowed from Apache Storm [2]. 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 the figure 10. 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.

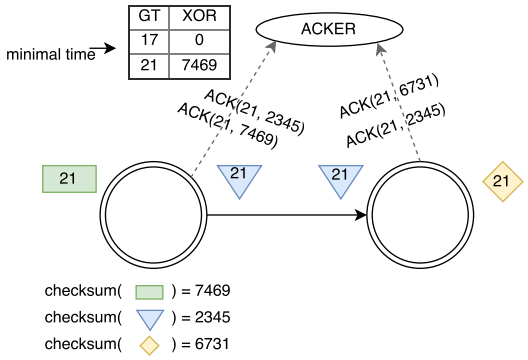

Figure 10: Acker

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 are able to generate item with the certain timestamp, each front periodically sends to acker special message called *report*, which promises that front will not generate items with a timestamp lower than the reported. The value in the ack table can become a zero only after the corresponding report arrives.

The proposed mechanism could be isolated by hash range. This change allows us making invalidation and releasing from barrier independent also known as early key availability.

4 IMPLEMENTATION

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

From a user perspective, system provides an API to define processing in the 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, client writes it directly to Zookeeper, whereas Zookeeper notifies workers when the new tick appears. As soon as tick ends, system starts to execute the next one if any was submitted to Zookeeper. Otherwise, processing is completely stopped.

The underlaying purpose of ticks is to provide an ability to periodically rebuild and redeploy graph. We keep in mind that the possible client of FlameStream is a system that automatically builds 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 current load, the performance of operations, metrics, etc. Hence, frequent graph redeployment is a crucial for our system.

4.1 Fault tolerance

Currently, there are no fault tolerance mechanisms implemented in our prototype. However, it is worth to share few initial ideas on the topic.

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 kind of reliable buffer.

Node failure also can be easily determined by the acker. The only difficulty is the loss of the grouping state, because it can lead to the fail of exactly-once semantics. Hence, there is a need for grouping state replication.

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

5 EXPERIMENTS

We evaluated the prototype of the system by incrementally computing inverted index.

The computation 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 a change log of the inverted index structure. More precisely, each input page should trigger the output of the corresponding changelog.

In the real-world, such scenario can be found in freshness-aware systems e.g. news processing engines. By the notion of *latency* we assume the time between two events:

(1) Input page is taken into the stream;
(2) The last item of the corresponding changelog leaves the stream.

In FlameStream this algorithm is implemented as the typical conversion of MapReduce transformation, which is shown above. The only difference is that the last map operation outputs a new reduced state of the inverted index as well as a changelog. The changelog is received by a sink but is filtered before it reaches merge. Therefore, the correctness of MapReduce transformation is preserved.

The same task was executed by Apache Flink for performance comparison.

For Apache Flink, this algorithm is adopted by the usage of *FlatMapFunction* for map step and stateful *RichMapFunction* for reduce step and finding the difference.

Our experiments were performed on clusters of 1,2,5,7, and 10 nodes. Each node is an Amazon EC2 micro instance with 1GB RAM and 1 core CPU. We used 10000 Wikipedia articles as a dataset.

The latencies of FlameStream across multiple workers are shown on the figures 11 and 12. These figures demonstrate the scalability of the system in terms of latency.

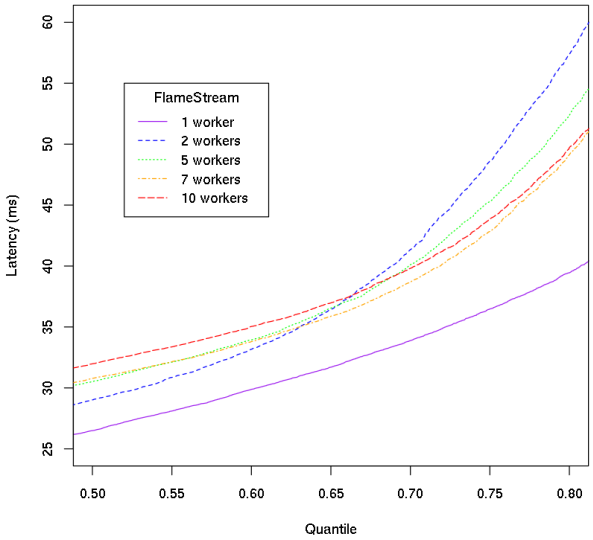

Figure 11: FlameStream median latencies

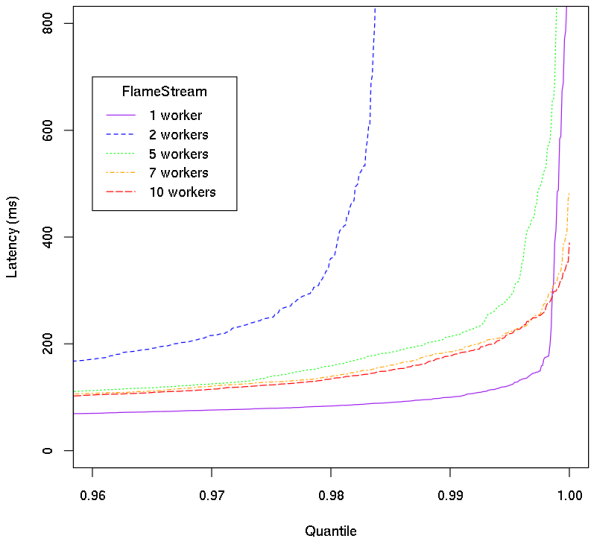

Figure 12: FlameStream tail latencies

The comparison in latencies between FlameStream and Flink within 10 nodes is shown on the figure 13. These results show that FlameStream is able to deliver better latency.

6 RELATED WORK

Data flow: One specific detail of our computational model is cyclic data flow graph support. Naiad [11] 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

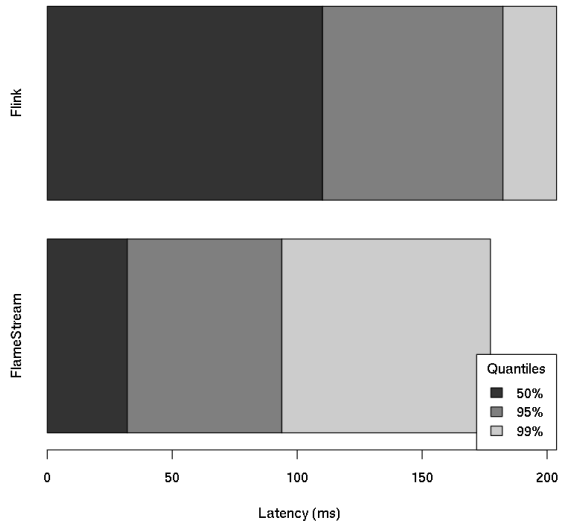

Figure 13: Apache Flink and FlameSream latency distribution comparison

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 [3] 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 [6], Storm [2], Samza [12], Naiad [11]. 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

Tracking mechanisms within stream: One important task that FlameStream faces is handling of the minimal global time. In Apache Storm [2] acker is used to eliminate item traces. Unlike Strom, we use acker to track the least global time of in-flight items and to detect package losses.

7 CONCLUSION AND FUTURE WORK

We introduced the model and implementation of distributed scalable stream processing system. Our key results are the following:

- Proposed computational model is stateless in terms of business-logic. On the other hand, the model is MapReduce complete. Therefore, our system can solve a wide range of analytical and algorithmic tasks.
- The user of the system is able to influence the computational layout. Such approach is motivated by the fact that typically user has deeper knowledge of underlying data distribution.
- The collision management is implemented in optimistic manner. Unlike prevention-based collision management, optimistic approach does not impose overhead in the failure-free case.
- The computational process is deterministic up to the time assignment at fronts.
- The adaptive micro-batching determines a moment when data items can be released from the stream without loosing exactly-once guarantee.
- It was shown that our system is able to outperform Apache Flink on the real-life computations.

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

- Fault tolerance, and, hence, at least once and exactly once guarantees
- Early key availability

REFERENCES

[1] 2017. Apache Hadoop. (Oct. 2017). http://hadoop.apache.org/
[2] 2017. Apache Storm. (Oct. 2017). http://storm.apache.org/
[3] 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 Endow.* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076
[4] Sattam Alsubaiee, Alexander Behm, Raman Grover, Rares Vernica, Vinayak Borkar, Michael J. Carey, and Chen Li. 2012. ASTERIX: Scalable Warehouse-style Web Data Integration. In *Proceedings of the Ninth International Workshop on Information Integration on the Web (IIWeb '12)*. ACM, New York, NY, USA, Article 2, 4 pages. https://doi.org/10.1145/2331801.2331803
[5] 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 Endow.* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777
[6] 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).
[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[8] Christos Doulkeridis and Kjetil Norvaag. 2014. A Survey of Large-scale Analytical Query Processing in MapReduce. *The VLDB Journal* 23, 3 (June 2014), 355–380. https://doi.org/10.1007/s00778-013-0319-9
[9] C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56âĂŞ66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf
[10] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 215–226.
[11] 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
[12] 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
[13] 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