An optimistic approach to handle out-of-order events within analytical stream processing

Igor E. Kuralenok [#1], Nikita Marshalkin [#2], Artem Trofimov [#3], Boris Novikov [#4]

[#] *JetBrains Research*
Saint Petersburg, Russia
[1] ikuralenok@gmail.com [2] marnikitta@gmail.com [3] trofimov9artem@gmail.com [4] 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 experience difficulties with out-of-order data arrival. The most common solution to this issue is buffering. Its main problem is the extra cost for blocking before each order-sensitive operation. The goal of this paper is to propose and evaluate an optimistic approach to handle out-of-order events. We introduce a method that is suitable for any stateful operation and needs a single buffer for the complete computational pipeline. Such technique requires extra network transfers and re-computations, but the experiments demonstrate that a prototype of our approach is able to create low overhead while ensuring the correct ordering.**

I. 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 [1], Samza [2], Storm [3], Heron [4], are able to provide low-latency and high-throughput in distributed environment for this kind of problems. However, providers of the order-sensitive computations remain suboptimal. Most of these systems assume that events are fed to the system with monotonically increasing timestamps or with minor inversions. 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. In this case, order-sensitive operations located deeper in data flow pipeline can be broken. Figure 1 shows the example of common distributed stream processing pipeline that breaks the input order of the operation 2, even if inputs are monotonic and links between operations guarantee FIFO order.

The typical way to achieve in-order processing is to set up a special buffer in front of operation. 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. This issue becomes even more significant if the processing pipeline contains several operations that require ordered input. The problem here is that each buffer increases latency, because of the additional waiting time.

The alternative is to develop business logic tolerant to out-of-order items. However, this approach is suitable only for a limited set of tasks. Moreover, it may dramatically complicate the business-logic, which can lead to the maintenance cost increase and is error-prone.

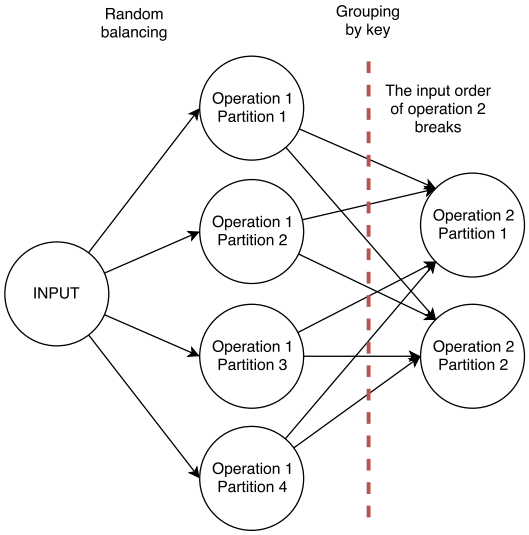

Fig. 1. An example of common distributed stream processing pipeline that breaks the input order of operation

In this paper we introduce an optimistic approach to handle out-of-order items. Our evaluation demonstrates the low overhead of our method. The contributions of this paper are the following:

- Definition of new optimistic technique to handle out-of-order items in stateful operations that requires single buffer per computational pipeline
- Analysis of properties of this approach
- Demonstration of working example that applies proposed method

The rest of the paper is structured as follows: in section II we formalize the preliminaries of stream processing, the examples of tasks that require ordered input are described in section III, the typical approaches for handling out-of-order events are discussed in IV, our optimistic technique is detailed in V and its performance is demonstrated in VI, the main differences between proposed method and existing ones are shown in VII, finally we discuss the results and our plans in VIII.

II. Stream processing concepts

In this section we define some preliminaries for distributed stream processing. It allows us to unify required terms and to introduce definitions, which are needed for the subsequent statements.

In this paper a stream processing system is considered as a shared-nothing distributed runtime. It handles input items and processes them one-by-one according to user-provided logic. It is able to handle a potentially unlimited number of items. The main requirement of this kind of data processing systems is to provide low latency between event occurrence and its processing under a fixed load. The term *distributed* implies that user's procedures can be partitioned into distinct computational units or shards. The following subsections detail the properties of such systems more precisely.

A. Data flow

The basic data flow abstraction is a *stream*. The stream is an infinite sequence of heterogeneous data items. Each data item contains a payload defined by a user. Besides, it can be accompanied by meta-information. Typically, meta-information is used to define an order on data items. For instance, it can be represented as a UNIX timestamp in milliseconds.

B. Computation flow

Commonly, the computational pipeline is defined in the form of *logical graph*. The vertices of the logical graph are operations, and the edges are links between them. Logical graph defines only relations between operations, but it does not describe the physical deployment. The logical graph for the pipeline shown in Figure 1 is presented in Figure 2

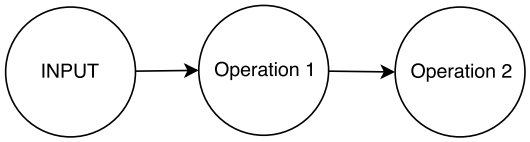

Fig. 2. The logical graph for the physical graph shown above

C. Operations

There are two main types of streaming operations: *stateless* and *stateful*. Stateless operations do not need any knowledge about past inputs to process current one correctly. A simple illustration is a map operation that multiplies by two any numeric input item's payload. On the other hand, stateful operations are able to keep some aggregations or summaries of received events. In such case, the output of the operation depends not only on the input but also on its current state. As an example, one can define an operation that sums all previous items with numerical payloads.

D. Physical deployment

As mentioned above, each operation can be partitioned between multiple computational units. Data items can be balanced between partitions by key extracted from an item's payload for stateful operations. For stateless operations items can be balanced randomly. The schema of physical partitioning of operations is sometimes called *physical graph*. Regarding physical links between operations, in the most cases, it is assumed that they guarantee FIFO order.

E. Guarantees

Recognized important property of stream processing systems is the type of guarantees it provides in case of failures. There are three main types of such guarantees. *At most once* semantics states that each input event is processed once or not processed at all. *At least once* guarantees that each input item is processed, but possibly multiple times, that can lead to result duplication. *Exactly once* semantics guarantee that each input event is processed exactly one time.

III. Tasks that require in-order input

In this section we outline common problems that require the order persistence of input items and describe a couple of computation scenarios, which can be found in many real-life projects.

A. Tasks requiring complete event retrieval

The processing of the single event could be split into multiple independent parts that are executed in parallel. After execution finishes, the results must be combined into a single cumulative item. This task could be naturally implemented using order guarantees: the final part of the task could be flagged and receiving the flagged result guarantees that the rest of the operation is completed. Unfortunately, as it was shown in Figure 1, independent processing via different paths can lead to reordering.

As an example, we can mention the computation of inverted index. Pipeline shown in Figure 1 can be applied for the task. In this case, operation 1 accepts documents from the input and for each word produces corresponding positions. Operation 2 receives pairs of word and positions and computes changelog of the inverted index for each word. In order to produce changes for each document in the form of single update, there is a need for retrieval all changelogs regarding the document.

B. Tasks that depend on the order of input events

This class includes all non-commutative operations. Such tasks strictly require the order of input items, because there are no any other methods to compute a valid result.

Generally, this class of examples includes all windowed aggregations. Moving average calculation over a numerical stream is a typical case. Even if values inside window could be arbitrary reordered, the order between windows is required to ensure that incomplete windows are not produced.

IV. Existing solutions

There are two most common methods that are used to implement order-sensitive operators: in-order processing (IOP) [5], [6], [7] and out-of-order processing (OOP) [8].

A. In-order processing

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. Figure 3 shows the union operation that combines multiple streams into the single one. Both input streams are ordered, as predecessors must meet ordering constraint. Nevertheless, if there is arrival time skew between input streams, the union must buffer the earlier stream to produce ordered output. It is known that IOP is memory demanding and has unpredictable latencies and limited scalability [8].

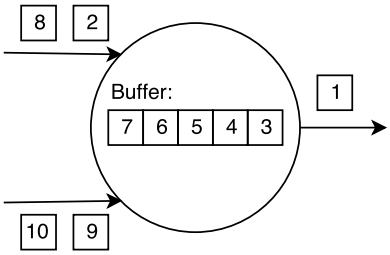

Fig. 3. IOP union operation. Due to delay of the upper stream operation must buffer elements

B. Out-of-order processing

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 [9], low watermarks [10], or heartbeats [11]. They go along 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 any elements with lesser meta-information. Therefore, indicators must be monotonic, but data items between two consecutive indicators can be arbitrarily reordered. Data sources periodically yield them.

A timed window operation can be mentioned as an example of OOP approach. A window operation buffers partial results until a progress indicator arrives. After that, the window flushes corresponding buffers and propagates the indicator to the next operation down the stream.

OOP addresses the downsides of IOP, but the direct implementation has flaws too. Even if the input stream is totally ordered, the operation must wait for the progress indicator. Figure 4 illustrates such case. Bottom window is complete but must be blocked until the indicator for the item 11 arrives. Another issue of OOP is that periodical flushes can result in load bursts and an increase in latency.

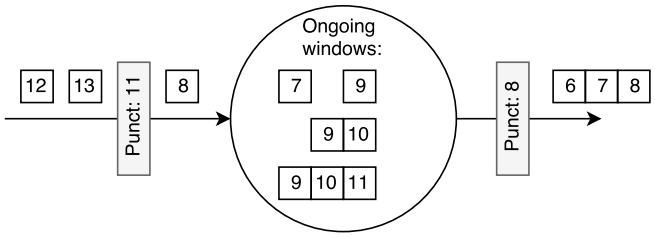

Fig. 4. OOP sliding window, range=3, slide=1. Operation must block lower window until next punctuation arrival

V. Optimistic approach

The main idea of our method is to represent stateful transformations as a sequence of a map and windowed grouping operations and handle out-of-order items within them.

Following our approach, we make some assumptions about stream processing system, that is suitable for applying it. Such system must support meta-information on data items, allow cycles in the logical graph, and its set of predefined operations must be sufficient to express map and windowed grouping operations. Additionally, OOP indicators should be supported.

The ordering model of data items is defined at the beginning of this section. Then, we show that any stateful transformation can be implemented using the combination of windowed grouping and map operations. After that, we demonstrate an optimistic approach to handle out-of-order items within these operations. At the end of the section, the limitations of such technique are discussed.

A. Ordering model

We assume that there is a total order on data items. If there are multiple data sources and there is no natural ordering between them, the *source id* can be used in addition to the locally assigned timestamp. The items are compared lexicographically: timestamps first, then *source id*. 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 are inserted in output stream sequentially. Moreover, the output follows corresponding input but precedes the next item. The ordering model is shown in Figure 5. $F(x)$ is an arbitrary transformation. Replication and union are used to inject original unmodified items into the resulting stream to show the order between items.

B. Semantics of map and windowed grouping operations

1) Map: Map transforms input item into a sequence of its derivatives, according to a user-provided function f. This sequence can consist of any number of items or even be empty.

2) Windowed grouping: Windowed grouping is a stateful operation with a numeric parameter *window*. It is supposed that payloads of input items of grouping operation have key-value form. The state of this operation is represented by a set of buckets, one for each key.

Windowed grouping has the following semantics:

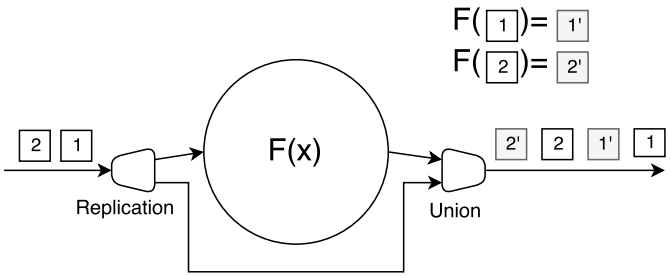

Fig. 5. Ordering model

- Each input item is appended to the corresponding bucket
- The output of grouping operation is a window-sized tuple of the last items in the corresponding bucket. If bucket size is less than window, the output contains full bucket

The pseudocode is presented in Algorithm 1. *Emit* function is called to send new items downstream.

Algorithm 1 Grouping semantics

```
1: function INSERT(item, bucket)
2:     APPEND(bucket, item)
3:     left ← max(0, bucket_length − window)
4:     right ← bucket_length
5:     EMIT(new DataItem(bucket[left, right]))
6: end function
```

The following example illustrates the semantics of the windowed grouping operation. In this example, payloads of input items are 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:

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

The special case of grouping with $window = 2$ in conjunction with a stateless map is used to implement arbitrary stateful transformation.

C. Stateful transformations using defined operations

Figure 6 shows the part of the logical pipeline, that can be used for stateful transformation. The input of windowed grouping operation is supposed to be ordered. There are several functional steps to produce output and update state. There are two cases of these steps:

- When the first item arrives at grouping, it is inserted into the empty bucket. The grouping outputs single-element tuple, and then it is sent to the combined map. Combined map generates state object and sends it back to the grouping in the form of an ordinal key-value data item. The key of the state item is the same as in the item in tuple and value is the state. Combined map can generate some additional output and send it further down the stream.
- When the state item arrives at grouping, it is inserted in the tail of the corresponding bucket after the item that triggers state updating. The ordering model guarantees that the state item would be processed before the next item. Grouping outputs tuple with this item and the state. However, combine map filters out such tuple, because its last element is the state. This fact implies that the state has been already combined with the first item in the tuple.
- When new regular input item arrives at windowed grouping, it is inserted into the corresponding bucket's tail, because of the order assumptions. Additionally, the right ordering guarantees that input item is grouped into the tuple with previously generated state item. The next map operation combines new item and previous state into the new state item. After that, the new state item is returned to the grouping through the cycle. As in the first case, combined map can generate some additional output.

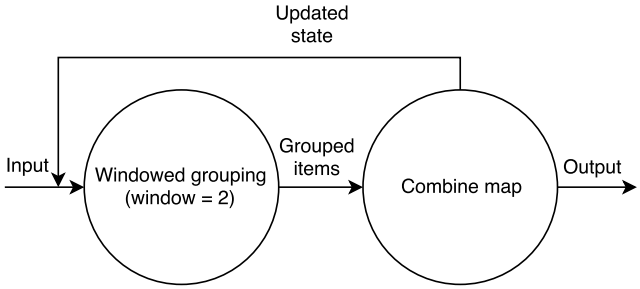

Fig. 6. The part of the logical pipeline for stateful transformation

The example of square matrix multiplication within proposed approach is shown in Figure 7. In this example, input items are represented as key-value pairs, where the key is the dimension of a matrix, and the value is the matrix itself. The reaction on three input matrices are the following:

- When the first matrix *A* arrives at grouping, it is put into the empty bucket for 3x3 matrices. After that, the single-element tuple with matrix *A* is sent to combine map operation. Combine map creates state object for matrix *A*, which is just *A* itself. In the last step, state item is sent back to grouping, and it is inserted right after item for matrix *A*
- Matrix *B* is arrived and inserted into the bucket right after state item. The tuple containing state item and item for matrix *B* is sent to combine map. Combine map multiplies matrix in the state by matrix *B*. The result of this operation is matrix *AB*. New state item for matrix *AB* is created and sent back to the grouping. It is inserted in bucket right after item with matrix *B*
- Matrix *C* is arrived and went through the pipeline in a similar way as matrix *B*

D. Handling out-of-order events

When we introduce the model for stateful operations, we assume that all items arrive at grouping in the right order.

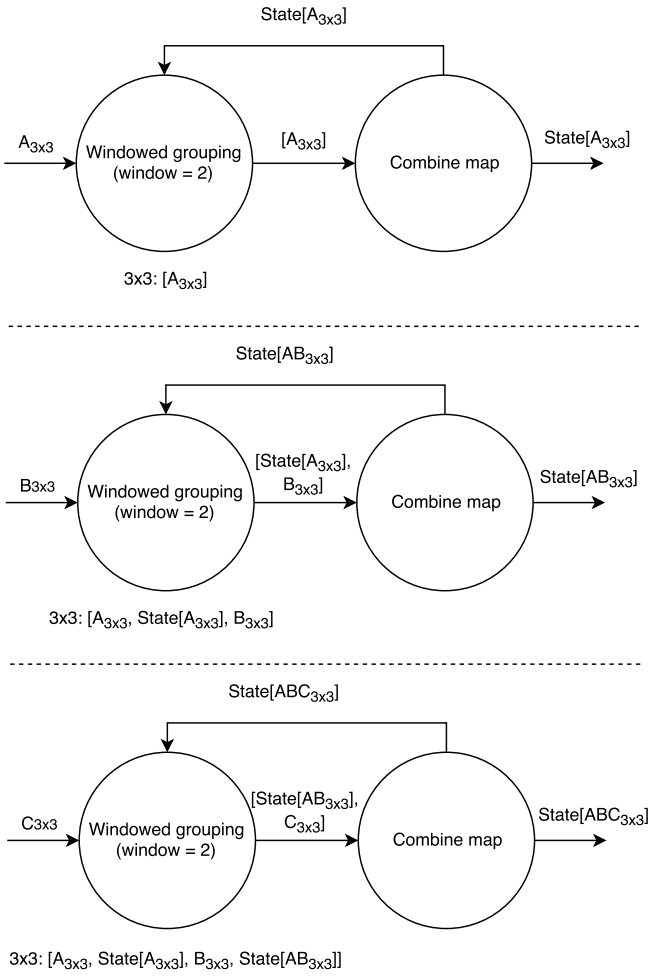

Fig. 7. Matrix multiplication example

However, as it was shown above, it is not possible in practice without additional buffering. We propose the following approach to handle events in grouping:

- If an item arrives in order, it is processed as usual
- If two items are out-of-order, and the grouping observes the second one then it is inserted into the corresponding bucket at the position defined by the meta-information. After that, tuples, which contain new item are generated and sent further down the stream. 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 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 through the same path in the computational pipeline.

The pseudocode of the grouping is shown in Algorithm 2. The functions accept new element, depending on its type. They also receive a bucket for element's hash. *Emit* function is called to send new items downstream.

This technique guarantees that all correct tuples are eventually produced. However, invalid ones are also generated.

Algorithm 2 Implementation of grouping semantics

```
1:  function InsertTombstone(item, bucket)
2:      cleanPosition ← lowerBound(item, bucket)
3:      for all group : group ∩ cleanPosition ≠ ∅ do
4:          ▷ Send tombstones for invalid groups
5:          Emit(new DataItem_tomb(group))
6:      end for
7:      Remove(bucket, cleanPosition)
8:      for all group : group ∩ cleanPosition ≠ ∅ do
9:          ▷ Emit new groups that appeared after collapse
10:         Emit(new DataItem(group))
11:     end for
12: end function
13:
14: function InsertRegular(item, bucket)
15:     insertPosition ← lowerBound(item, bucket)
16:     for all group : group ∩ insertPosition ≠ ∅ do
17:         ▷ Send tombstones for groups that would disappear
18:         ▷ after insert
19:         Emit(new DataItem_tomb(group))
20:     end for
21:     Insert(bucket, insertPosition)
22:     for all group : group ∩ insertPosition ≠ ∅ do
23:         ▷ Emit new groups
24:         Emit(new DataItem(group))
25:     end for
26: end function
```

Therefore, there is a need for *barrier* at the pipeline's sink, that filters invalid items when corresponding tombstones arrive. The barrier 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. This guarantee can be provided by punctuations or low watermarks, as it is implemented in the most stream processing systems. The fundamental idea behind this approach is to shift blocking as far as possible down the stream. Notably, this is the only buffer in the whole system, unlike existing solutions.

The pseudocode for the barrier is shown in Algorithm 3. The function $Insert$ is called by the system on the new item's arrival. $Punctuation$ is called when there is a guarantee that there are no tombstones up the stream with the specified time. The OOP architecture [8] can be employed to provide such guarantee.

E. 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.

Algorithm 3 Barrier

```
1: buffer ← ∅
2:
3: function INSERT(item)
4:     position ← lowerBound(item, buffer)
5:     if isTombstone(item) then
6:         REMOVE(buffer, position)
7:     else
8:         INSERT(buffer, position)
9:     end if
10: end function
11:
12: function PUNCTUATION(time)
13:     for all item : item ∈ buffer & item_ts < time do
14:         EMIT(item)
15:         REMOVE(item, buffer)
16:     end for
17: end function
```

Regarding the weaknesses, this method can generate additional items, which lead to extra network traffic and computations. Experiments, which are shown in the section VI demonstrate that the number of extra items is low.

VI. Experiments

A. Setup

We performed the series of experiments to estimate the performance of our system 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
3) The load distribution is skewed, because of Zipf's law

These properties make the task sufficient to comprehensively analyze the performance of the proposed solution. 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.

The logical pipeline of this computation is shown in Figure 8. First map operation accepts Wikipedia documents and outputs pairs of words and corresponding positions. The next part of the pipeline accepts pairs of word and positions and computes updated posting list and the actual changelog.

This stateful transformation is implemented in the form of groping and map operation with a cycle, as it was shown in the previous section. Regarding the physical deployment, the full logical graph is deployed on each computational unit or worker. Documents are randomly shuffled before the first map operation. Word positions are partitioned by word before grouping. The other links are implemented as simple chain calls.

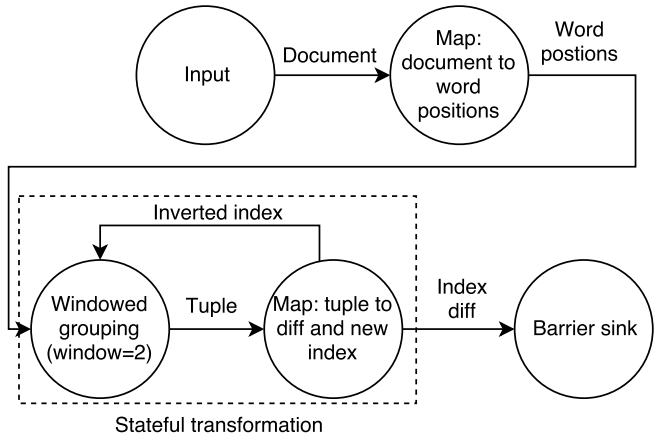

Fig. 8. Logical pipeline for inverted index

Our experiments were performed on clusters of 10 nodes. Each node is an AWS EC2 micro instance with 1GB RAM and 1 core CPU.

B. Overhead and scalability

As a key metric in our experiment, we take the ratio of arrived at the barrier items count to the number of the valid items among them. This value clearly represents the overhead of our approach, as it was mentioned at the end of the previous section.

The relation between the number of workers, the delay between input documents and the proposed ratio is shown in Figure 9. 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 non-extreme load, 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.

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

C. Comparison against Apache Flink

We compared the performance of our optimistic approach with state-of-the-art stream processing system Apache Flink [1]. 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 watermark is received. Watermarks are sent after each document. The network buffer timeout is set to 0 to minimize latency.

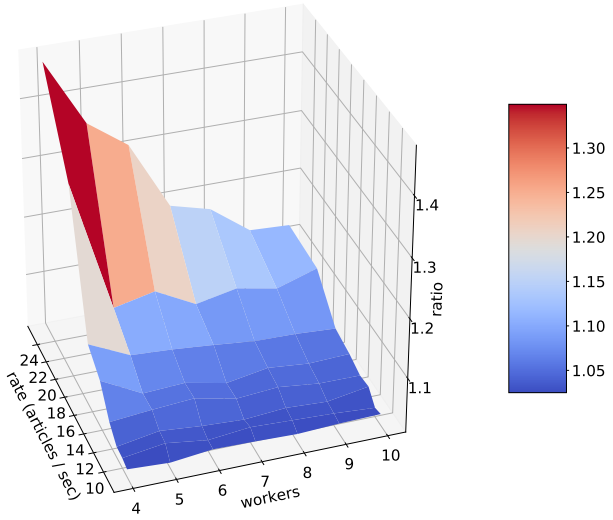

Fig. 9. The relation between the number of workers, the delay between input documents and the replay ratio

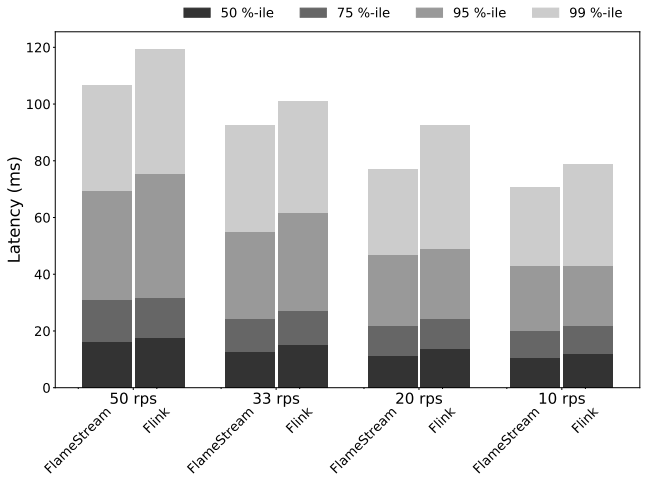

Fig. 10. The comparison in latencies between FlameStream and Flink within 10 nodes and distinct document rates

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.

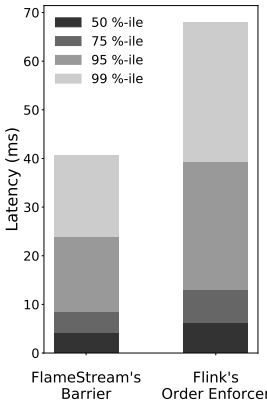

Fig. 11. Comparison between waiting time in Flink's buffer and FlameStream's barrier

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 9. Nevertheless, it should be noted that FlameStream demonstrates better latency under non-extreme load.

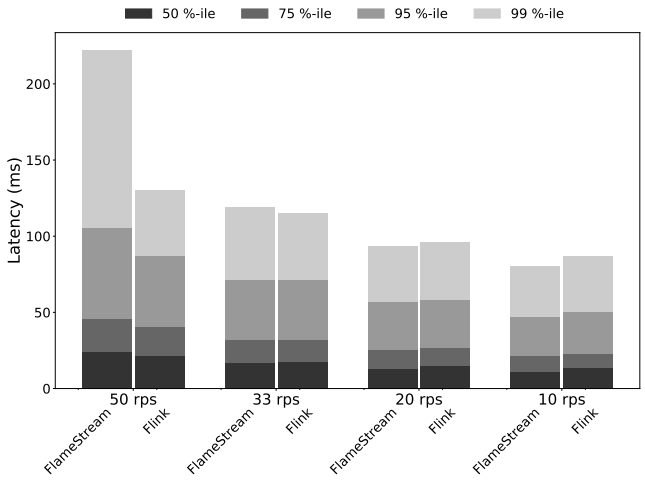

Fig. 12. The comparison in latencies between FlameStream and Flink within 5 nodes and distinct document rates

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.

VII. Related work

Research works on this topic analyze different methods of handling out-of-order items. Most of them are based on buffering.

K-slack technique can be applied, if network delay is predictable [12], [13]. 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, that are mentioned in the section IV, are popular within research works and industrial applications. IOP architecture is applied in [6], [14], [5], [15], [16], [17]. OOP approach is introduced in [8] and it is widely used in the industrial stream processing systems, for instance, Flink [1] and Millwheel [10].

Optimistic techniques are less covered in literature. In [18] so-called *aggressive* approach is proposed. This approach uses the idea that operation can immediately output *insertion* message on the first input. After that, if that message became invalid, because of the arrival of out-of-order items, an operation can send *deletion* message to remove the previous result and then send new insertion item. The idea of deletion messages 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 arbitrary operations.

Yet another optimistic strategy is detailed in [19]. This method is probabilistic: it guarantees the right order with some probability. Besides, it supports only the limited number of query operators.

VIII. Conslusion

In this paper we introduce an optimistic approach for handling out-of-order events. Our technique has the following key properties:

- It does not require buffering before each order-sensitive operation
- The method handles properly any stateful operation
- The overhead of the proposed approach is low (under 10% in most of our experiments)
- The total overhead could be managed by optimization of the computational layout

The optimistic nature of this method is able to help to reduce the cost of waiting for punctuations or watermarks. It is implied by the fact, that at the moment when the watermark arrives all computations are already done.

The experiments show that the number of the extra items does not increase with the growth of the number of the computational units. Therefore, this approach can potentially provide lower latency in stream processing systems.

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[2] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017.

[3] (2017, Oct.) Apache storm. [Online]. Available: http://storm.apache.org/

[4] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proc. of the 2015 ACM SIGMOD Intnl. Conf. on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250.

[5] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00778-004-0147-z

[6] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 647–651. [Online]. Available: http://doi.acm.org/10.1145/872757.872838

[7] M. Hammad, W. Aref, and A. Elmagarmid, "Optimizing in-order execution of continuous queries over streamed sensor data," 2004.

[8] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: A new architecture for high-performance stream systems," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 274–288, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453890

[9] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, no. 3, pp. 555–568, Mar. 2003. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2003.1198390

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

[11] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proc. PODS*, ser. PODS '04. New York, NY, USA: ACM, 2004, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/1055558.1055596

[12] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 545–580, Sep. 2004. [Online]. Available: http://doi.acm.org/10.1145/1016028.1016032

[13] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani, "Event stream processing with out-of-order data arrival," in *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, ser. ICDCSW '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 67–. [Online]. Available: http://dx.doi.org/10.1109/ICDCSW.2007.35

[14] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, Aug. 2003. [Online]. Available: http://dx.doi.org/10.1007/s00778-003-0095-z

[15] L. Ding and E. A. Rundensteiner, "Evaluating window joins over punctuated streams," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, ser. CIKM '04. New York, NY, USA: ACM, 2004, pp. 98–107. [Online]. Available: http://doi.acm.org/10.1145/1031171.1031189

[16] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, "Scheduling for shared window joins over data streams," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 297–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315478

[17] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid, "Optimizing in-order execution of continuous queries over streamed sensor data," in *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, ser. SSDBM'2005. Berkeley, CA, US: Lawrence Berkeley Laboratory, 2005, pp. 143–146. [Online]. Available: http://dl.acm.org/citation.cfm?id=1116877.1116897

[18] M. Wei, M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. Claypool, "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*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1031–1034.

[19] C.-W. Li, Y. Gu, G. Yu, and B. Hong, "Aggressive complex event processing with confidence over out-of-order streams," *Journal of Computer Science and Technology*, vol. 26, no. 4, pp. 685–696, Jul 2011. [Online]. Available: https://doi.org/10.1007/s11390-011-1168-x