FlameStream: Model and Distributed Runtime for Analytical Stream Processing

Igor E. Kuralenok
Igor's affiliation
St. Petersburg, Russia
ikuralenok@gmail.com

Artem Trofimov, Nikita Marshalkin, Boris Novikov
St. Petersburg state University
St. Petersburg, Russia
trofimov9artem@gmail.com,marnikitta@gmail.com, borisnov@acm.org

ABSTRACT

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

FlameStreamis 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 this implementation can 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 MapReduce [6]. These systems are able to run data processing in a massively parallel mode on a clusters consisting of thousands of commodity computational units. The main advantages of this approach are fault-tolerance and practically unlimited scalability.

However, the initial models and architectures of this kind suffered from several drawbacks are deeply analyzed in [7]. Many of these drawbacks were addressed in the next generation of scalable distributed data processing architectures, e.g. Asterix [3], Spark [9, 16], and Flink [4].

The goal of the FlameStream is to address special case of the data processing: real time stream based computing. This case emerges when we talk on 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

These nice properties we trade for in-memory processing: the total state volume of the system must be less then total available memory of the compute units. To soften this restriction we have analysed where the states are coming from and how to reshape the operations to optimize the state management while being map/reduce complete. The optimistic collision management is done over collisions that already happen and it is often more lightweight then prevention mechanisms. We get this property with the ability 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 arguable topic, we provide this guarantee inside our system but the overall contract could be violated because of source/sink flaws. To get this objective fulfiled we introduce dynamic batching technique, based on Storm Ackers idea. The same mechanism lets us speak on early key availability. We follow these principles in our system architecture:

- The set of basic operations should not require state handling. 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 provided to the developer as 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 layout the computation differently
- The computation is deterministic, that is, repeated processing of the same data should yield 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

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 characterized by meta-information, 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 describe logical routing of data items between operations.

The FlameStream runtime organizes the data items to be processed by operations into queues that are prioritized based on timestamps. The logical workflow graph is replicated to every computing unit in the cluster, providing for scalable execution. The data items are partitioned (shuffled, sharded) between computational units based on hashing.

In contrast with existing solutions, FlameStream provides the only operation that maintains state called *grouping*. Notably, grouping does not require any state management by user. Moreover, the structure of its state can be easily made persistent, which is useful for capturing asynchronous snapshots.

EDBT 2018, March 26-29, 2018, Vienna, Austria
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-3-89318-078-3.
https://doi.org/

The contributions of this paper are the following:

- Definition of the computational model
- Implementation and proof of the concept

The rest of the paper is structured as follows Describe sections here: model 2 implementation 3 experiments 4 related work 5.

2 COMPUTATIONAL MODEL

In our model the dataflow is represented by a directed graph. This graph does not contain information about physical deployment, so further we call it *logical* graph. Items are get into the stream through the node called *front* and get out through the node called *sink*. Fronts and sinks are detailed in the next section; currently, it is only important that fronts and sinks play the role of the graph enter and graph exit correspondingly. Each other node of the logical graph contains single operation, also called job or procedure. Edges show the order of these operations. The data items are processed one-by-one in a "streaming" manner.

Notably, despite the fact that commonly dataflow graphs assumed to be acyclic (DAGs) [4, 16], our model does not have this restriction. Moreover, as we show further in this section, there are cases when cycles are required, e.g. for MapReduce-based algorithms.

In this section, firstly, the structure of data stream items is defined. After that, we introduce supported operations and their properties. Finally, the way to implement any MapReduce transformation using our model is described.

2.1 The structure of data items

Generally, the items of the stream are represented as payload and meta-information. Meta-information is assigned when the payload element arrives at the front.

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

The meta-information of data item is represented as *global time* and the trace of *local times*.

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

Global time is assigned to data item once when the item enters the system. It is represented as the concatenation of milliseconds since the epoch start and the identifier of front. Firstly, such design makes it possible to maintain global time strongly monotonic within single front. Secondly, it avoids assigning the same global time for distinct input elements. Global times can be compared lexicographically.

$$GlobalTime := (frontTs, frontId)$$

Local time is the concatenation of logical time of operation and the ordinal number of output item which we call *child id*. The logical time is represented as simple items counter within each operation. Therefore, child id is required, because some jobs can generate multiple items from one, e.g. flat map. The items with the same child id are called *brothers*. When the item leaves the operation, its trace of local times is appended by new corresponding local time. The traces of local times can be compared lexicographically.

$$Trace := [LocalTime]$$
$$LocalTime := (logicalTime, childId)$$

Notably, 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 of local times is used to distinguish two elements with the same global time. The main purpose of such approach is to provide information for items invalidation which is detailed in the next section. The metas can be compared lexicographically: initially by global time and then by the trace of local times.

It is important to mention that our concept of meta-information is similar to vector clocks [8, 12] . However, unlike vector clocks, meta-information provides for the total order of data items due to the global time.

2.2 Supported operations

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

Map applies specified transformation to the payload of input item. Global time of the item is not changed and the trace of local times is appended by the new corresponding element.

Flat map applies specified function to the payload of input item. This function returns a sequence of new payloads. Each of these payloads is put into distinct data item. Output items inherit global time from the initial. The traces of output items are appended by local times with the same logical time but different child id.

Filter applies specified predicate to the payload of input item. If the result of predicate is positive, filter outputs initial item with updated trace of local times. Otherwise, filter outputs nothing.

Broadcast replicates input item to the specified number of operations or sinks. Similarly to the flat map, the traces of replicated items are appended by local times with the same logical time but distinct child id. Broadcast is the only operation with multiple outputs.

Merge operation is initialized with specified number of input nodes. Each input item from all input nodes is sent to the single output. It should be mentioned that merge operation does not provide any guarantees about the order of items. Global time of the item is not changed and the trace of local times is appended by the new corresponding element. Merge is the only operation with multiple inputs.

Grouping has two properties: number called *window* and hash function. Grouping stores input items in distinct buckets by the value of the hash function applied to payload. When next in turn item is got in the grouping, it is put to the tail of corresponding bucket. After that, grouping outputs window-sized *tuple item*, which consists of the last items of this bucket. If the size of bucket is less than window, all items of bucket are taken. Notably, grouping is the only operation that maintains state.

Here and elsewhere, we assume that hash functions in grouping are perfect, i.e. do not have any collisions. However, practically, user can define equivalence relation to ensure that items with distinct payloads are got in the distinct buffers.

To completely clarify the semantics of this operation, consider the following example. 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:

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

As it can be observed, the result of operation depends on the order of the input items. Currently, we assume that all items are got in the grouping in order defined by global time. The implementation details are described in the next section.

The important property of the grouping is that the result is uniquely determined by the last element in the tuple. Therefore, grouping is the bijective mapping. Additionally, the results among items with different values of hash function are independent.

Tuple item inherits global time from the last element. The trace of tuple item is the trace of the last item appended with the corresponding local time.

Figure 1 shows the topology of each operation and how it affects the trace of local times.

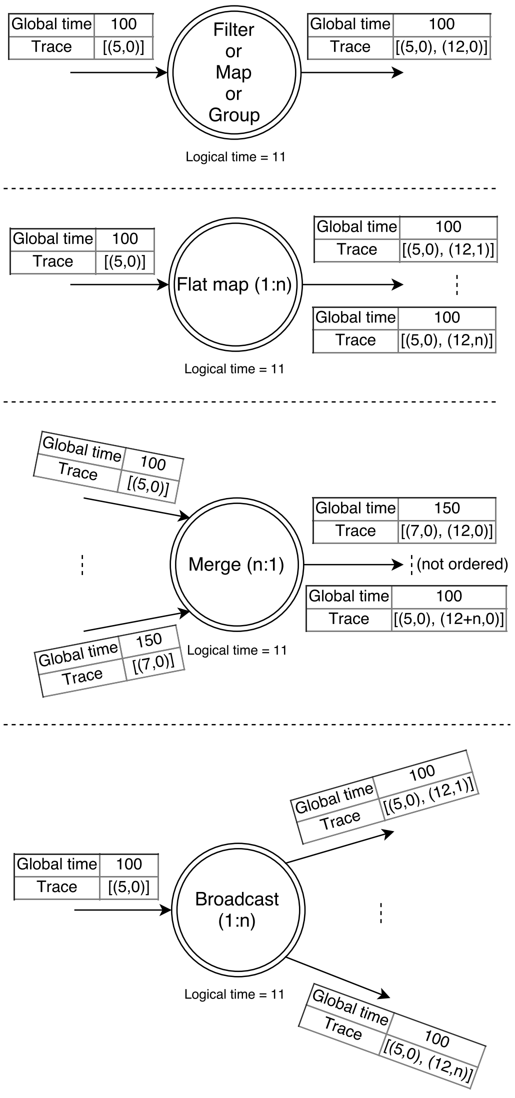

Figure 1: Supported operations

2.3 MapReduce transformations on stream

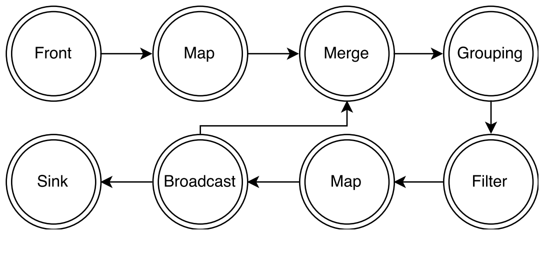

Figure 2: Logical graph for MapReduce transformations

Consider the dataflow that is shown on the figure 2. Data items of this stream are: *input* items, *mapped* items, and *reduced* items. Each type of item and the way they were generated are detailed further in this section. However, it is worth to mention that mapped and reduced items have key-value structure of payload. The operations of the stream have the following properties:

- The first map operation accepts input items and outputs mapped items, according to the business logic. This operation is semantically matched with map step of MapReduce model. Additionally, it should be noted that stream onwards can contain only mapped and reduced items.
- Merge operation is used for cycle implementation.
- The window of grouping is set to 2. The hash function is designed to return distinct values for payloads with distinct keys.
- Filter operation removes tuples with structure *(mapped item; reduced item)*, i.e. tuples, where mapped item was generated before reduced item.
- The all possible inputs of the second map are the following tuples: *(mapped item), (mapped item; reduced item)*. Any other tuples cannot get in this map because of the filter operation and assumption about the order of items which get in grouping. The first type of input tuples is transformed into reduced item with key inherited from mapped item and some initial value. The second type is combined into reduce item with the same key. The value of this item is the result of specified reduce function applied to the values of tuple items. Actually, this transformation is equivalent to the reduce step of MapReduce model.
- Broadcast operation is used to return actual reduced item to grouping and, at the same time, to output it from stream.

According to the provided logical graph, any MapReduce transformation can be implemented using the sequence of map, merge, grouping, filter, and broadcast operations. The key idea is that each reduced item always arrives at grouping right after already combined mapped item and before new one. Hence, each mapped item would be grouped with the actual reduced item. Additionally, when filter accepts tuple *(mapped item; reduced item)*, then it means that mapped item was generated before reduced item, and therefore, it had been already combined into the reduced item. The cycle gives ability for new reduced items to get back in grouping operation. Thereby, the stream reacts to each input item by generating new reduced item, which contains the actual value of the reduce step.

The example of input/output items, which are generated/ transformed by the part of the logical graph, is shown on the figure 3. This example represents classical MapReduce-based algorithm for word counting. Map step of this algorithm transforms each input word into key-value pair where word is the key, and the value is 1. Reduce step sums all values into the final result for specific key. According to our graph for MapReduce transformations, the item *m[dog, 1]* represents mapped item with key "dog" and value 1. The item *r[dog, 1]* describes reduced 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 shown 4 stages separated by dotted lines:

(1) New mapped item with key "dog" arrives at grouping with empty state. Grouping outputs tuple with this single item. Filter accepts the tuple, and Map transforms it to the first reduced item for key "dog" and value 1.

(2) The reduced item arrives at grouping after it went through the cycle. It is grouped in tuple with mapped item that has been already in the state with key "dog". However, filter 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 reduced item in the bucket for key "dog". Grouping outputs tuple containing the reduced item and new mapped item. Filter accepts this tuple, because it has the right order. Map operation combines reduced and mapped items into new reduced items with key "dog" and value 2.
(4) New reduced item arrives at grouping through the cycle, but new generated tuple is not accepted by filter, as well as in the step 2.

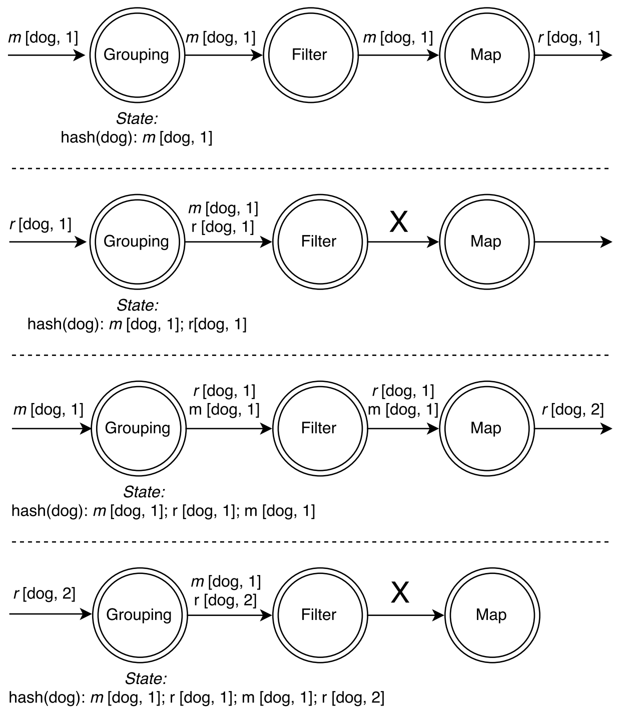

Figure 3: Part of the stream evalutaion for word counting

3 IMPLEMENTATION

3.1 Grouping semantics

The grouping operation was defined in the previous section. Particularly, we assumed that data items get in grouping in the order provided by the meta-information. However, this restriction is hard to satisfy, because of asynchrony and possible existence of multiple paths between two nodes. Therefore, our implementation of grouping satisfies three conditions:

(1) All correct tuples are eventually produced.
(2) All incorrect tuples can be determined.
(3) Only a limited number of incorrect tuples can be generated.

The correctness of tuple means that this tuple would be generated if the order assumption was satisfied.

As it was mentioned, grouping stores all input items in buckets by the value of hash function. Since the order of input items is arbitrary, we can only maintain the order of items within buckets. The next three subsections detail how the conditions of implementation are satisfied.

3.1.1 Replaying. Replaying is used to eventually produce all correct tuples. If the item gets in grouping, according to the meta-information order, nothing is replayed. If the item is out-of-order, all tuples, which contain this element, are reproduced. Thereby, replaying guarantees that eventually all correct tuples are generated.

The example of replaying is shown on the figure 4. In this example the green item is out-of-order and the window of grouping is 2.

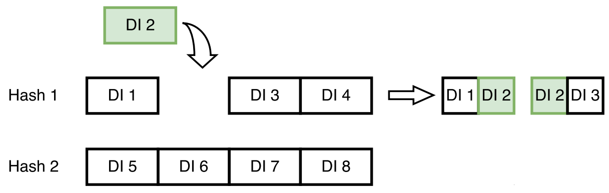

Figure 4: Replaying in grouping

3.1.2 Invalidation. Replaying can generate incorrect tuples, i.e. multiple tuples with the same last item. Therefore, only one tuple from such set is correct. To find out which tuples are incorrect we introduce the invalidation relation between two data items. Therefore, if new item B invalidates item A, item B can replace item A.

The data item A is said to be invalidated by the data item B if:

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

If the first difference is in child id, e.g. they were generated by broadcast operation, there is no invalidation relation between them. Hence, the invalidation relation is a partial order. Notably, the invalidation relation cannot be lost, when item is go through other operations, because the trace of local times is append-only.

3.1.3 Invalidation in grouping. As it was mentioned above, grouping can generate incorrect tuples. Because of the cycles, such tuples can get back in grouping. Therefore, there is a need to remove them from the buckets, because they can lead to the unbounded number of invalid tuples in stream. Such behavior is implemented by removing items from bucket, if the item which invalidates them is arrived. Additionally, in order to generate all correct tuples, invalidation should trigger replaying.

The example of invalidation in grouping is shown on the figure 5. In this example the green item invalidates red item. The window of grouping is 2.

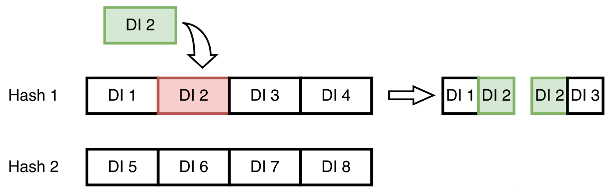

Figure 5: Invalidation in grouping

3.2 Physical deployment and partitioning

Each computational unit in our distributed runtime is assigned by integer interval. Intervals are not intersected and cover the range of 32-bit signed integer. Moreover, each unit contains complete logical graph.

As it was mentioned in the previous section, logical graph does not provide information about physical deployment. Physical graph extends logical one by assigning hash function to each input of each operation. This hash function is applied to the payload of data items and determines partitioning. More precisely, the value of hash function is computed before next operation and corresponding data item is sent to the unit which is responsible for the computed value. Therefore, load balancing explicitly depends on the hash functions of the operations. Optimal balancing requires the knowledge of payload distribution. Hence, the hash functions are assigned by business logic.

Notably, the hash function within grouping should be the same as partitioning hash function for grouping, because it guarantees that the result does not depend on the physical graph deployment. Other operations always do not depend on the physical graph, so they do not have this restriction.

Figure 6 shows the example workflow of logical graph. Possible partitioning of this logical graph on two nodes is shown on the figure 7.

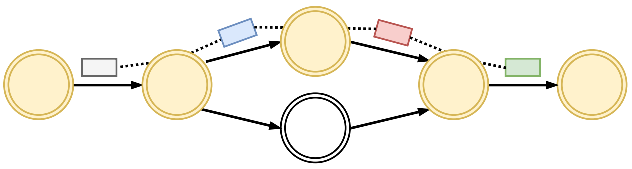

Figure 6: Logical graph workflow

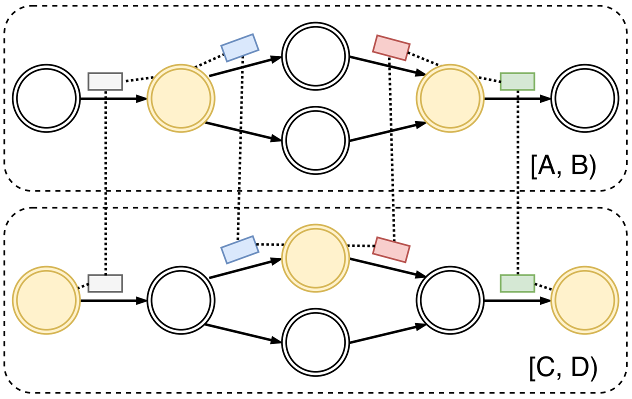

Figure 7: Possible partitioning of the logical graph

3.3 System barriers

3.3.1 Front. Front plays the role of data source. We assume that link between client's data and front is reliable and stable. The failure of this link is considered as the failure of client.

When data payloads are arrived at front, meta-information is assigned to them. As soon as the data obtains its meta-information, it is supposed that the system is responsible for this data.

Each front is connected to corresponding operation. Similarly to the common physical graph vertex, data is sent to the appropriate node according to the value of hash function. Figure 8 shows the example of interaction between client, front, and the other graph components.

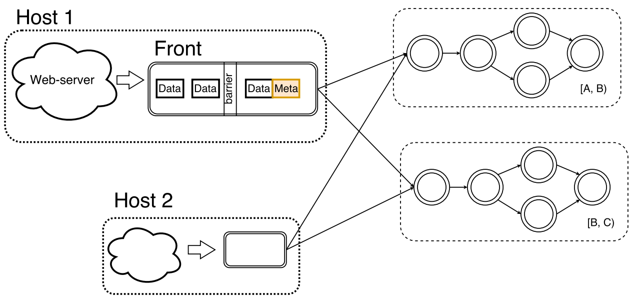

Figure 8: Example of fronts

3.3.2 Sink. Sink is the node which outputs items from the graph. It contains a buffer for data items. The items from buffer are released at some points in time. As it was mentioned above, grouping operation can generate invalid items. Consequently, there is a need to remove them from buffer before they get out from the system.

However, there are two main difficulties. Firstly, sinks are the parts of the physical graph, so they are also partitioned by the business-logic hash function. Hence, item and corresponding invalidation item can arrive at distinct sinks. This issue can be solved by use of special partitioning for sinks by global time. Secondly, it is unclear when the items should be released from the buffer. To do it the system should ensure that there are no in-flight items which can invalidate items in buffer. The solution of this problem is detailed in the next section.

Figure 9 illustrates possible sink issues. Red items with labels 4 and 7 got out from the system, despite the fact that they should be invalidated by corresponding green items.

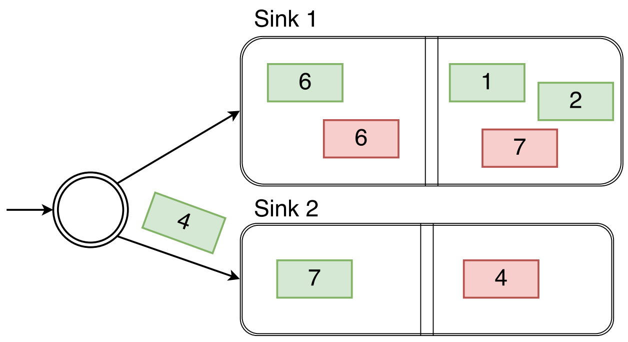

Figure 9: Possible sink issues

3.4 Minimal time within stream

In the previous subsection possible approach to output only correct items from sink was demonstrated. However, there is a need to ensure that there are no in-flight data items which can invalidate items in the sink buffer. In this subsection we offer sufficient condition that stream does not contain such items. Additionally, we describe how this condition is used in implementation.

Claim 1. Let D represent data item in sink buffer and let GT represent its global time. If the items with global time less than or equal to GT do not exist and cannot appear in the stream, then all items that invalidate D had already arrived at the sink buffer.

Proof. Let $D\prime$ invalidate D. According to the definition of invalidation relation, $D\prime$ and D have the same global time GT, but different traces of local times. Let LT and $LT\prime$ be the first distinct local times of D and $D\prime$ respectively. Such difference could appear only as a result of grouping replay. Hence, D and $D\prime$ are tuple items.

The global time of tuple item is inherited from the last item in the tuple, i.e. the last item in tuple $D\prime$ has global time GT. Therefore, considering the properties of grouping operation, $D\prime$ could be generated only if item with global time less than or equal to GT arrived at grouping.

This implies that if stream does not contain items with global time less than or equal to GT and such items cannot appear, then all items which invalidate D had already arrived at sink buffer. □

Regarding this claim, to output item from sink buffer we should ensure that:

(1) There are no items in stream with global time less than or equal to the global time of this item;
(2) Such items cannot be generated.

To ensure that stream does not contain these items, we use module called *acker*. Its idea was proposed by Apache Storm [1]. Acker tracks data item within the stream using a checksum hash. When item is sent or received by operation, its hash is XORed into the checksum. Therefore, if all items arrive at sink successfully, the checksum is zero.

To find out the least global time of the items in stream, checksums are grouped by timestamps of global time into the structure called *ack table*. Hence, if the value of the specific timestamp in the ack table is zero, there are no items with corresponding global time into the stream.

Notably, to ensure that no fronts would generate item with this timestamp, each front periodically sends to acker special message called *report*, which contains the least timestamp that can be assigned to data item by the front. The value in the ack table can become a zero only after corresponding report is arrived.

3.5 System architecture

3.5.1 FlameStream API. To submit the graph, client should describe it in terms of predefined operations, specify fronts, sinks, and wrap them in a *tick*. Tick is a unit of graph deployment. It consists of an *tick start timestamp*, *duration*, *state dependencies*, *hash mappings* and the graph itself. Tick start timestamp and duration specify which data items belong to the tick, according to their global time. State dependencies represent the set of ticks, which the current tick depends on. Hash mapping is a map from a hash ranges to computational units.

Typically, deployment is assumed to be within several seconds ticks with the same graph, and each tick depends on the state of the previous one. In the case of sudden load growth, subsequent ticks could be deployed with different hash mapping, splitting some hash ranges.

We suppose that the main client of FlameStream would be a system which builds graph based on declarative language. However, the performance of graph execution can be influenced by current load, the performance of operations, metrics, etc. Hence, graph redeployment is a crucial goal for our system.

3.5.2 Process model. Figure **??** shows the FlameStream process model. It is a homogenous set of worker nodes and a Zookeeper [10] cluster.

Zookeeper is used to store cluster configuration: cluster management, IP to computation unit mapping, workers and fronts locations, types of fronts and for coordination: state management, ticks, and its statuses. Such extensive usage of Zookeeper mitigates the need for the dedicated master node.

Workers execute graph, one for each tick and hash range and manage part of the state that belongs to the graph. Acker is deployed to one of the workers that is randomly chosen. There is one acker per tick. If the new graph is the same as the old part of the steps can be omitted.

To deploy a tick a client writes it to the Zookeeper. Workers are notified when the new tick appears. They set up a new graph, waits for the relevant state on which tick depends and fetch it once it is ready.

3.6 Fault-tolerance

The runtime supports exactly-once computation semantics. In failure-free case, it is trivial to provide such guarantees. In presence of process failures system should be able to restore consistent state and replay lost items if any. FlameStream here relies on reply property of a front. It is expected that front is able to replay any number of events. Such front could be a file, database table, of message broker e.g. Apache Kafka [11] or Amazon Kinesis [?].

To prevent replaying the whole stream from the beginning consistent snapshots of operators' state is flushed to persistent storage as long as determinants, information that is enough to restore read position, of the fronts, e.g. offsets in Kafka.

The recovery would be a rollback of operators to the most recent snapshot, recovery line, and fronts restore with the corresponding determinants.

Next, we will describe how and when snapshots are taken.

3.6.1 Commit protocol. When the minimal time within the stream is equal to the ticks interval's end there are no elements that belong to the current tick left. Acker initiates a commit protocol as follows:

(1) Acker broadcasts *Commit* message to all participants of the tick: fronts and graph operations
(2) As soon as operation receives commit it flushes state that belongs to the tick and responses with *CommitDone* message
(3) Front responses with its determinant
(4) Once all responses are collected, acker records tick status in Zookeeper

3.6.2 State properties. The most important property of FlameStream's state is that its structure allows starting processing of the next tick without await of the previous one's commit.

Grouping is the only operation in FlameStream that has state and it has a well-defined structure: buckets, that are ordered by meta information, one for each hash **??**. Such bucket has interesting persistent properties. The meta information of the elements from the next tick is strictly greater than previous'(global time is greater). Therefore the bucket would be divided into two parts: prefix that "belongs" to the first tick and suffix to the second.

At commit, grouping saves each bucket's prefix and the next tick won't mess it up.

This is made possible because a common business logic state is a part of the stream. See the map/reduce example, the *reduce item* travels via backward edge. Serialization, copy-on-write behavior, and immutability are already expressed in stateless operations.

3.6.3 *Guarantees.*
At least once = on min time: State + replay + determinism

3.6.4 *Exactly once.*
Exactly once = on min time: State + replay + determinism + idempotent sink

On commit: State + replay = Exactly once

4 EXPERIMENTS

The performance of Flame Stream is compared with the performance of Flink both running on 1, 2, 3, 5 and 10 computers with data set size of 1/3, 2/3, and 1 of the whole data set (describe the data set here).

The latency and overall processing time is measured.

More details on computers and operating environment are needed here.

Put results, preferable charts but tables are also good, here.

The results of experiments clearly show that we are in certain sense good. (Please be more specific here.)

5 RELATED WORK

5.1 Dataflow

Naiad [13] used cycles for iterative processing and logical timestamps to monitor progress. In contrast to their pessimistic approach, using the notification to propagate the latest timestamp, we optimistically produce possibly invalid items and filter them down the stream such approach reduces latencies in reorder-free cases.

Using map and group operations as core processing primitives was expressed in the Dataflow model [2]. FlameStream grouping is aligned with fixed-sized sliding window, other kinds of windows: fixed, session is possible to implement with cycle and grouping by window-affiliation hash. Later approach keeps the model simpier by eliminating window assignment from business logic.

5.2 Delivery guaranties

To track minimal time within stream FlameStream uses acker, an adaptation of Storm's [1] *acker task*. The original parental relationship between messages that induces a tree is extended with invalidation relation between subtrees. Unlike Storm, *xor* optimization couldn't fully free from the burden of tree management. The full history of item's transformation must travel with it, possibly introducing additional overhead.

5.3 State management

TODO after state management chapter.

Flink [5] async snapshots, solve many problems but there are some left. Eg. state management.

Apache Samza [14] should be mentioned but I haven't read the paper yet:).

5.4 Other streaming systems

Spark [15], combines batch and strem. Sacrifice latency.

6 DISCUSSION AND FUTURE WORK

A brief outline of the overall architecture and planned features: types, declarative workflow specification, ? ? ?

7 CONCLUSION

Some of us are coders, but not writers. Hopefully, almost everyone wil prove the previos sentence is wrong.

REFERENCES

[1] [n. d.]. Apache Storm. ([n. d.]). http://storm.apache.org/

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

[3] 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

[4] 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

[5] 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).

[6] 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

[7] 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

[8] 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

[9] Michael Franklin. 2015. Making Sense of Big Data with the Berkeley Data Analytics Stack. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining (WSDM '15)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/2684822.2685326

[10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. Boston, MA, USA, 9.

[11] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing.

[12] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 215–226.

[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] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing (HotCloud'12)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=2342763.2342773

[16] 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