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*—Abstract**

I. Introduction

Nowadays, a lot of real-life applications use stream processing for network monitoring, financial analysis, 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 a wide range of analytical problems. However, issues related to the order-sensitive computations still remain. Most of these systems assume that events arrive with monotonically increasing timestamps. Often, such timestamps can be assigned at system's entry. Nevertheless, even if input items arrive at system's entry monotonically, they can be reordered because of parallel and asynchronous processing. In this case, order-sensitive operations located in data flow pipeline quite deeply can be broken. Figure 1 shows the example of common distributed stream processing pipeline that breaks the input order of operation, even if input events are monotonic and links between operations guarantee FIFO order. Basically, ordering constraints make sense only for stateful operations.

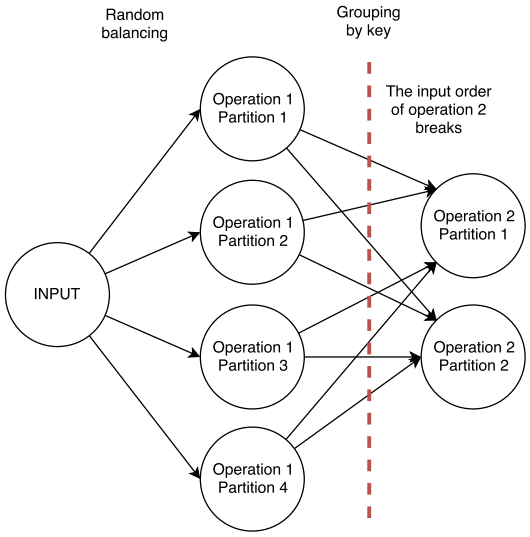

Fig. 1. An example of common distributed stream processing pipeline that breaks the input order of operation

The typical way to achieve in-order processing is to buffer operation's input for some time to ensure that there are no out-of-order items. Most of the modern stream processing systems provide functionality to buffer all input items before specified operations until some user-provided conditions are satisfied. Conditions can be set on timing or the count of elements in buffer. The main disadvantage of such techniques is that it can lead to significantly large latency, especially if the processing pipeline contains several operations that require ordered input.

An alternative option is to handle out-of-order items as a special case in business logic of operation. However, this way is suitable only for limited number of tasks. Moreover, it may dramatically complicate the business-logic, which can lead to increasing the cost of its maintenance and is error-prone.

In this paper we propose an optimistic approach to handle out-of order events in any stateful operation. In addition, we demonstrate its advantages compared to existing solutions. The contributions of this paper are the following:

- Definition of new optimistic technique to handle out-of-order items in stateful operations
- Demonstration of properties of this approach, particularly, reducing waiting time for multiple operations with order constraints without affecting the complexity of business logic
- 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 stream processing. It allows us to unify required terms and to introduce definitions, which are needed for the further statements.

Commonly, stream processing system is a shared-nothing distributed runtime that handles potentially unbounded number of input events and process them one-by-one, according to the procedures provided by user. The main purpose of this kind of data processing systems is to provide low latency

between event occurrence and its processing. Term *distributed* implies that user's procedures can have partitions on 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*. Stream is an infinite sequence of events or data items. Each data item contains payload defined by user. Besides, it can be assigned by meta-information. Usually, meta-information is used to define an order on data items. For instance, it can be represented as a UNIX timestamp in milliseconds with a trace of applied operations.

B. Computation flow

Commonly, computational pipeline is defined in the form of *logical graph*. The vertices of logical graph are operations, and edges are links between them. Logical graph defines only relations between operations, but it does not detail its physical deployment.

C. Operations

There are two main types of streaming operations: *stateless* and *stateful*. Stateless operations do not need any knowledge about past inputs to correctly process current. A simple illustration is a map operation that multiplies by 2 any 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 operation depends not only on input, but on its current state. As an example, an operation that reacts on each input item's payload with the sum of it and all previous payloads can be mentioned.

D. Physical deployment

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

E. Guarantees

Guarantees of stream processing declares the properties of data safety in case of system failure. 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 mention common problems that require the order of input items. Additionally, we note a couple of computation scenarios, which can be found in many real-life projects.

A. Tasks requiring complete event retrieval

Quite often we suppose that business-logic event is an atomic object, i.e. single data item. However, there are cases when single input event is split into multiple items within a stream. These input item derivatives can be processed independently, despite the fact that they all have the same meta-information. As it was shown in the figure 1, independent processing can lead to reordering. Therefore, operations which require complete event data to process valid output cannot simply detect the completeness by the order of input items.

There are other solutions for this kind of problems rather than ordering of input items. They are shown in the section IV.

B. Tasks that depend on the order of input events

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

C. Examples

1) ???:

2) Inverted index: Pipeline shown in the figure 1 can be used for computing inverted index. In this case, operation 1 accepts documents from input and for each word produces corresponding positions. Operation 2 accepts pairs of word and positions and computes change log of inverted index for each word. Because of the fact that output change logs must be ordered in order to get valid index after applying them, operation 2 requires ordered input.

IV. Existing solutions

There are two 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. IOP

In IOP approach each operation enforces total order on elements. For example union combines multiple streams into one 2. Each input stream of union is ordered, as predecessor must meet ordering constraint. If there is arrival time skew between input stream merge must buffer the earlier stream to produce ordered output. Operators such as projection and selection applies function and propagete items down the stream without any additional buffering.

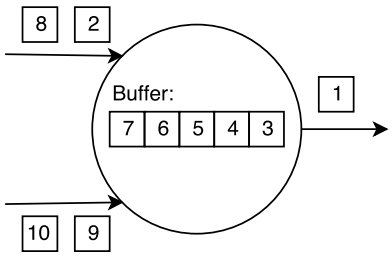

Fig. 2. IOP union operation. Due to delay of the first stream operation must buffer elements

IOP is know to be memory demanding, to have unpredictible latencies and limitid scalability.[8]

B. OOP

OOP is an architecture of streaming streaming that doesn't require order maintaince. To unblock blocking operations OOP systems use progress indicators such as such as punchuations [9], low watermarks [10] or hearbeats [11]. Punctuations are periodically yielded by data sources and carries timestamp that promises that there won't be any elements older that heartbeats value. Punctuations are monotonic and data items between two consecutive heartbeats can be arbitrarily reordered, pic 123. Operations propagates them with respect to their semantics.

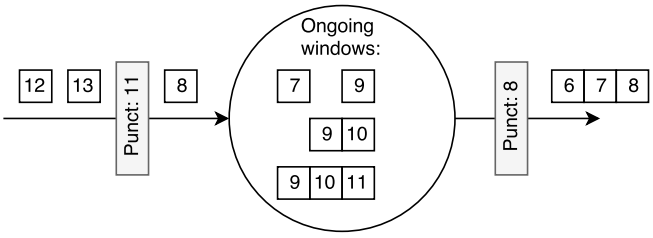

Fig. 3. OOP sliding window, range=3, slide=1. Operation must block last window until next punctuation arrival

For example, windows collects partial aggregates until punctuation that guarantees that there no element that belogs to the ongoing window can be delivered to the window's input. On punctuation's arrival it flushes completed windows and propagates punctuation to the next operation down the stream 3

V. Optimistic approach

The main idea of our method is to represent stateful transformations as a sequence of 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 logical graph, and its set of predefined operations must be sufficient to express map and windowed grouping operations.

In this section, firstly, we define the ordering model of data items. 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. In the end of the section, the limitations of such technique are discussed.

A. Ordering model

We assume that there is a total order on meta-information of data items. Besides, ordering is preserved, when item is going through the operations. More precisely, the order of output items is the same as the order of corresponding input items. Moreover, the output follows corresponding input, but precedes the next item. The ordering model is shown in the figure 4.

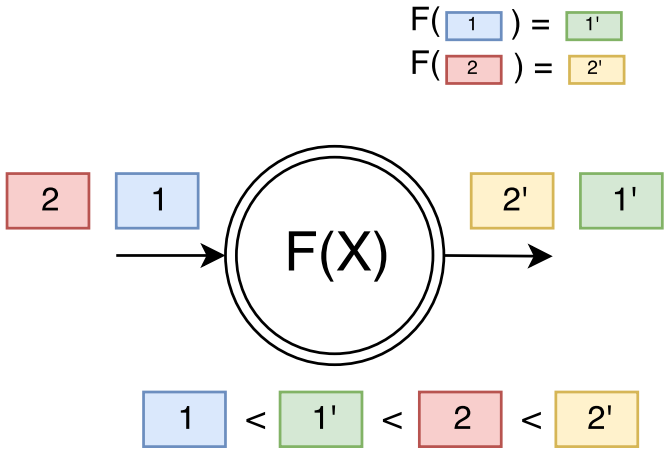

Fig. 4. Ordering

B. Semantics of map and windowed grouping operations

1) Map: Map transforms input item into a sequence of its derivatives, according to a function on item's payload provided by user. 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:

- Each input item is put into corresponding bucket at position defined by its meta-information
- 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 contain full bucket

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:

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

C. Stateful transformations using defined operations

Figure 5 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. These steps can be detailed by considering two cases:

- When the first item arrives at grouping, it is inserted into the empty bucket. Grouping outputs single-element tuple, and then it is sent to combined map. Combined map generates state object and sends it back to the grouping in the form of 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 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 preceding case, combined map can generate some additional output

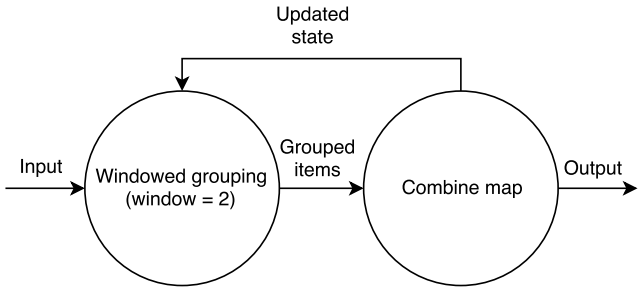

Fig. 5. The part of the logical pipeline for stateful transformation

The example of square matrix multiplication within proposed approach is shown in the figure 6. In this example, input items are represented as key-value pairs, where 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, single-element tuple with matrix *A* is sent to combine map operation. Combine map creates state object for matrix *A*, which is actually 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. 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 the similar way as matrix *B*

D. Handling out-of-order events

E. Limitations

VI. Experiments

Show that our idea works. Problem: how to perform experiments without introducing our MapReduce model?

VII. Related work

Seems to be related: [12], [13], [14].

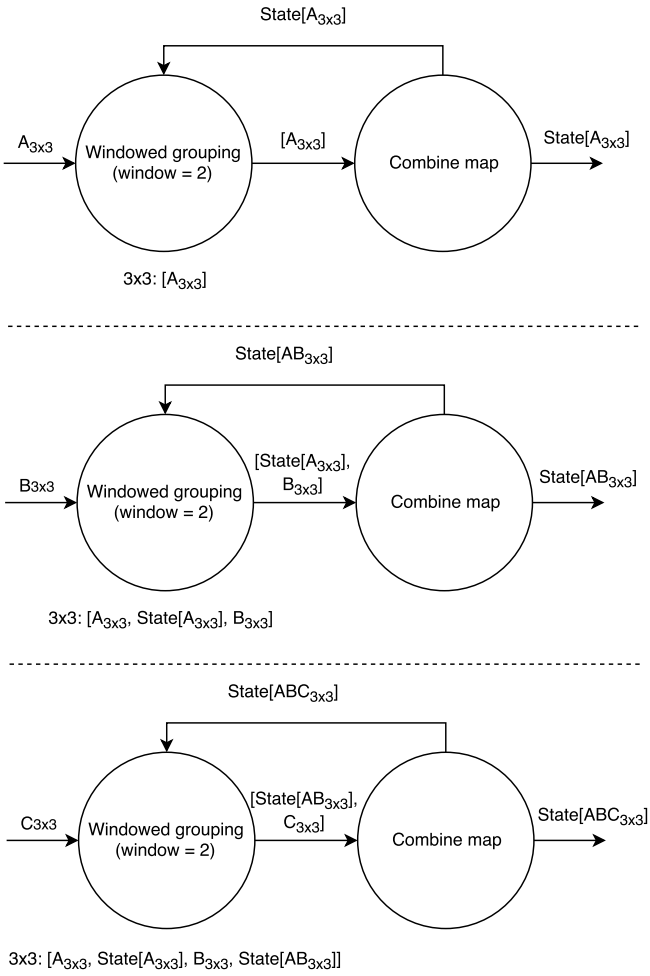

Fig. 6. Matrix multiplication example

VIII. Conslusion

We are winners :)

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. [Online]. Available: https://doi.org/10.14778/3137765.3137770

[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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788

[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 Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536229

[11] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '04. New York, NY, USA: ACM, 2004, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/1055558.1055596

[12] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani, "Event stream processing with out-of-order data arrival," in *Distributed Computing Systems Workshops, 2007. ICDCSW '07. 27th International Conference on*, June 2007, pp. 67–67.

[13] 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 *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1031–1034. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559973

[14] C. Mutschler and M. Philippsen, "Adaptive speculative processing of out-of-order event streams," *ACM Trans. Internet Technol.*, vol. 14, no. 1, pp. 4:1–4:24, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2633686