Substream Management in Distributed Streaming Dataflows

Artem Trofimov
tyoma@lzy.ai
lzy.ai
Tel Aviv, Israel

Nikita Sokolov
faucct@gmail.com
Yandex Cloud
Saint Petersburg, Russia

Nikita Marshalkin
marnikitta@gmail.com
No affiliation
Limassol, Cyprus

Igor Kuralenok
igor.kuralenok@huawei.com
Huawei
Saint Petersburg, Russia

Boris Novikov
borisnov@acm.org
HSE University
Saint Petersburg, Russia

ABSTRACT

Most state-of-the-art SPEs use punctuations to divide a stream into bounded substreams of messages, such as epochs and windows. The punctuation approach is powerful but has limitations: it does not support cyclic dataflows, is poorly scalable in some cases due to intensive use of broadcasts, and becomes inefficient when the number of chunks or cluster size becomes significant. We introduce a new substream tracking technique called trAcker that overcomes the limits of punctuations. We experimentally evaluate the properties of trAcker in both synthetic and real-world environments. Experiments show that our technique outperforms punctuations for a large number of substreams and efficiently handles real-world cyclic dataflows.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

Data streams, punctuations, watermarks, substreams, stream join, state management

ACM Reference Format:
Artem Trofimov, Nikita Sokolov, Nikita Marshalkin, Igor Kuralenok, and Boris Novikov. 2022. Substream Management in Distributed Streaming Dataflows. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524860.3539809

1 INTRODUCTION

The processing of a data stream without insights into the properties of its data elements can be challenging. For example, it may be unclear when a system can prune outdated keyed state [27], release windowed aggregations [5], or create a state snapshot for an epoch [7].

Each of these scenarios is a particular case of a problem of monitoring substreams emergence and termination that we call a *substream management problem*. A substream is a part of the stream such that all its elements satisfy some predicate. For example, in the case of state pruning, the predicate is *[a data element key equals to K]*, for time window aggregations, the predicate is *[a data element has a timestamp less than T]*, and for state snapshotting it is *[a data element belongs to the epoch E]*.

In this paper, we focus only on two signals: substream start and its termination. Tracking a start of a substream is a straightforward task: the first event of a substream will naturally trigger its start. On the contrary, generating a substream termination event is a challenging task, and various properties may be required by practical problems:

- Deterministic windowed join[1] requires an order of termination signals to respect the order of input elements (termination events from data producers) [12, 20].
- An epoch is a substream that an SPE should process atomically. A termination event for an epoch should arrive before any elements of the next epoch [8].
- State pruning problem does not require any specific properties from termination events. However, late termination event receiving may cause sub-optimal memory utilization.

A popular substream management method is the punctuations framework [28]. The main idea behind this framework is to divide the stream by injecting special elements called *punctuations* that define substreams "borders". These special elements are propagated via the same network channels as data elements. While the punctuation approach is robust and easy to implement, it has several limitations.

Punctuations are not applicable for cyclic dataflows in a general case because elements belonging to a substream can remain in transit within a cycle for an uncertain time [6]. The technique proposed in [7] mitigates this issue for the state snapshotting problem. The main idea of this technique is to include in a snapshot all in-transit elements (possibly from previous epochs) within a cycle and then resend them on rollback. However, it provides a solution for a specific problem that does not allow a system to determine a substream termination for cyclic dataflows using punctuations.

The high network overhead forms another limitation. Network traffic complexity for this method is $O(K|\Pi|^2)$, where $|\Pi|$ is the number of processes and K is the number of substreams because

[1] given the same sequences of input tuples, the same output tuples will be produced

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539809

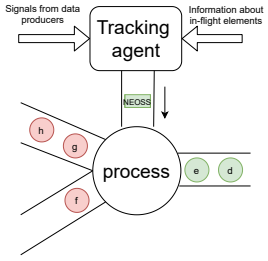


Figure 1: trAcker framework: tracking agent aggregates information about substreams and produces NEOSS

each process should propagate punctuations to all output channels. This complexity boundary covers the worst case when all processes are interconnected. However, SPEs prefer to distribute the work among nodes evenly to ensure scalability [1, 9, 15]. This load balancing implies that each process effectively occupies channels to all other processes. The worst-case complexity boundary is tight for scenarios when an execution graph contains at least one operator that repartitions data.

Substreams can be *fine-grained*: for example, each user session defines a substream. If there are a lot of small substreams, an inefficient substream management system can degrade the latency [4] and the throughput of an SPE [18] or affect the performance of state checkpointing [33].

In this work we formalize the substream management problem and show that the network traffic overhead of the punctuations framework is far from the optimal. We also formally define properties of a substream management technique required by various problems such as state snapshotting to ensure that a newly proposed method satisfy them.

We introduce a new substream management framework called trAcker. Figure 1 shows the high-level scheme of our method. Within this framework, we use a dedicated agent that receives information about substreams from the entire SPE and sends back *end-of-substream notifications* (NEOSS). NEOSS messages are propagated through this agent without broadcasting between processes, reducing the amount of extra traffic. Such propagation method is suitable for cyclic dataflows because there is no need to forward service traffic through the cycles.

Basic comparison between the trAcker framework and its alternatives is shown in Table 1. Regarding network traffic, $|\Pi|$ is the number of computational nodes and K is the number of substreams. We can outline that the punctuations framework is the only substream management mechanism that supports arbitrary predicates for substreams, so we use it as a baseline approach in the experiments. The commonalities and differences between the trAcker framework and alternative solutions are detailed in Section 6.

In summary, our contributions are as follows:

(1) We provide a formal model of substream management. This model allows us to compare the properties of various substream management systems.
(2) We present a novel substream management technique that achieves a lower bound of network traffic overhead.

Table 1: An overview of substream management techniques

Method	Arbitrary predicates	Cycles	Traffic
Punctuations	+	-	$O(K\|\Pi\|^2)$
MillWheel*	-	N/A	N/A
Naiad*	-	+	$O(K\|\Pi\|^2)$
Acker	-	+	$O(K\|\Pi\|)$
trAcker	+	+	$O(K\|\Pi\|)$

*progress tracker

(3) We demonstrate trAcker performance in comparison to a state-of-the-art approach on diverse workloads.

The rest of the paper is organized as follows: Section 2 formalizes the substream management problem and indicates its main properties. In Section 3, we introduce a general design of the trAcker framework and demonstrate the properties of this substream management solution. Section 4 summarizes the implementation of trAckerİn Section 5, we show that the proposed technique is scalable and can outperform alternatives employed in state-of-the-art stream processing engines. The relevant prior research is outlined in Section 6. Finally, we discuss our conclusions in Section 7.

2 SUBSTREAM MANAGEMENT

First, in this section, we formalize a stream processing engine based on Chandy-Lamport definition of a distributed system. Then we define the substream management problem based on the notions from the proposed model. Finally, we discuss a state-of-the-art substream management technique called punctuations.

2.1 Processing model

Typically, distributed stream processing engines are shared-nothing runtimes that continuously ingest input elements, transform them according to a logical dataflow graph, and deliver output elements. The logical dataflow graph consists of user-defined operators. Operators are functions of a single input data element that produce a number of output data elements. Operators can be stateless or stateful: output elements may depend on the current state. A logical graph is mapped to a physical, distributed graph on deployment.

Table 2: Notations used throughout the paper

p	Process (node in a physical execution graph)
I_p, O_p	input and output channels of a process p
$func_p(U, M)$	User-defined operator run by process p. It receives current operator state U and an incoming message M
Π	The set of all processes
K	Number of substreams
c	A network channel between processes
$\mathcal{E}$	The set of all network channels
$s_p = U_p \cup B_p$	State of the process p consists of a mailbox B_p and a state U_p of $func_p$
mbc_p	Mailbox controller of a process p
e_p	Event of a process p
$Pred(e)$	Propositional formula defined on events
$pred(M)$	Propositional formula defined on messages
$t(M)$	Coarse time label

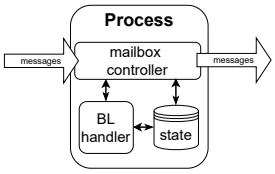


Figure 2: Structure of the SPE process

Commonly, a single logical operator can be deployed on multiple computational nodes. Further, we denote physical instances of logical operators as *processes*.

A deployed physical graph is a distributed system and could be described in terms of the Chandy-Lamport model [6, 11]. In this model, the authors introduce *events* that allow observing a state of the entire system. This approach allows defining system-wide guarantees: in the original paper, it is used to introduce the notion of *consistent state*, we use this approach for the definition of a substream management problem.

Following the notation from [6, 11], the distributed system is observed with events. Each event is a tuple of 5 elements $e = (p, s, s', c, M)$, where p is one of the deployed processes, s and s' are state of the process before and after processing, c is one of network FIFO channels that connect processes, and M is a message generated during processing. The generated event M comes to a channel state C until the destination process receives it. Processes and channels form a physical graph of the system $G = \{\Pi, \mathcal{E}\}$. We denote all input channels as I_p and output channels as O_p.

In a stream processing engine, we need to specify a process p to reflect the nature of SPE. In our model we split a process into two separate blocks: *business logic handler* (BLH), and *mailbox controller* (MBC). The first block encapsulates a user-defined operator. In this model user-defined operator does not directly communicate with other processes in the system. Instead of this, it receives and generates *messages* – data elements that are tagged by their source and destination. Further delivery of these messages along the communication channels is then handled by a mailbox controller that preserves the order of message generation. Figure 2 illustrates the scheme of a process. This system layout is not new, and it is widely used in practice (Akka, YDB, Millwheel, etc.).

More formally, when a process receives a message, it is handled by the mailbox controller that puts this message into a special segment of the process state (*mailbox* B_p). The business logic handler gets a message provided by MBC and triggers a user-defined operator. The user-defined operator processes the data element that the message contains and generates an arbitrary number of outgoing messages. BLH puts generated messages back to a mailbox. MBC sends outgoing messages along communication channels to destination processes. All mailbox controller operations respect the order of messages in the mailbox. If a user-defined operator has a state U_p, the joined process state will consist of the mailbox and this state $s = U_p \cup B_p$. In the Chandy-Lamport paradigm this algorithm produces the following events within a process:

- Communication events: $\langle recv, p, M\rangle$, $\langle send, p, M\rangle$ – these events are handled by mailbox controller
- Processing of an incoming message $\langle proc, p, M, M'\rangle$

Lets translate these events into 5-tuple language. Communication events move a message between communication channel and mailbox section of the state:

$$\langle recv, p, M\rangle = \left(p, s_p, s'_p = U_p \cup \left(B_p \cup \{M\}\right), c_{qp}, M\right) \quad (1)$$

$$\langle send, p, M\rangle = \left(p, s_p, s'_p = U_p \cup \left(B_p \setminus \{M\}\right), c_{p,dst(M)}, M\right) \quad (2)$$

Note that we need to be able to get a destination process directly from the message $dst(M)$. This function translates a destination element from logical dataflow graph nodes, used in user-defined code, to physical communication channels between processes. A practical case of this abstraction is a sharding scheme for some key: user-defined procedure emits event for some key, and a system is responsible for finding a proper physical channel to deliver this message.

Incoming message processing does not influence the communication channels and only ingest results of a message processing $(U', M') = func_p(U, M)$:

$$\langle proc, p, M, M'\rangle = \left(p, s_p, s'_p = U'_p \cup \left(B_p \setminus \{M\} \cup M'\right), \emptyset, \emptyset\right) \quad (3)$$

Note that in this case, M' may contain more than one message. Following the Chandy-Lamport model, we assume processes are single-threaded, so within the specific process p, all events are ordered by a local causal order relation $<_p$: $e_p^0, e_p^1, \ldots, e_p^i, \ldots$. Please note that each process has its own local causal order relation, so we do not assume any total order among events from different processes. This model is indeed practical, e.g., implemented in actor-based systems.

2.2 Substream management events

2.2.1 Substreams lifespan. For each process, we want to get the first and the last element of a substream. The first one could be found naturally when it emerges, but verification that there will be no more events of a substream could be problematic. Strict substream termination guarantee consists of two parts: source must promise that no more messages from substream may emerge, and the system must ensure it contains no substream messages. The first task requires a contract with a particular data source and is thus out of scope for this paper, though it is discussed in relevant literature [3]. Instead, we focus on the second task; this is challenging due to distributed nature of the system and the lack of a common message lifetime limit. This difficulty increases with the introduction of cycles into dataflow. Crucially, processes are not isolated from one another, and substream messages can move from one process to another. That is why we need to observe all in-flight messages in the system.

Formally, a substream can be defined via the propositional formula $Pred(e)$ for any system event. We have to use system events as they are ordered inside each process and can define a border of a substream. Sometimes it is more practical to induce this predicate to messages ($pred(M)$) involved in processing: $Pred(e) = (e = \langle proc, p, M, M'\rangle) \wedge pred(M)$.

In this paper we are interested in such $Pred(e)$, that has limited lifespan within a process and want to know when substream starts and terminates: $\forall p, \exists t_0^p, t_1^p : \exists e : e_{t_0^p} <_p e <_p e_{t_1^p}, Pred(e)\ \&\ \forall e' : e_{t_1^p} <_p e', \neg Pred(e')$. Lets boil this formula down: for each process p in the system there must be two event indices t_0^p for substream

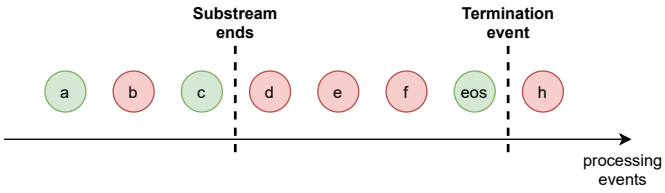


Figure 3: Substream management: soft bound

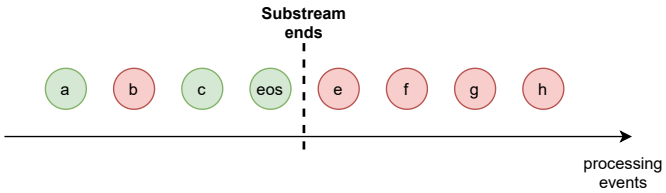


Figure 4: Substream management: firm bound

start and t_1^p for its termination, such that events satisfying $Pred$ must be between these indices.

Substream management problem is to define a special mechanism that estimates substream bound for each process. In our system model, we need to define a system event that indicates the bound of a substream for a process. We call this termination event or end-of-substream event, this appropriate :

$$\langle eoss, p, Pred\rangle = (p, B_p, B_p \cup eoss(Pred), \emptyset, \emptyset) \tag{4}$$

As we mentioned before, some problems require certain properties of the termination events. For example, the state pruning problem does not require any special properties, while for the state snapshotting problem, the substream management system should detect the exact substream bound. In the following sections, we formalize these properties.

2.2.2 Soft bound. Many applications that apply substream management systems do not require any special properties of termination events. In this case, we denote the guarantee provided by such events as *soft bound*, because termination events indicate only the fact that the substream ended some time ago, and other input elements could be processed after that. More formally, we define the soft bound guarantee of the termination event (end-of-substream) $\langle eoss_{soft}, p, Pred\rangle$ as follows:

$$\forall e, e >_p \langle eoss_{soft}, p, Pred\rangle \Rightarrow \neg Pred(e) \tag{5}$$

Figure 3 illustrates this notion. Terms $a, b, c, d...$ denote ordered processing events of a process p. The substream ends after event c. Note that there are several other events between the end-of-substream and c. This is the property of a *soft bound* guarantee: if $\langle eoss_{soft}, p, Pred\rangle$ occurs, all subsequent elements do not satisfy the predicate, but it is not necessarily the exact substream "border".

2.2.3 Firm bound. The guarantee that any new event will not satisfy the predicate is sufficient for many real-life problems, e.g., SPE can initiate process state pruning on such events. However, some problems require a *firm bound*: guarantee that the substream ends *exactly* after the termination event.

For example, epoch-based snapshotting protocol [8, 13] relies on the notion of *epoch*. An epoch is a special substream that must be processed atomically. Therefore, the SPE requires the termination event for a given epoch to occur immediately after the last processing event for that epoch. Otherwise, the snapshot can be inconsistent, capturing elements from multiple epochs. To support such scenarios, the end-of-substream event $\langle eoss_{firm}, p, Pred\rangle$ should satisfy the following condition:

$$\langle eoss_{firm}, p, Pred\rangle = \inf_{<_p}\langle eoss_{soft}, p, Pred\rangle \tag{6}$$

Unlike the soft bound, within the firm guarantee, the first element outside the substream $Pred$ must be ordered after the firm bound event in the process p. This position satisfies the first possible soft bound in the events ordering. Figure 4 illustrates the notion of the firm bound. As in the previous example, terms $a, b, c, d...$ denote ordered processing events of a process p. However, in this case, event $\langle eoss_{firm}, p, Pred\rangle$ occurs right after the substream terminates.

2.2.4 Consistent termination events order. Some specific applications, including the mentioned earlier epoch-based snapshotting method and techniques for enforcing deterministic processing [16] require an order of termination events to be synchronized with the order of substreams last elements processing. For example, if termination events are reordered, then snapshots for consecutive epochs can be inconsistent. Another example is deterministic join that also requires the defined order of termination events [12].

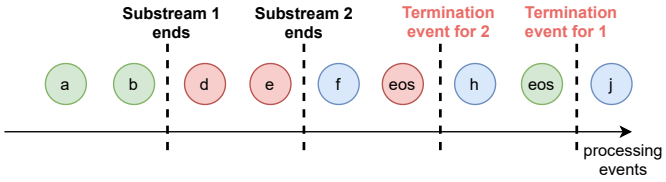


Figure 5: An example of termination events reordering

Termination events reordering in case of the soft bound guarantee is illustrated in Figure 5. Terms $a, b, c, d...$ denote ordered processing events of a process p. Although the substream containing events a, b terminates earlier, the end-of-substream event for this substream occurs after the termination event for the substream containing events d, e.

Let e_1^* and e_2^* be the last elements of substreams defined by predicates $Pred_1$ and $Pred_2$. Termination events $\langle eoss, p, Pred_1\rangle$ and $\langle eoss, p, Pred_2\rangle$ are *consistently ordered* iff:

$$e_1^* >_p e_2^* \Leftrightarrow \langle eoss, p, Pred_1\rangle >_p \langle eoss, p, Pred_2\rangle \tag{7}$$

2.2.5 Optimal traffic overhead. A vital performance property of a substream management system is the amount of extra network traffic. Let $|\Pi|$ be a number of processes, and K be a number of substreams.

Lemma 1. *The network overhead induced by a substream management system cannot be lower than* $O(K|\Pi|)$.

Proof. We assume one by one processing of substreams for them to be isolated (e.g. epochs). When a substream management system detects the termination of a substream, each stateful process should be informed about this. Hence, at least one network message (termination notification) must be received by each process for each substream. □

2.3 Punctuations framework

The main idea behind the punctuations framework is to inject special data elements $\mathcal{P}^{pred}$ into data stream one per substream. These elements, called punctuations, flow down the workflow as ordinary data elements. The injector promises that all elements after punctuations won't satisfy the predicate. Hence, the punctuation itself defines the "border" of a substream.

Figure 6 illustrates the punctuations framework. Green elements indicate elements that belong to some substream, while red elements do not. As we can see, punctuations play the role of delimiter between the substream elements and all further items.

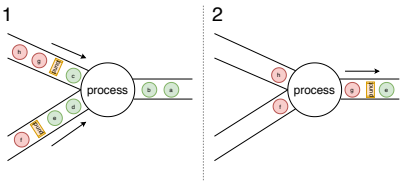


Figure 6: Punctuations handling by a single process

Processes within SPE do not apply user-defined operators to punctuations. Instead, each process p propagates punctuation messages $\mathcal{P}^{pred}_{pq}$ to all outgoing channels $c_{pq} \in O_p$ when it receives corresponding punctuations from all input channels I_p.

Lemma 2. *Generating event by following rule make it a soft bound of the substream pred:*

$$\forall q \in I_p, \exists \mathcal{P}^{pred}_{qp} \in B_p, \forall M \in B : \neg pred(M) \vee dst(M) \neq p \quad (8)$$

Proof. We can use indirect proof. Let $\langle proc, p, M^*, M' \rangle$ be a processing event that happens after the soft bound termination event but $pred(M^*)$. In other words, there is a message $M*, pred(M^*)$ that arrived after all punctuations for the predicate $pred$ had been arrived. According to the definition of a distributed system from Section 2.1, message M^* could emerge either from the mailbox of a process or from incoming channels. The emergence from the mailbox contradicts the condition of the termination event generation rule $\forall M \in B : \neg pred(M) \vee dst(M) \neq p$. On the other side, the emergence from an incoming channel contradicts the condition that punctuations arrived from all channels $\forall q \in I_p, \exists \mathcal{P}^{pred}_{qp}$ because message $M*$ cannot be reordered with punctuations by design. □

To satisfy the firm bound guarantee, the mailbox controller should block processing of all incoming messages from a channel as soon as it receives punctuation from this channel. In [7] such behavior is called *watermark (punctuation) alignment*. Formally we can rewrite this requirement in terms of event ordering:

Lemma 3. *A soft bound becomes firm if a process event order satisfy the following conditions:*

$$\forall e_1, e_2 = \langle recv, p, \mathcal{P}^{pred}_{q_{1,2}p} \rangle, \nexists e' = \langle proc, p, M_{q_1 p}, M' \rangle, e_1 <_p e' <_p e_2 \quad (9)$$

Proof. Let us suppose that there is a message M_{qp} of a next substream that was processed after the last element of the current substream, but before the generation of a bound event. This message either came from the channel q before a punctuation from that channel generating a bound, or was processed before all channels delivers their punctuations. The first case could happen if M_{qp} was reordered with the punctuation along the processing path and contradicts with FIFO processing logic (see previous proof for details). The second case is impossible because of processing limitations introduced by 9. □

3 TRACKER FRAMEWORK

A substream management system should inform all processes that a substream ends, so the amount of extra traffic cannot be lower than $O(K||\Pi||)$. To achieve this lower bound, one can apply an additional agent (process) that receives information about substreams from processes and sends back information about terminated substreams.

In this case, the fact that substream terminates is propagated through this agent without broadcasting between processes, so the amount of extra traffic can be linear by the number of processes. Such propagation method is suitable for cyclic dataflows because there is no need to forward service traffic through the cycles. Therefore, we design a *tracking agent* that:

(1) Receives signals from data producers that a substream has terminated.
(2) Watches for in-flight elements and substreams.
(3) Notifies dataflow processes when the substream ends *for them*, i.e., when they stop receiving elements which satisfy some predicate.

The general scheme of the trAcker mechanism is shown in Figure 1. A special tracking agent receives signals from data sources, fetches information about in-flight elements, and then decides where to send *end-of-substream notifications* (NEOSS).

This substream notifications distribution can be more efficient in terms of network traffic but provides new challenges. Before diving into implementation details, we should answer the following questions regarding trAcker framework:

Q1 How to monitor in-flight elements? To detect that a substream ends, the tracking agent should receive the corresponding signal from data producers and ensure no substream in-flight elements.

Q2 How to ensure bound guarantees? While there are no longer special elements in the stream that denotes the substream end, we need to design soft and firm substream bound conditions based on NEOSS from the shared agent.

Q3 How to provide a consistent termination events order? Unlike punctuations, trAcker notifications are completely async with dataflow elements because they go through another network channel. Hence, dataflow items and notifications are not ordered, making it hard to ensure that the notifications order is consistent.

Q4 What functional and performance properties does the trAcker have? trAcker framework is designed to eliminate the limitations of punctuations framework. We should demonstrate that it is suitable for cyclic dataflows as well as can provide lower network overhead.

3.1 Answering Q1: How to monitor in-flight elements?

To start tracking of a data element the system sends an $SND(pred, M, \emptyset)$ notification to the tracker. Then each process sends the following report messages on each $\langle proc, p, M, M' \rangle$ event:

(1) For all output elements $m \in M'$ for all substreams they belong to $pred(m) = 1$: $SND(pred, m, p)$
(2) For the input element M and all satisfying substreams $pred(M) = 1$: $RCV(pred, M, p)$

Further in this paper, we will denote them as *SND report* and *RCV report*. Note that this communication scheme is heavily optimized in practice. The order of SND and RCV messages is important because each pair of these events forms a chain ring, and sending SND before RCV links these rings together. We can use these chains to track data element processing for the whole workflow graph or its part.

Chains of SND and RCV messages allow the trAcker to track processing of a data element along with a workflow graph. This idea is not new and used in Apache Storm Acker but despite the technical similarity of the core idea[2], Acker and trAcker play different roles in SPE. Acker ensures that the system entirely processed an input element and notifies the user when the processing runs out of time. trAcker tracks an entire substream and allows to define its bounds.

3.2 Answering Q2: How to ensure bound guarantees?

To detect a substream bound, a process needs to ensure that input channels will provide no more elements of this substream. In case of the punctuations framework, the watermark messages carry this guarantee. In case of shared agent, NEOSS messages play the same role. We can assume that each input channel c comes from a segment of the workflow W_c graph. NEOSS is sent to a process when:

- for all incoming channels $c \in I_p$ corresponding segment W_c contains no elements of the substream in-flight (has unpaired SND report);
- all data providers have promised to send no more elements of the substream.

It is easy to show that we can join workflow segments for all incoming channels $W_p = \cup_{c \in I_p} W_c$ and track a single subgraph W_p per process. Using the properties of NEOSS now we can define a soft bound criterion:

Lemma 4. *Soft substream bound could be generated by following rule:*

$$\exists NEOSS \in B_p, \forall M \in B_p : \neg pred(M) \vee dst(M) \neq p \quad (10)$$

[2]Good old XOR commutativity trick.

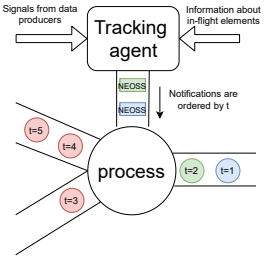


Figure 7: trAcker framework: tracking agent sends NEOSS elements according to the order on t(m)

Proof. If a substream data element is processed after the defined point in events order, it either comes from the mailbox or from one of the incoming channels $c \in I_p$. The first case contradicts $\forall m \in B_p : \neg pred(m)$. The second case could happen because a new substream element enters the system (source broke the promise) or a substream element inside the W_p without the SND/RCV chain (contradicts with SND/RCV chains generation rule). □

To satisfy the firm bound guarantee, one needs to hold elements that do not belong to the substream in the mailbox until NEOSS has arrived. This technique is quite similar to the punctuations alignment behavior mentioned in the previous section. If this condition is satisfied, then $\langle eoss_{firm}, Pred \rangle = \langle eoss_{soft}, Pred \rangle$ for the trAcker.

3.3 Answering Q3: How to provide the consistent termination events order?

In the punctuations framework, such order is provided by design because punctuations and ordinary data items go through the same FIFO network channels. In trAcker this order should be enforced. Assume that SPE assigns to the messages a special totally ordered label $t(M)$. All messages generated by single processing inherit the label from the input message.

In this case, if the order on $t(M)$ coincides with the order of input elements, then trAcker can produce the NEOSS events according to this order as well. In other words, trAcker can reorder the NEOSS events such that they will be consistent with the substreams order. An example of this concept is shown in Figure 7. The substream containing element with $t = 1$ ends before the substream containing element with $t = 2$. As we can see, the order of NEOSS elements from trAcker coincides with $t(M)$.

A vital question here is how to implement the assignment of ordered labels $t(m)$. One way is to use the *time oracle* service [31] which can provide totally ordered labels. A simple alternative is discussed in the next section.

3.4 Answering Q4: What are the functional and performance properties of trAcker?

trAcker does not require regular broadcasting of the elements to all computational nodes because all service traffic goes through a single agent. This change allows trAcker to have the following features by design:

(1) **Cyclic dataflows support.** Because the tracking agent is monitoring the properties of in-flight elements without directly injecting service items into a dataflow, trAcker does not have the problem of throwing them through a cycle.
(2) **Low network overhead.** Processes can send reports once per a fixed time period, so there is a constant time of such reports per a finite substream. The reports require $O(|\Pi|)$ extra messages, while the NEOSS events $O(K|\Pi|)$. The total amount is $O(K|\Pi| + |\Pi|) = O(K|\Pi|)$ that is optimal for the substream management problem.
(3) **Low latency and impact on SPE throughput.** Punctuations can be stuck by other data elements if they are sent with some delay after the last substream element. In trAcker, service traffic goes through other network channels that can reduce latency between actual substream termination and the corresponding event. Together with the low amount of service traffic, this scheme does not significantly reduce the throughput of an SPE, as we show in Section 5.

4 TRACKER IMPLEMENTATION

In the previous section, we introduced a general schema of the trAcker framework. In this section, we deepen into its implementation details. We describe and explore the properties of the tracking agent that produces the substream termination notifications (NEOSS). After that, the technique to achieve consistent termination events order is detailed.

4.1 Bound guarantees

The tracking agent splits the workflow graph into partially ordered segments and tracks them separately. For each process p, we can generate a list of preceding segments that include a set of incoming messages generators W_p for the process. As soon as all these segments contain no elements of a substream, the agent sends to a process *NEOSS*.

To track the messages path through segments, the agent receives SND/RCV reports containing a segment identifier and a list of predicates the message satisfies. The agent aggregates this information into the table illustrated in Table 3.

Table 3: trAcker table: a general example

Notified	Predicate	Segment	**Substream elements**
✓	h(x)	A	**No**
		B	**No**
	q(x)	A	**No**
		B	**Yes**
✓	z(x)	A	**No**
		B	**No**

There are several possible methods to build the indicator that the segment contains elements from a substream using the reports from processes. Our implementation uses the trick applied in Apache Storm to monitor the completeness of processing [23].

Each report is labeled by a random number X, and this number is the same for the send action and the corresponding receive action. This trick makes it easy to check if the segment contains a full set of *SND*/*RCV* pairs for a message: XOR operation for all numbers received from the chain will turn into 0. The result of the XOR operation can accidentally become zero, but the probability of this event is controlled by the length of the random number X so that it can be neglected in practice [23].

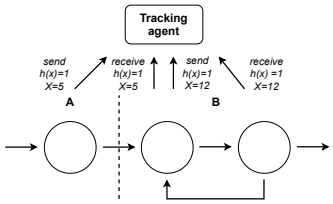


Figure 8: Reports example

Figure 8 illustrates the reports with random numbers. The elements satisfying a predicate $h(x)$ flow through a dataflow. The first process generates an output and sends the SND report with $X = 5$. The corresponding RCV report by the second process also has $X = 5$ because this process receives the element from the first one. The second process sends a new element that satisfies $h(x)$ further, and the new SND report with $X = 12$ is produced. The third operator receives this element and also produces an RCV report with $X = 12$. If there are no more elements such that $h(x) = 1$, then we can send NEOSS for the substream defined by $h(x)$ ends because *5 XOR 5 XOR 12 XOR 12 = 0*. Table 4 illustrates the actual trAcker table for the mentioned technique.

Table 4: trAcker table: XORing technique example

Notified	Predicate	Segment	**Segment XOR**	**XOR**
✓	h(x)	A	**000**	**000**
		B	**000**	
	q(x)	A	**000**	**110**
		B	**110**	
✓	z(x)	A	**000**	**000**
		B	**000**	

Due to the associativity of XOR, we can optimize tracking agent incoming traffic by aggregating reports locally within the processes. For each process, we introduce a *local tracking agent* component. It serves as a mediator between the process and the global agent, buffering the outgoing reports and flushing them periodically.

The flushing window is the parameter that allows us to balance the service traffic and latency between the actual substream termination (the event from the data producer) and the termination event. Note that substreams last a finite time period by definition, so we assume that each process sends aggregated reports a constant number of times that does not depend on the number of substreams and processes. Therefore, the amount of extra network traffic for the reports is $O(|\Pi|)$, so the total estimation with the overhead on the notifications is $O(K|\Pi|)$.

4.2 Consistent termination events order

If the order of NEOSS is consistent, the order of termination events will be consistent as well. To achieve consistent order of NEOSS, we need to define $t(M)$ such that the order on $t(M)$ respects the order of input elements. All reports that processes send to the tracking agent should be labeled with $t(M)$. In turn, the tracking agent sends the NEOSS elements according to the order on $t(<)$.

Table 5 illustrates the trAcker table in case of consistent NEOSS order. Column *min t(x)* indicates the minimal $t(x)$ among the elements that satisfy the corresponding predicate. The tracking agent sends notifications for the substream if the *XOR* value is 0 and all substreams that contain elements with less *min t(x)* have finished (notifications have been produced). Therefore, NEOSS for the substream defined by the predicate $h(x)$ is not sent until the NEOSS for the predicate $q(x)$ is generated.

Table 5: trAcker table: consistent NEOSS order example

Notified	Predicate	**min t(x)**	XOR
waits for q(x) finish	h(x)	**5**	000
	q(x)	**4**	110
✓	z(x)	**1**	000

If input elements arrive through a single node, $t(x)$ can denote the monotonic system time of the element x arrival. If there are multiple source processes, one can use time oracle agent [31] as a service for generation a monotonic sequence of unique timestamps. However, in this case, there is a need to manage one more subsystem.

A simple technique to build $t(x)$ without extra agents bases on systematic synchronization of the system clocks. We call this method and associated labels *coarse time*. Assume that clock differences are no more than some fixed δ, which we reference as synchronization slack. Let $\tau(x)$ be precise physical time of input data item x arrival, and $s(x)$ be local system time of the source node where x arrived. The true order of events $\tau(d_1) > \tau(d_2)$ coming from different sources can be sometimes restored by their system timestamps $s(d_1)$ and $s(d_2)$. If these timestamps differ more than time synchronization slack, then the order is clear: $s(d_1) > s(d_2) + \delta \Rightarrow \tau(d_1) > \tau(d_2)$.

This fact allows us to define $t(x)$ such that $t(x) = [s(x)/\delta]$. This way we make $t(x)$ less precise, but this trick gives us an ability to compare global time associated by different source nodes. If $t(x_1)$ is greater than the $(t(x_2) + 1)$ then their order is defined even if they arrived from different source nodes: $t(x_1) > t(x_2) + 1 \Rightarrow \tau(x_1) > \tau(x_2)$. Therefore, the order on $t(x)$ coincides with the order of input elements, so it is suitable for the defined problem. Here is a summary of the mechanism that ensures consitent termination events order:

(1) On input item arrival, source node gets the system timestamp
(2) The system timestamp is shrunk up to synchronization slack (practically we achieve 10ms slack)
(3) Each report for the tracking agent is labeled by the result of $t(x)$
(4) Tracking agent sends NEOSS according to the order on $t(x)$
(5) Termination events are generated according to the order of NEOSS arriving

5 EXPERIMENTS

In previous sections we put several statements out: trAcker framework provides low service traffic, low latency between the actual substream termination and termination event receiving, and SPE throughput does not depend on the substream size. In the experimental part of the paper, we will examine these properties on synthetic and real-world examples.

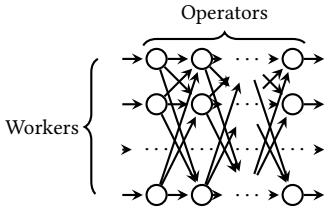


Figure 9: Physical execution graph for experiments

As a baseline approach, we utilize the punctuations-based method employed in many state-of-the-art stream processing systems such as Flink [7], Storm [24], Heron [15], IBM Streams [13]. To the best of our knowledge, the punctuation framework is the only existing general-purpose substream management technique.

To compare tracking mechanisms, we have to implement them on top of a single SPE. Otherwise, the performance could be affected by inequalities in serialization, network protocols, etc. The difficulty here is that the tracking mechanism is usually a core part of SPE, and the implementation is often optimized for it.

We implemented both punctuations and trAcker techniques on top of FlameStream [17]. It is an open-source Java-based SPE that allows tracking mechanism customization by its design. In [16], authors demonstrated that the performance of the FlameStream is comparable to state-of-the-art SPE Flink. None of the system-specific features were exploited during the implementation of substream management methods. All experiments are performed on virtual machines with a dual-core CPU and 4 GB RAM from one of the major cloud providers.

In the next sections, we will explore our system in four experimental setups:

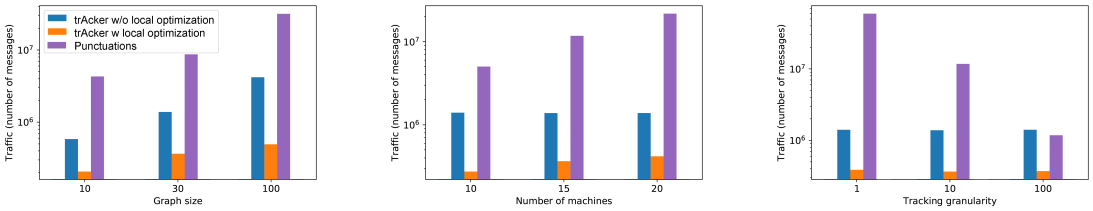
Service traffic: the last setup aims to demonstrate the asymptotic behavior of competitive approaches. We show that our theoretical estimation meets practical measurements.

Latency: we explore both the notification latency (a delay between a moment when a substream ends and the reception of the termination event for this substream) and the end-to-end processing latency. We compare the trAcker and punctuations approaches on synthetic and real-world workloads in soft and firm bounds.

Maximum SPE throughput: we study the influence of the notification mechanism on the maximum throughput of the system using the synthetic workload. This experiment shows an overhead induced by a substream management on regular data processing.

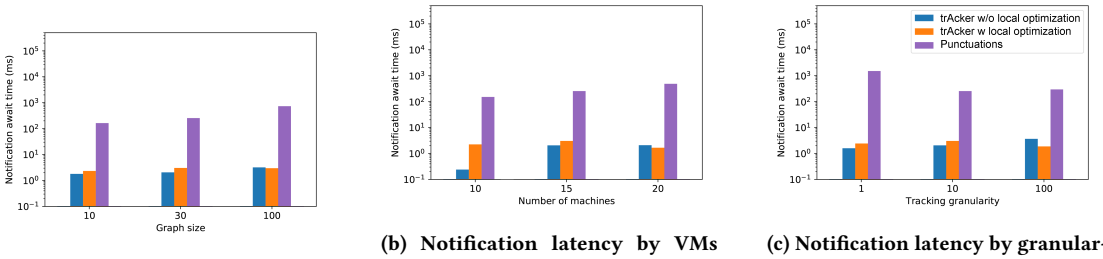
Substream management for cyclic graphs: trAcker framework supports substream management for cyclic dataflows. In this setup, we demonstrate the benefits of this approach for a novel problem setup: state smart caching.

In our study, we use a common synthetic workload shown in Fig. 9. All vertices pass an input element to the next operation. On each step, items are re-partitioned (round-robin). This shape of the physical execution graph allows us to measure the properties of the system with the growth of key switches in a workload. This



(a) Traffic by graph size (b) Traffic by number of VMs (c) Traffic by tracking frequency

Figure 10: Service network traffic of punctuations approach and various trAcker setups



(a) Notification latency by graph size (b) Notification latency by VMs number (c) Notification latency by granularity

Figure 11: Notification latency

setup has no computational overhead while covering many realistic scenarios. Graphs with few vertices (under 30) may fit almost any acyclic streaming pipeline [2]. Longer instances of this workload correspond to flattened iterative dataflows such as PageRank or Connected Components [19, 29]. We reference this synthetic workload as RR-N, where N is the number of steps.

5.1 Service traffic

In the theoretical part of the paper, we put a statement that the service traffic of the trAcker grows linearly with the number of processes and granularity of the tracking. In this section, we prove this statement in practice.

We can measure the extra load provided by tracking mechanisms in a number of service messages sent over the network. In trAcker, there are several types of these messages: SND and RCV reports, and NEOSS. In the baseline approach, all service messages are punctuations. If it is not specified, the graph size is 30, the number of VMs is 15, and the granularity is 10.

Figure 10 demonstrates the dependency between service network messages and the size of the logical graph, the number of computational nodes, and the granularity of tracking. The extra service traffic is generated by $50K$ input elements sent with 100 items per second arrival rate.

As shown in Figure 10a, service traffic for punctuations linearly depends on the dataflow size because each new logical vertex adds network broadcasts of punctuations on a physical level. Dependency from the number of computational nodes is quadratic [3] due to the need to broadcast punctuations to each node after each operator, as it is demonstrated in Figure 10b.

[3] At first glance, the dependency may seem linear, but please note that the X-axis covers range from 10 to 20, and the Y-axis is log-scaled

Figure 10c indicates that the number of sent service messages for punctuations also directly depends on the tracking granularity. For example, the system should broadcast punctuations after each streaming element in every operator to implement tracking of individual items.

In the case of trAcker, service traffic depends on the logical graph size and the number of machines because their product forms the number of processes. The growth has a linear trend but can be significantly reduced with the local trAcker optimization. Tracker without optimizations provides 1.5-5x less service traffic than puctuations. Local trAcker optimization allows the system to reduce traffic up to 30 times compared to the punctuations.

5.2 Microbenchmarks: notification latency

One of the key performance metrics is the latency of notifications: a delay between a moment when a substream ends and the reception of the termination event. This time is added to any operation triggered by termination events [7, 16]. For example, Flink finishes its state snapshotting protocol for the epoch (set of input elements) and delivers corresponding output elements to data consumers only after receiving a notification that the whole epoch is completed. We examine end-to-end latency in the next section.

In this experiment, we measure the notification latency as an interval duration between a promise from the data source that it will never generate substream elements and the reception of the termination event for this substream. As an experimental workload, we use synthetic workload RR-N. We investigate the dependency between the notification latency and cluster sizes, the length N of the workload, and the granularity of tracking (number of elements within a substream). Figure 11 shows the results of the experiment.

The notification latency of a punctuations-based technique depends on the graph and cluster sizes and the granularity of tracking as figures 11a,11b, and 11c indicate. These results are in-line with the overhead induced by punctuations shown in Section 2.3. Notification latency of trAcker slightly fluctuates but does not directly depend on the investigated parameters.

The difference in latency between the punctuations and trAcker appears because each operator in a dataflow must wait for punctuations from all partitions to sign off the previous operator to send it further to ensure the correctness.

5.3 End-to-end latency

In the previous section, we demonstrated that trAcker framework provides lower notification latency than the baseline. In this part, we show how this difference influences end-to-end latency. We use real-world workloads: a window join and a state snapshot. We measure the latency of window join as a time between the last element of the window enters the system and the output of window aggregation. We also examine how substream management techniques can affect the duration of taking a state snapshot. In both scenarios, end-to-end latency is directly affected by the notification latency.

5.3.1 Window join. For the windowed join scenario, we apply NEXMark benchmark [26] designed to inspect the performance of streaming queries. This benchmark extends the XMark benchmark [22] online auction model, where users can start auctions for items and bid on items. We accept Query 8 from the NEXMark benchmark, defined as follows: *Select people who have entered the system and created auctions in the last period.* This query can be implemented using windowed join of persons and auctions. We apply 10 seconds window and 500 RPS per node input rate.

Figure 13a illustrates the results. The end-to-end latency (a time between the last element of the window enters the system and the output of window aggregation) within trAcker is under 20ms for all setups, while within punctuations, latency grows up to 300ms on 30 nodes. After that, it fluctuates slightly. This behavior can be explained by the fact that the process waits for punctuations from all channels to produce query results. However, with the growth of machines number, the probability that some node delays punctuation increases. After some limit (30 nodes in our case), many nodes start to delay punctuations, so the latency achieves its maximum.

5.3.2 State snapshotting. As we mentioned above, an important application of substream tracking mechanisms is state snapshotting. Typically, state snapshotting is implemented as follows: a streaming system divides input records into the contiguous substreams called epochs. When an operator entirely processes all items from a particular epoch, it blocks all inputs and persistently saves its local state. Each operator receives the termination event for an epoch and starts to save its state independently from other operators. Flink [7], Storm [24], IBM Streams [13], and Heron [15] implement this state snapshotting scheme. All these systems use punctuation-based techniques to provide notifications for operators.

Punctuations mechanism can imply latency overhead on this protocol. The overhead is caused by blocking an operator after the first punctuation is received until the operator receives punctuations from all inputs. This behavior is known as *punctuations alignment* issue [7]. In the case of trAcker, an operator must buffer elements from the next epoch until the NEOSS for the previous epoch is received.

Note that local state snapshotting process can be asynchronous in the modern SPEs such as Flink [7]. It implies that local snapshotting duration is not necessarily linked to the state size. Nevertheless, the latency of the termination signal for an epoch directly affects the latency of data elements during global state snapshotting because an operator cannot start process elements from a new epoch until it receives a termination signal for the previous one.

Figure 12 demonstrates SPE latency spikes during state snapshotting for punctuations and trAcker, depending on the persistent save duration. In general, trAcker provides 50-120 milliseconds fewer latency spikes. This difference can be significant for latency-conscious applications [32]. The low notification latency explains this difference, as we demonstrated in Section 5.2.

5.4 End-to-end throughput

In this experiment, our goal is to find how the substream management influences the maximum throughput of an SPE. We measure the median latency of RR-30 workload using a cluster of 20 nodes, depending on the input rate (input elements per millisecond). The growth of median latency indicates system overloading. Input rate that corresponds to the point where latency starts to grow indicates a *sustainable throughput* [14].

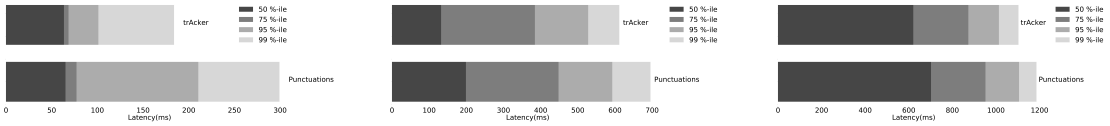
Figure 13b shows that a system without tracking at all starts to be overloaded at $\sim 9K$ requests (items) per second input rate. The system with the finest-grained trAcker setup sustains $\sim 7K$ RPS throughput. Overloading with the punctuations-based approach depends on the granularity of tracking: the finest-grained setup does not sustain even $1K$ RPS, while the setup with the granularity of 10 has $\sim 2K$ RPS throughput. Punctuations achieve similar to trAcker throughput ($\sim 5K$ RPS) only when they are injected once per 50 input elements.

This experiment shows that punctuations significantly bound throughput of regular processing within the fine-grained setups. It is explained by the heavy extra network traffic that we demonstrated in Section 5.1. Note that this additional traffic goes through the same network channels as ordinary data items to ensure that punctuations do not overtake ordinary records. On the other hand, trAcker provides less additional system load due to lower extra network usage and the exploiting of additional network channels.

5.5 Substream management for cyclic graphs

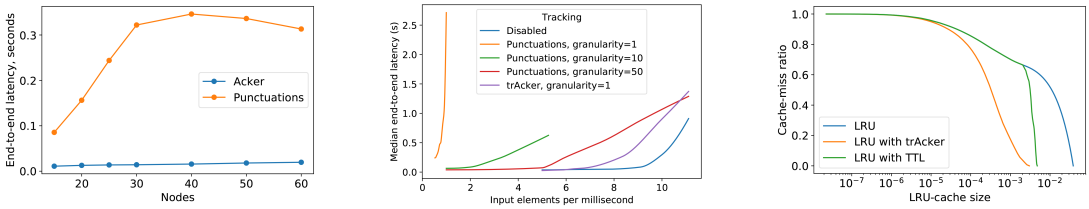
One more practical application of substream tracking is in-memory state optimization. In practice, we often use either short-life keys such as session-id, cart-id, or a subset of a wide variety of keys, such as users of social networks. The state associated with such keys should be removed from the memory once the key becomes obsolete. We refer to this task as *distributed garbage collection.*

Substream management can be used to solve this task because we can consider elements bearing a specific key as a substream. In this experiment, we use a trAcker for *distributed garbage collection* and compare this approach to more widespread caching techniques.



(a) 100 ms snapshot duration (b) 500 ms snapshot duration (c) 1000 ms snapshot duration

Figure 12: SPE latency spikes during state snapshotting



(a) Nexmark window join scenario (b) Overhead on an SPE throughput (c) State smart cache scenario

Figure 13: Real-world scenarios and maximum throughput

In the experimental setting, we solve a depth search problem for a graph of Twitter users. This task is relevant for social recommendation systems and requires both scalability and freshness of the results due to the dynamic nature of the graph and the number of nodes. An SPE is a suitable candidate host for such a solution due to the low latency (freshness) requirement.

Within this model, the user is the key, and the set of her subscribers is the state associated with the key. A stream of queries of user neighborhoods is issued to the system. A resulting set of unique users is expected in return. The execution graph for the depth search problem is cyclic, so trAcker suits this task.

We compare *distributed garbage collection* approach with LRU caching for this task. The keys are removed from the hot set as soon as none of the currently processed messages contain a reference for it. We used a ratio of cache-misses during processing as a measure of the effectiveness of the memory management mechanism.

In this experiment, we used 4 nodes and fixed the request per second rate to 20. In the Fig. 13c results of this experiment are presented. With the growth of the allocated memory, the cache becomes effective, though the difference between *distributed garbage collection* and LRU (with or without optimal TTL) counts in order of magnitude. We believe that this type of task needs to be resolved with GC instead of caching.

6 RELATED WORK

A comparison between various substream management techniques is summarized in Table 1. To the best of our knowledge, the punctuations framework is the only substream management mechanism that supports arbitrary predicates. Flink [7], Storm [25], Samza [21] apply punctuations for both window aggregations and state snapshotting problems. MillWheel [1] has another state management model but also uses punctuations as a window end indicator.

Spark Streaming [30] method for state pruning is based on the punctuations as well. A recent work [4] proposes an idea that a centralized agent can aggregate watermarks. This agent computes minimum over all of its node watermarks and then broadcasts reports. However, the aggregation and reporting techniques are not detailed, so the formal properties of this approach are unclear.

Several techniques aim to track some specific properties of a stream but do not provide the general substream management framework. Apache Storm *Acker* [23] is a method for completeness monitoring that also uses XOR operation properties under the hood. It allows SPE to detect if some element has been lost during the processing. However, Acker does not provide mechanisms for tracking arbitrary substreams, so it is not applicable for window termination and state pruning problems.

Naiad [19] has a mechanism for tracking the progress of distributed iterative computations. This method can be used to check the convergence criteria of iterative algorithms or limit iterations. The main commonality between the Naiad progress tracker and the trAcker is that information about iterations or substreams is propagated through network channels which are not used for data processing. However, in trAcker all termination events are provided by the tracking agent, while in Naiad each node generates progress events based on information from all other nodes. It implies that the amount of extra traffic required by this mechanism is quadratic from the number of nodes even if all updates go through a centralized accumulator.

Another technique for tracking centralized iterative processing is introduced in [10], but it is not yet adapted for distributed processing. Although these methods are robust for progress tracking, they are also currently unsuitable for window termination or state pruning problems.

7 CONCLUSION

In this work, we formalized the problem of substreams management. We designed and implemented a new substreams management technique called trAcker that does not require injecting service elements directly into the stream. Instead, we mark all data elements with ordered labels and use the distributed agent, which notifies

operators that a substream ends. Our approach has the following features:

- **Cyclic dataflows support:** the method is suitable for problems that require non-linear executions: graph traversing, iterative algorithms, etc. We evaluated this feature within the real-life problem.
- **Low overhead:** we showed that our implementation achieves the lower bound of service traffic overhead. We demonstrated that trAcker insignificantly affects the throughput of an SPE in practice.
- **Fine-grained substreams support:** trAcker framework is suitable for substreams consisting of a small number of elements. This feature is achieved due to low traffic overhead and another way of notifications propagation.

The centralized agent is a limitation of a solution presented in this paper. The first problem is scalability. Due to the page limit, we omitted the discussion about the distributed trAcker agent. Fault tolerance is another problem because we should ensure recovery of the trAcker agent in case of failures. We are leaving both topics for future work.

REFERENCES

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

[2] T. Akidau, S. Chernyak, and R. Lax. 2018. *Streaming Systems: The What, Where, When, and how of Large-scale Data Processing.* O'Reilly Media, Incorporated. https://books.google.ru/books?id=48-BAQAACAAJ

[3] Ahmed Awad, Jonas Traub, and Sherif Sakr. 2019. Adaptive Watermarks: A Concept Drift-based Approach for Predicting Event-Time Progress in Data Streams.. In *EDBT*. 622–625.

[4] Edmon Begoli, Tyler Akidau, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow. *Proc. VLDB Endow.* 14, 12 (2021), 3135–3147. http://www.vldb.org/pvldb/vol14/p3135-begoli.pdf

[5] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1757–1772. https://doi.org/10.1145/3299869.3314040

[6] Paris Carbone. 2018. *Scalable and Reliable Data Stream Processing.* Ph.D. Dissertation. KTH Royal Institute of Technology.

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

[8] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. *ArXiv e-prints* (June 2015). arXiv:1506.08603 [cs.DC]

[9] 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).

[10] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[11] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[12] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data* (2016).

[13] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent regions: Guaranteed tuple processing in ibm streams. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1341–1352.

[14] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1507–1518.

[15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proc. of the 2015 ACM SIGMOD Intnl. Conf. on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[16] Igor E. Kuralenok, Artem Trofimov, Nikita Marshalkin, and Boris Novikov. 2018. Deterministic Model for Distributed Speculative Stream Processing. In *Advances in Databases and Information Systems*, András Benczúr, Bernhard Thalheim, and Tomáš Horváth (Eds.). Springer International Publishing, Cham, 233–246.

[17] Igor E. Kuralenok, Artem Trofimov, Nikita Marshalkin, and Boris Novikov. 2018. FlameStream: Model and Runtime for Distributed Stream Processing. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond* (Houston, TX, USA) *(BeyondMR'18)*. ACM, New York, NY, USA, Article 8, 2 pages. https://doi.org/10.1145/3206333.3209273

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

[19] 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* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[20] Hannaneh Najdataei, Yiannis Nikolakopoulos, Marina Papatriantafilou, Philippas Tsigas, and Vincenzo Gulisano. 2019. Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*. 7–18.

[21] 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.

[22] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J Carey, Ioana Manolescu, and Ralph Busse. 2002. XMark: A benchmark for XML data management. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 974–985.

[23] storm-site 2018. *Apache Storm documentation, Guaranteeing Message Processing.* https://storm.apache.org/releases/current/Guaranteeing-message-processing.html

[24] storm-site 2018. *Apache Storm documentation, Storm State Management.* http://storm.apache.org/releases/1.2.1/State-checkpointing.html

[25] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[26] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT).* Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

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

[28] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[29] Chen Xu, Markus Holzemer, Manohar Kaul, and Volker Markl. 2016. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 613–624.

[30] 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 *Proc. of the 4th USENIX Conf. on Hot Topics in Cloud Ccomputing* (Boston, MA) *(HotCloud'12)*. USENIX Association, Berkeley, CA, USA, 10–10.

[31] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[32] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 614–630.

[33] Zhan Zhang, Wenhao Li, Xiao Qing, Xian Liu, and Hongwei Liu. 2021. Research on Optimal Checkpointing-Interval for Flink Stream Processing Applications. *Mobile Networks and Applications* (2021), 1–10.