

# Event Stream Processing with Out-of-Order Data Arrival

Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner and Murali Mani  
Department of Computer Science, Worcester Polytechnic Institute  
Worcester, Massachusetts 01609, USA  
(minglee|liumo|lisading|rundenst|mmani)@cs.wpi.edu

## Abstract

*Complex event processing has become increasingly important in modern applications, ranging from supply chain management for RFID tracking to real-time intrusion detection. The goal is to extract patterns from such event streams in order to make informed decisions in real-time. However, networking latencies and even machine failure may cause events to arrive out-of-order at the event stream processing engine. In this work, we address the problem of processing event pattern queries specified over event streams that may contain out-of-order data. First, we analyze the problems state-of-the-art event stream processing technology would experience when faced with out-of-order data arrival. We then propose a new solution of physical implementation strategies for the core stream algebra operators such as sequence scan and pattern construction, including stack-based data structures and associated purge algorithms. Optimizations for sequence scan and construction as well as state purging to minimize CPU cost and memory consumption are also introduced. Lastly, we conduct an experimental study demonstrating the effectiveness of our approach.*

## 1 Introduction

Event stream processing has raised increased interest in the database and distributed systems communities in the past few years [1, 10, 2, 3, 9]. A wide range and ever growing numbers of applications nowadays, including network monitoring, e-business, health-care, financial analysis, and security supervision, rely on being able to process queries over data streams that take the form of time ordered series of events.

Let us consider a popular application for applying event sequence tracking techniques, namely, shoplifting in bookstores. RFID tags are attached to each book and RFID readers are placed at different locations throughout the store, such as book shelves, checkout counters and the store exit. If a book shelf and a store exit sensed the same book but none of the checkout counters sensed it in between the occurrence of the first two events, then we can conclude that this book is being shoplifted.

Event queries, such as those needed above to detect shoplifting, have been tackled in the literature. For instance, SASE [10] proposes an expressive yet easy-to-understand language to support pattern queries on such sequential streams. It also proposes customized algebra operators for the efficient processing of such sequence queries with sliding windows. This technology, being specifically designed for handling sequence queries over event streams, is shown to be superior to generic stream processing solutions [7].

For an event stream processing system if the order in which the events are received by the system is the same as their timestamp order, we say the data arrival of the system satisfies the *total order assumption*. Most systems [10, 2], both event-based and stream-based ones, assume a total ordering among event arrivals. By this assumption, the later arrival of an event implies that it has a larger timestamp than the other events which have already arrived earlier. For example, the query evaluation approach of [10] relies on such total ordering assumption for locating the expected event sequences.

However, out-of-order events are not uncommon in practice. For example, in a distributed computing environment, event sequences might arrive out-of-order at the processing engine due to network traffic and possible node failure. We will illustrate that the existing technology would fail in such circumstances, either missing resulting matches or incorrectly producing incorrect matches. Clearly, for handling out-of-order data arrival, a more sophisticated mechanism is needed. This is the problem we tackle in the paper.

We propose the first solution for evaluating sequence queries over event streams with out-of-order data arrival in this work. The main contributions of this work include:

- We analyze the problems that state-of-the-art event stream processing technology would experience when faced with out-of-order data arrival.
- We propose new physical implementation strategies for the core stream algebra operators such as sequence scan, pattern construction and runtime purge. In particular, we introduce stack-based data structures as well as the associated sequence retrieval, event pattern construction and state purge mechanisms.

- Optimizations for sequence scan and state purging to minimize CPU cost and memory consumption are introduced.
- We conduct an experimental study that demonstrates the effectiveness of our proposed approach over existing solutions.

In Section 2 we describe the event sequence query model and the basic execution approach we assume. Problems caused by the out-of-order data arrival are identified in Section 3. In Section 4, we propose our solution of event stream processing with out-of-order data arrival. An experimental analysis is given in Section 5, while related work is discussed in Section 6. Section 7 concludes this work.

## 2 Preliminary

### 2.1 Events, Event Stream and Sequence Query

**Events.** An event is defined to be an instantaneous occurrence of interest at a point in time. It can be a primitive event or a composite event [2]. Throughout this report, we use capitalized letters to represent event types and lower-case letters to represent event instances. A schema is associated with each event type. It includes the event type ID, a set of application-specific attributes and the timestamp which records the time when the event is generated.

**Event stream.** In most event processing scenarios, it is assumed that the input to the query system is a potentially infinite event stream that contains all events that might be of interest [10, 2, 1]. Therefore, the event stream is heterogeneous populated with event instances of different event types, thereby having different schemas. For example, in the RFID-based retail management scenario explained in [10], all the RFID readings are merged into a single stream and sorted by their timestamps. Hence the stream will contain the SHELF-READING events, the COUNTER-READING events and the EXIT-READING events.

**Event Sequence Query.** Event sequence queries are queries on the sequential event stream. [10] defines a language that can specify how individual event is filtered and how multiple events are correlated via time-based and value-based constraints. In this work, we utilize the SASE query language to express sequence queries with sliding windows. The following is an example using the SASE language for our previous case study, which finds out if any book is being taken out of a book store without going through the store’s check-out counter:

```
EVENT SEQ(SHELF s, !(CHECKOUT c), EXIT e)
WHERE s.id = c.id AND c.id = e.id
WITHIN 1 hour
```

### 2.2 Overview of Query Algebra

We assume here that the input event query has been translated into an algebraic query plan as proposed by [10]. We focus on a query plan based on the following operators: sequence scan (SS), sequence construction (SC), window (WD), selection (SL), and transformation (TF). The SS

operator employs an NFA to detect matches to the event pattern specified in the query. The SC operator constructs the expected event sequences based on events retrieved by SS. SS and SC together form the SSC component in SASE. The SL operator filters event sequences by applying all the predicates specified in the query. The WD operator checks whether events in the input event sequence occur within a sliding window. The TF operator converts each input event sequence into a composite event.

**Example 1.** Figure 1 shows an example query plan for the sequence query  $Q$  depicted in the same figure using the SASE algebra.

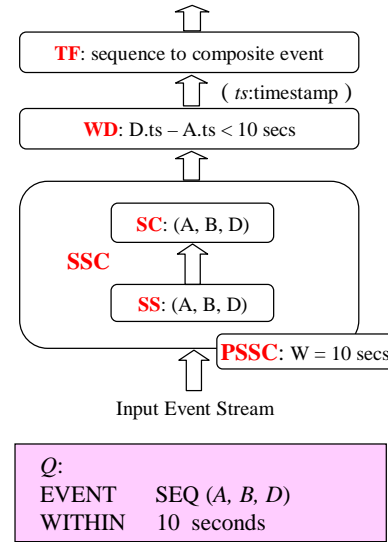


Figure 1. Event Query Plan

#### 2.2.1 Sequence Scan and Construction (SSC)

SSC as the bottom-most operator constructs a nondeterministic finite automaton. Let  $N$  denote the number of events in the query that are not involved in the negation query patterns. Then the number of states in the NFA equal to  $N+1$  (including the starting state). A data structure named Active Instance Stack (AIS) is proposed by [10] for the execution of SSC. That is, instead of using a single stack for the NFA (Figure 2(a)), AIS associates a stack with each state of the NFA storing the events that triggered the NFA transition to this state. The events stored in each stack are called the active instances of this stack. In addition, for each active instance  $e$  in a stack, an extra field is created to record the most recent instance in the stack of the previous state (RIP). Figure 2(c) shows a partial input event stream. The events marked with an underscore are the ones being extracted during the sequence scan. All the retrieved events of type A, B and D are kept by AIS. Figure 2(b) shows the content of the three AIS stacks after the current of stream S depicted in Figure 2(c) has been received. In each stack, the active instances are listed from top to bottom in the order

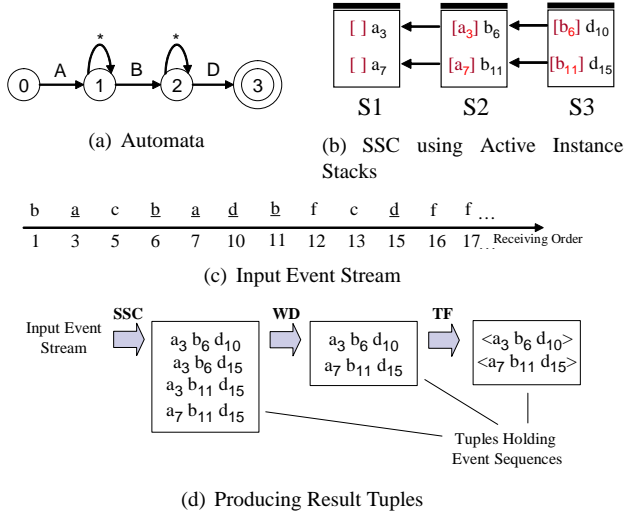


Figure 2. Query Evaluation of SASE

of their arrival. Take the active instance  $b11$  in stack S2 in Figure 2(b). The most recent instance in stack S1 (holding event instances of type A) before  $b11$  is  $a7$ . The RIP field of  $b11$  is thus set to  $a7$ , as shown in the parenthesis preceding  $b11$  in Figure 2(b).

The sequence construction is initiated for each active instance of the accepting state, in our case,  $d10$  and then  $d15$ . With AIS, the construction is simply done by a depth first search in the DAG that is rooted at this instance and contains all the RIP edges reachable from the root. Each root-to-leaf path in the DAG corresponds to one matched event sequence to be returned by this SSC operator. For example, the three event sequences created for the active instance  $d15$  are “ $a3 b6 d15$ ”, “ $a3 b11 d15$ ” and “ $a7 b11 d15$ ”. Thus, after receiving the events in the input stream S depicted in Figure 2(c), the SSC operator should output four event sequences and then two of them will be removed by the WD operator. Totally there are two result sequences being produced, as shown in Figure 2(d).

### 2.2.2 Purge at SSC (PSSC)

State purge on SSC is conducted based on window constraints for removing outdated events from AIS. In this paper we conceptualize this as the *PSSC function* of the SSC operator. Event instances in AIS which fall out of the sliding window will no longer be able to contribute to the query result. PSSC dynamically prunes the event instances at AIS by removing such outdated events. For example, when  $d15$  is retrieved,  $a3$  can be removed from stack S1 because the distance between  $a3$  and  $d15$  is already larger than the allowed window range ( $15 - 3 > 10$ ). Similarly, once  $f17$  is received,  $b6$  can be safely pruned from S2 at AIS because it has been slid out of the window ( $17 - 6 > 10$ ).

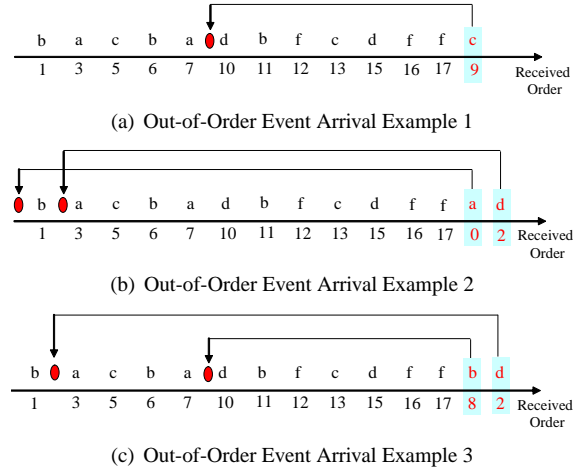


Figure 3. Out-of-Order Data Arrival Examples

## 3 PROBLEMS CAUSED BY OUT-OF-ORDER DATA ARRIVAL

### 3.1 Out-of-Order Event Stream

Operators in SASE queries work on an event’s timestamp, a special purpose attribute showing its generation time. SASE approach assumes a total ordering of all event arrivals, i.e., the order in which the events are received by the query system equal to their timestamp order. The query evaluation approach of SASE relies on this total order assumption for identifying event sequences. However, as mentioned in the introduction, if the input stream were to contain any out-of-order events, such a handling approach becomes insufficient for sequence query evaluation.

**Out-of-Order Event.** For a newly arrived event  $e_m$ , supposed the events that we received before  $e_m$  are  $e_1, e_2, e_3, \dots, e_{m-1}$ , if there exists any  $e_i$  satisfying  $e_m.timestamp < e_i.timestamp$  ( $1 \leq i \leq m - 1$ ),  $e_m$  is an *out-of-order event*.

In the example stream S shown in Figure 3(a), the events are listed under their received order. We can see that event  $c9$  received after event  $f17$  is an out-of-order event. The input event stream no longer satisfies the total order assumption. The out-of-order event  $c9$  should have arrived at the position indicated by a dot above the axis.

### 3.2 Problem for Sequence Scan and Construction

#### 3.2.1 Incomplete Event Retrieval

The current execution logic of NFA in SSC relies on the total ordering assumption. If this assumption no longer holds, some events which should have been kept might be discarded by the sequence scan. We refer to this as *incomplete event retrieval*.

**Example 2.** Consider the example event stream in Figure 3(b). Two out-of-order events,  $a0$  and  $d2$ , came after  $f17$ . The dots in the figure indicate the positions at which these

two out-of-order events should have arrived under the event timestamp order. We can see that “ $a0\ b1\ d2$ ” is an event sequence which should be constructed by the SSC. However, during the event retrieval of SSC by using NFA, when  $b1$  arrives, automaton state  $s2$  hasn’t been activated yet. Hence,  $b1$  will simply be discarded. At the moment when the  $a0$  and  $d2$  are received, the event  $b1$  is gone. Thus the sequence “ $a0\ b1\ d2$ ” is missed.

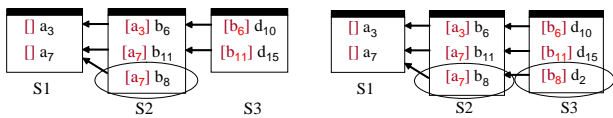
From the above example we observe that such incomplete event retrieval potentially causes some qualified event sequence being missed.

### 3.2.2 Event Misplacement

The retrieved events during the sequence scan will be placed in AIS for event sequence construction. Based on the total order assumption, newly arriving events are placed on top of the corresponding stack in AIS. For example, when  $a7$  is retrieved, it will be put on top of stack  $S1$ . Under the total order assumption, this simple “append” approach works correctly. However, with out-of-order event inputs, located events might be placed into the wrong spot in AIS during sequence scan. We refer to this as *event misplacement*.

**Example 3.** Still consider the example event stream in Figure 3(c). Assume the out-of-order event,  $b8$  and  $d2$  arrive after  $f17$ . The dots above the axis show the position where  $b8$  should have arrived under the event timestamp order. If evaluating correctly, one candidate event sequence, “ $a7\ b8\ d10$ ”, should be produced after receiving  $b8$ . However, by simply appending  $b8$  to stack  $S2$ ,  $b8$  will be placed under  $b11$ , with the RIP field set to  $a7$  (Figure 4(a)). The event sequence “ $a7\ b8\ d10$ ” thus would never be constructed. Similarly, by simply appending  $d2$  to stack  $S3$ ,  $d2$ ’s RIP will be pointing to the newly appended  $b8$  (Figure 4(b)). Thus incorrect sequences such as “ $a3\ b6\ d2$ ” and “ $a3\ b11\ d2$ ” will be produced in the sequence construction.

From the above example we observe that such event misplacement potentially causes the SSC operator to miss event sequences and to produce incorrect event sequences.



(a) Incorrect AIS Update when Out-of-Order Event  $b8$  Arrives (b) Incorrect AIS Update when Out-of-Order Event  $d2$  Arrives

Figure 4. Event Misplacement in AIS

### 3.3 Problem for Purge at SSC

A basic mechanism for window constraint-based AIS checking is to compare the difference between the checked event and the latest event received by the system. According to the sliding window semantics, any matching event sequence “ $e_1\ e_2\ \dots\ e_m$ ” for event pattern  $SEQ(E_1,$

$E_2, \dots, E_m)$  must satisfy the time-based constraint that  $(e_m.timestamp - e_1.timestamp) < W$ . For any event instance  $e_i$  kept in AIS, it can be purged from the stack once an event  $e_k$  with  $(e_k.timestamp - e_i.timestamp) > W$  is received by the query engine. However, with out-of-order data arrivals, the above window constraint-based AIS purge is no longer “safe”.

**Example 4.** In Figure 3(c), the out-of-order event  $b8$  comes after  $f17$ . The out-of-order  $b8$  should be put together with  $a3$  and  $d10$  to form a candidate event sequence output (“ $a3\ b8\ d10$ ”) during the sequence construction. However by the above AIS purging,  $a3$  would have already been removed.

Suppose the problem from Section 3.2.1 and 3.2.2 are solved. Retrieving an out-of-order event might then trigger the construction of a new candidate event sequence, such as “ $a3\ b6\ d8$ ” in Example 4. We refer to such event sequences which consist of some out-of-order event, like the sequence “ $a3\ b6\ d8$ ”, as *out-of-order event sequence*. Out-of-order data arrival triggers the construction of out-of-order event sequences. We can see from the above example that PSSC purges some events from AIS which might be needed for forming such out-of-order event sequences in the future. We refer to this as *unauthorized AIS purge*. It prevents some out-of-order event sequences from being constructed by the SSC operator. For example, “ $a3\ b6\ d8$ ” can never be constructed due to the AIS purging on  $a3$  or  $b6$ . Intuitively we can see that once out-of-order data arrival is possible, any data purge at AIS becomes “unsafe”, as expressed by the claim below.

**Claim 1:** Any data purge of active instance stack (AIS) is unauthorized unless the total order on the data arrival holds for the input stream.

*Proof.* For any event instance kept by  $e_i$  in AIS, suppose that it is purged at some moment during the evaluation, and let’s assume the event received right before  $e_i$  is purged is  $e_k$ . There can be out-of-order events  $e_1, e_2, e_3, \dots, e_i, e_{i+1}, e_{i+2}, \dots, e_m$  received after  $e_k$ , with  $e_1.timestamp < e_2.timestamp < \dots < e_{i-1}.timestamp < e_i.timestamp < e_{i+1}.timestamp < \dots < e_m.timestamp$  and  $(e_m.timestamp - e_1.timestamp) < W$ . Thus,  $e_i$  can be used to form a future potential out-of-order event sequence “ $e_1\ e_2\ \dots\ e_i\ \dots\ e_m$ ”. Hence the purge on  $e_i$  is an unauthorized AIS purge.

### 3.4 Summary

Above we have discussed the SSC operator and its state purge function causing the *missing sequences* and *producing incorrect sequences*, as shown in Figure 5 from (1) to (2), corresponding to the SSC operator and the PSSC function described in Section 3.2 and 3.3. In Figure 5, the query plan of the following event sequence query is given as “ $EVENT\ SEQ(E_1, E_2, \dots, E_m)\ WITHIN\ W$ ”. We can see that the problems are all related to the in-memory data structures (AIS) at SSC. Assuming that precise query result is required, evaluation approach in Section 2 is no longer sufficient once out-of-order data arrival is possible.

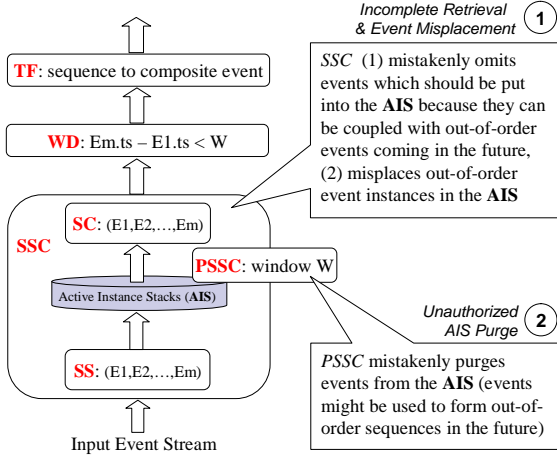


Figure 5. Problem Observation

## 4 SOLUTION

In this section, we propose a solution framework to handle out-of-order data arrival in sequence query evaluation.

### 4.1 Assumption on Un-Ordered SSC Output

Construction of the out-of-order event sequence actually is delayed by its out-of-order event components. Suppose  $a0$  and  $d2$  in *Example 2* both were to arrive in order. Then the sequence “ $a0 b1 d2$ ” would have been constructed before “ $a3 b6 d10$ ”. Assuming the execution of SSC produces output event sequences whenever new sequences are being formed, with out-of-order data arrival, the output order of the SSC can no longer be guaranteed.

If ordered output is needed from the SSC operator, additional semantic information such as K-Slack factor or punctuation is needed to “unblocked” the on-hold candidate sequences from being output by the SSC operator. Since the input event stream to the query engine is unordered, it is reasonable to produce unordered output events to downstreams. Thus in this work, we permit unordered sequence output at the SSC operator.

### 4.2 Solution for SSC

SSC operator consists of three major procedures: (1) event retrieval, (2) AIS construction and (3) event sequence production, with the first two affected by out-of-order data arrival as our previous discussion in Section 3.2. The following is our proposed mechanism for event retrieval and AIS construction.

**Event Retrieval Mechanism.** To avoid incomplete retrieval, all states of the NFA need to be set active before the retrieval over the event stream. Let’s look at *Example 2*. With all the automaton states activated at the beginning,  $b1$  can be retrieved by the automaton even though no A events have appeared before it.

**AIS Construction Mechanism.** For avoiding event misplacement, we have to insert the retrieved events into the right position of AIS. In the case of total order, any new received event can be simply appended to the end of the sequence. We refer to this as the “*append semantics*”. When events can arrive out of order, the “*sort semantics*” need to be applied: for each event instance that triggers a transition in NFA, instead of simply appending it to the stack, we search for a proper insertion place in the corresponding stack to guarantee that the event instances in the same stack are in chronological order from bottom to top. Also, the context pointer (RIP) of the inserted event  $e_i$  needs to be correctly set. Besides that, if  $e_i$  is not the rightmost event type in the sequence query, RIP of the event instances in the right-adjacent stack might need to be updated as well. If the timestamp of  $e_i$  is in between of an event  $e_k$  in the right-adjacent stack and the event pointed by  $e_k$ ’s RIP field,  $e_k$ ’s RIP field will need to be reset to  $e_i$ .

**Example 5.** Similarly to *Example 3*, let’s again consider the event stream in Figure 3(c) with out-of-order event  $b8$  arriving after event  $f17$ . Once  $b8$  is received, it is inserted between  $b6$  and  $b11$  in stack S2. Event  $b8$  is not of a final state event type in the sequence query. Thus we need to check the D instances in stack S3 to see whether any of their RIP field needs to be reset. Since  $b8$  becomes the most recent event in stack S2 whose timestamp is smaller than the timestamp of  $d10$ , the RIP field of  $d10$  should be reset from the original  $b6$  to  $b8$ .

Once a new event  $e_i$  is retrieved, it might trigger the construction of event sequences in SSC. By the total order assumption, only events from the rightmost event pattern in the sequence query (D events in *Example 1*) trigger the event sequence construction in SSC. However, with out-of-order data arrival, any located event might trigger sequence construction in SSC. If the event retrieval and AIS construction are correctly handled as above, the SSC operator needs to produce out-of-order event sequences whenever some new opportunity arises. For instance, two out-of-order event sequences - “ $a3 b8 d10$ ” and “ $a3 b8 d15$ ” - should be constructed by SSC after  $b8$  is inserted into the stack S2 in *Example 5*. Generally, the proposed process for the SSC operator which handles out-of-order data arrival is shown by the below *Algorithm 1*.

---

#### Algorithm 1 Out-of-Order Handling Incorporated SSC

---

**Input:**

- (1) Sequence Query “*EVENT SEQ*( $E_1, E_2, \dots, E_m$ ) *WITHIN*  $W$ ”;
- (2) AIS constructed from previously input events;
- (3) Newly received event  $e_i$  (under event type  $E_i$ )

**Output:**

- (1) Updated AIS;
- (2) Sequence output of SSC

**if** event type  $E_i$  is among  $E_1, E_2, \dots, E_m$  **then**

insert  $e_i$  into stack  $S_i$  (using “*sort semantics*”)

set  $e_i$ ’s RIP

check RIPs of the instances in  $S_{i+1}$  and reset the ones being affected by  $e_i$

produce event sequences containing  $e_i$  if any

**end if**

---

**Optimization.** Lines 2 and 3 in *Algorithm 1* add a newly located event into AIS by applying the “sort semantics” and then sets its RIP field. Line 4 checks the RIP field of the event instance in the right-adjacent stack and resets the ones being affected by the newly located event. However, if the received event is “in-time”, we will continue to follow the previous “append semantics”: that is we simply put the event at the end of the corresponding stack and set its RIP as the most recent event in the left-adjacent stack. Line 4 is no longer necessary for such in-time events. Besides that, sequence construction at Line 5 of *Algorithm 1* would only be triggered when the received event type is at the rightmost in the query sequence (events of type D in the above example).

To avoid such overhead caused by treating every event as a “potential” out-of-order event, the SSC operator can maintain an “AIS-CLOCK” value, which equals to the largest timestamp of events at AIS. *Algorithm 2* shows the optimized approach. Once a newly retrieved event is with a timestamp larger than the current AIS-CLOCK, AIS-CLOCK will be updated to this value. Such an event can be handled simply by the “append semantics” and corresponding steps for in-order events (Lines 3 to 7 in *Algorithm 2*). Whenever a newly retrieved event is with a timestamp smaller than the AIS-CLOCK, we instead apply “sort semantics” and conduct the corresponding out-of-order-specific steps (Lines 9 to 13 in *Algorithm 2*).

---

**Algorithm 2** Out-of-Order Handling Incorporated SSC with AIS-CLOCK

---

```

Input / Output:
Same as Algorithm 1
if event type  $E_i$  is among  $E_1, E_2, \dots, E_m$  then
  if  $e_i.timestamp \geq AIS-CLOCK$  then
    buffer  $e_i$ 
    insert  $e_i$  into stack  $S_i$  (using ‘sort semantics’)
    set  $e_i$ ’s RIP
    check the RIP field of the instances in stack  $S_{i+1}$ 
    & reset the ones being affected
    produce event sequences containing  $e_i$  if any
  else
    buffer  $e_i$ 
    insert  $e_i$  into stack  $S_i$  (using ‘append semantics’)
    set  $e_i$ ’s RIP
  if  $E_i = E_m$  then
    produce event sequences containing  $e_i$  if any
  end if
end if
end if

```

---

### 4.3 Solution for PSSC

When out-of-order data arrival is possible, based on *Claim 1*, no event instance in AIS can be purged safely by the PSSC. To avoid errors, no data purge can ever be applied on AIS. That is not a realistic solution due to its unbounded memory requirement.

Thus, for “unblocking” the PSSC, we need additional semantic knowledge on the stream source to enable the safe data purge on AIS. K-Slack is a well-known approach [6, 5, 3] for processing unordered data streams. In real applications, the K-Slack assumption holds in many sit-

uations when predictions about network delay can be considered. Besides that, it is very suitable for producing approximate answers if that is acceptable. Thus, we now propose our solution for data purging at SSC using the K-Slack semantics.

Here K-Slack is based on time units. It means that the out-of-ordering in event arrivals is within a range of K time units. That is, an event can be delayed for at most K time units. For example, in Figure 3(a), the out-of-order event  $c9$  is received after  $f16$ . Thus it is delayed for 7 ( $16 - 9 = 7$ ) time units. If we set the K value as 5, the out-of-order data arrival case in Figure 3(a) would never arise.

Window purge using K-Slack compares the distance between the checked event and the latest event received at the system. A CLOCK value which equals to the largest time-stamp seen so far for the received events is maintained. Each time the CLOCK value is updated, PSSC will be notified. According to the sliding window semantics, for any event instance  $e_i$  kept in AIS, it can be purged from the stack if  $(e_i.timestamp + W) < CLOCK$ . Thus, under the out-of-order assumption, the above condition will be  $(e_i.timestamp + W + K) < CLOCK$ . This is because after waiting for K time units, no out-of-order event with time-stamp less than  $(e_i + W)$  can arrive. Thus  $e_i$  can no longer contribute to forming a new candidate sequence.

SSC passes the updated CLOCK values up to the PSSC whenever a new event with a larger timestamp is seen. Thus, before Line 1 in *Algorithm 2*, we trigger PSSC by adding the following:

```

IF  $e_i.timestamp > CLOCK$ 
   $CLOCK = e_i.timestamp$ 
  pass a CLOCK triggering to PSSC

```

*Algorithm 3* depicts the basic approach for AIS purging incorporated into the out-of-order event handling by applying the K-Slack constraint. Each time the CLOCK is updated, PSSC gets triggered. Event instances in AIS will be purged when the previously introduced purge condition is satisfied.

---

**Algorithm 3** Out-of-Order Handling Incorporated SSC Operator State Purge

---

```

Input:
(1) Current AIS;
(2) CLOCK triggering from SSC
Output:
updated AIS
On receiving a CLOCK triggering for event instance  $e$  in AIS
if  $e.timestamp + W + K < CLOCK$  then
  purge  $e$ 
end if

```

---

**Example 6.** Let’s consider purge when evaluating sequence query  $SEQ(A, B, D)$  on the data in Figure 2(c). Event instance  $a3$ ,  $b6$  and  $d10$  are kept in AIS after  $d10$  is received. Event  $d10$ ’s RIP points to event instance  $b6$  and  $b6$ ’s RIP points to  $a3$ . Suppose event  $f21$  (which is not shown in the figure) is received after  $f16$  and the window size W equals to 7. Assume K value equal to 2 for the K-Slack constraint.

As more data is received, the CLOCK value increases and the order of those three event instances being purged from AIS is  $a3$  (due to  $13 > 3 + 7 + 2$ , when  $c13$  is met),  $b6$  (when  $f16$  is met) and then  $d10$  (when  $f21$  is met).

Holding the outdated event sequences in the AIS structure increases the workload of the SSC operator for event sequence construction. Take *Example 6* for instance. For data arrival under the total order assumption, when  $b15$  is received, both  $a3$  and  $b6$  can be purged from AIS (due to  $3 + 7 < 15$  and  $6 + 7 < 15$ ). So, there are only three instances in AIS at this moment:  $a7$  in stack S1,  $b11$  in S2 and  $d15$  in S3. Thus, by receiving  $d15$ , SSC operator produces one new event sequence output (“ $a7 b11 d15$ ”). In the out-of-order scenario, SSC might produce more sequence output than in the in-order case. In *Example 6*, assume the K value of K-Slack constraint is 10. When  $d15$  is met, event instances  $a3$ ,  $b6$  and  $d10$  are still kept in AIS. Thus, by receiving  $d15$ , the SSC operator produces three event sequences: “ $a3 b6 d15$ ”, “ $a3 b11 d15$ ” and “ $a6 b11 d15$ ”. The first two sequences actually should not be produced. This is because  $a3$ ,  $b6$  are both “outdated” event instances. They are held in AIS just for out-of-order event sequence construction once possible out-of-order events coming in the future. Thus, coupling the in-order event  $d15$  with the outdated events  $a3$  and  $b6$  is not necessary. An event sequence produced by such construction can never be a result sequence because they would be removed later by window-based filtering (functionality of the WD operator). Thus, it also brings burden to the window-based filtering computation. Many of the outdated event instances may be kept in the AIS stacks if the K value is large. Thus the above overhead on sequence construction and AIS filtering should be considered. Below we propose an optimization technique for decreasing such cost.

**Optimization.** We divide each stack in AIS into two parts: outdated event instances and up-to-date event instances. A divider is set for each stack: instances on or above it are outdated instances and instances below it are up-to-date ones. For a stack without outdated events, the divider is set to NULL. Besides applying the K-Slack-based purge in *Algorithm 3*, the basic data purge introduced in Section 2.2.2 is also applied. The divider for each stack will be set using such basic purge. While an in-order event triggers sequence construction in SSC (Line 12 and Line 13 in *Algorithm 2*), only the events under the divider in each stack will be considered.

Again let’s look at *Example 6* with a K value equal to 10 and window W equal to 7. When  $d15$  is met, the divider of stack S1 is set to  $a3$  and the divider of stack S2 is set to  $b5$ . Thus, only one new sequence (“ $a7 b11 d15$ ”) will be constructed when the in-order event  $d15$  is received. Construction of event sequences “ $a3 b11 d15$ ” and “ $a6 b11 d15$ ” is avoided by applying the AIS partition.

## 5 EXPERIMENTAL EVALUATION

We have implemented our proposed techniques in a prototype system using Java 1.4. We also implemented an

event sequence generator for simulating sequences under different properties. Experiments are run on two Pentium4 3.0Ghz machines each with 512M RAM. The percentage of the out-of-order events and the K-Slack factor are set in the generator. In our experiments, one machine generates and sends the event stream to the second machine, i.e., the query engine. Below we study our experimental result of our proposed techniques.

### 5.1 Sequence Scan and Construction

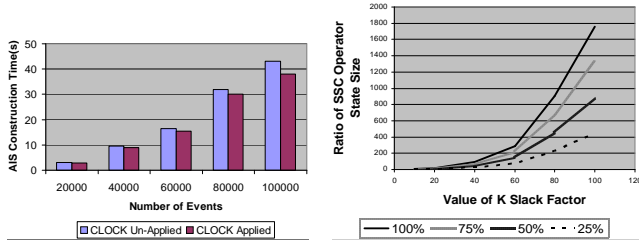
Figure 6 shows the CPU gain when applying the AIS-CLOCK technique introduced in Section 4.2. A sequence query of length 6 (i.e.,  $SEQ(A, B, C, D, E, F)$ ) is run on five different data sets, with the size ranging from 20000 to 100000. The percentage of out-of-order events is 90% in all datasets. Y axis shows the accumulated cost on runtime AIS construction (inserting events and resetting RIP) during the query evaluation. We observe that applying AIS-CLOCK can reduce the overall cost of AIS construction even though the percentage of in-order data is very small. For a decreased percentage of out-of-order data, the performance gain in CPU cost increases. Take the dataset with 80000 events as example. The gain of applying AIS-CLOCK jumps from 8% to 43% if the out-of-order percentage is decreased from 90% to 30%.

### 5.2 Purge at SSC

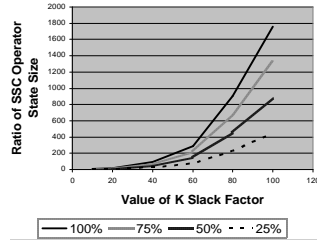
We now study the performance of applying AIS partition during the SSC purge. A sequence query of length 6 is run and the window size is set as 20. Performance gain on memory is shown in Figure 7 and in CPU cost is shown in Figure 8. Through partitioning AIS, construction of outdated event sequences will be avoided for the in-order portion of the input stream. We observe that either a larger percentage of “in-order” events or a larger value slack factor result in more memory and CPU gain by applying AIS partition. Studying quantitatively, the ratio of intermediate buffer size of SSC is directly proportional to the value of  $((K + W)/W)^S$ , where K is the value of the slack factor, W is the window size and S is the length of the query sequence. Due to the space limitation, further discussion is skipped in the paper.

### 5.3 Overhead of Out-of-Order Handling

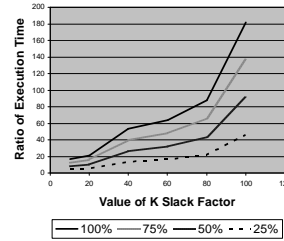
We now test the overhead of our out-of-order event stream processing techniques. We utilize the same setting in Section 5.1 but the out-of-order data percentage is set as 0%. In other words, all the input events are “in-order”. Thus, evaluation based on the total order assumption can be applied in this scenario. The simple approach based on total order assumption and the out-of-order incorporated approach are compared in Figure 9. The performance difference (execution time denoted on the Y axis) is then the overhead of applying the out-of-order handling. Proposed techniques of AIS-CLOCK and AIS partition are both applied in the out-of-order incorporated approach. The overhead ranges from



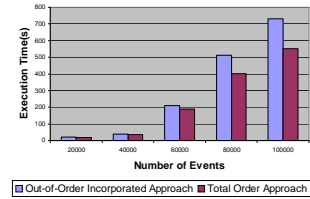
**Figure 6. Applying AIS-CLOCK**



**Figure 7. Applying AIS Partition (1)**



**Figure 8. Applying AIS Partition (2)**



**Figure 9. Technique Overhead**

5.1% to 24.6% in the five given datasets. The overhead increases while increasing the dataset size due to the cost of extra timestamp checking and AIS maintenance.

## 6 RELATED WORK

Most stream query processing research over the past few years has assumed complete ordering of input data [6, 4, 8]. They tend to work with homogeneous streams (timestamped relations), meaning, each stream contains only tuples of the same type. Thus the semantics of general stream processing which employs SQL-like queries composed of join, select, project, aggregation, is not that sensitive to the ordering of the data. Ordering is core for the sequence pattern matching queries we are targeting here.

However, there has also been some initial work of investigating the out-of-order problem for generic (homogenous-input) stream systems. One model, which we adopt for our work, introduces the notion of K-Slack [6]. Such solution is trivial in regular stream system as in fact the processing such as join proceeds as normal (with a K-delayed purging), and any tuple that arrives after K is simply discarded [5]. A native approach [3] on handling out-of-order event stream is using K-Slack as a priori bound on the out-of-orderedness of the input streams. It buffers incoming events in the input queue until ordering can be guaranteed. Compared with our proposed approach where each operator is order sensitive, such process requires additional space and introduces more latency before allowing events being evaluated.

A second solution proposed to handle out-of-order data arrival is applying punctuations, namely, assertions inserted directly in the data stream confirming that for instance a certain value or time stamp will no longer appear in the future input streams [4, 8]. Such techniques, while interesting, require for some service to first be creating and appropriately inserting such assertions - hence here we do not consider this further. Instead this remains our future work.

Lastly, we base our solution on the SASE [10] architecture which has been designed specifically for processing sequence queries over event streams. SASE proposes query language and algebra to support queries on sequential streams, which we adopt as the foundation of our work. However, [10] doesn't support out-of-order data arrival.

## 7 CONCLUSION AND FUTURE WORK

In this work, we address the problem of processing sequence queries over event streams with out-of-order data arrival: (1) we analyze the problems that state-of-the-art event stream processing technology would experience when faced with out-of-order data arrival; (2) we propose new implementation and optimization strategies for the core stream algebra operators such as sequence scan and construction as well as the associated state purging methods; (3) we conduct an experimental study that demonstrates the effectiveness of our proposed approach over existing solutions. Exploring alternative approaches for SSC state purge and handling negative patterns in the query over out-of-order event sequence are our future steps.

**Acknowledgements.** We thank Song Wang and Denis V. Golovnya for many valuable comments.

## References

- [1] A. Adi and O. Etzion. Amit - the situation manager. *VLDB J.*, 13(2):177–203, 2004.
- [2] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, 1994.
- [3] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.
- [4] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.
- [5] J.-H. H. etc. High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790, 2005.
- [6] S. B. etc. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.
- [7] S. C. etc. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [8] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.
- [9] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *ICDE*, pages 450–458, 1996.
- [10] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.