Sequence pattern query processing over out-of-order event streams

Sequence Pattern Query Processing over Out-of-Order Event Streams

Mo Liu[†], Ming Li[†], Denis Golovnya[†], Elke A. Rundensteiner[†], Kajal Claypool[‡]

[†] *Dept. of Computer Science, Worcester Polytechnic Institute, Worcester, Massachusetts, USA*
[‡] *Lincoln Labs, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA*
(liumo|minglee|denis|rundenst)@wpi.edu, claypool@ll.mit.edu

***Abstract*— Complex event processing has become increasingly important in modern applications, ranging from RFID tracking for supply chain management to real-time intrusion detection. A key aspect of complex event processing is to extract patterns from event streams to make informed decisions in real-time. However, network latencies and machine failures may cause events to arrive out-of-order at the event processing engine. State-of-the-art event stream processing technology experiences significant challenges when faced with out-of-order data arrival including output blocking, huge system latencies, memory resource overflow, and incorrect result generation. To address these problems, we propose two alternate solutions: aggressive and conservative strategies respectively to process sequence pattern queries on out-of-order event streams. The aggressive strategy produces maximal output under the optimistic assumption that out-of-order event arrival is rare. In contrast, to tackle the unexpected occurrence of an out-of-order event and with it any premature erroneous result generation, appropriate error compensation methods are designed for the aggressive strategy. The conservative method works under the assumption that out-of-order data may be common, and thus produces output only when its correctness can be guaranteed. A partial order guarantee (POG) model is proposed under which such correctness can be guaranteed. For robustness under spiky workloads, both strategies are supplemented with persistent storage support and customized access policies. Our experimental study evaluates the robustness of each method, and compares their respective scope of applicability with state-of-art methods.**

I. Introduction

Radio frequency identification (RFID) technology has over the years been widely adopted as technology for monitoring and tracking artifacts. A networked RFID system typically is comprised of three layers. One, identity tags attached to items provide a unique identifier for each item and electronically transmit that identifier. Two, networked RFID readers typically have two subsystems, *reader* and *event generator*. The reader collects signals from multiple tags at high rates (100s per second) and pre-processes the data to eliminate duplicates, redundancies, and bogus data from misreads. The event generator then transmits an event stream of RFID reads or aggregated RFID reads to the event processing system (EPS) via either its built-in RFID transmission capabilities or with additional transmitters over wired or Wi-Fi networks [1]. Lastly, monitoring and tracking applications collate event streams from multiple distributed RFID readers to enable a wide range of services from transportation payments, enterprise chain management to product tracking and management.

This pervasive use of RFIDs has fueled the need for better solutions for the processing of streaming events generated from these RFID tags. The real-time processing in time order of event streams generated from distributed RFID readers is a primary challenge for today's monitoring and tracking applications. To outline some of these challenges, consider a networked RFID system where RFID reader R_1 transmits its events to the event processing system *EPS* over a Wi-Fi network, while reader R_2 transmits over a wireless network, and reader R_3 transmits its events over a local area network. The variance in the network latencies, from milliseconds in wired LANs to 100s of seconds for a congested Wi-Fi network, often cause events to arrive out-of-sync with the order in which they were tracked by the RFID readers. Furthermore, machine or partial network failure or intermediate services such as filters, routers, or translators may introduce additional delays. Intermediate query processing servers also may introduce disorder [2], e.g., when a window is defined on an attribute other than the natural ordering attribute [3], or due to data prioritization [4]. This variance in the arrival of events makes it imperative that the EPS can deal with both in-order as well as out-of-order arrivals efficiently and in real-time.

Several solutions for processing event sequence queries that track RFID tags across an ordered set of RFID readers have been proposed in the recent literature [5], [6], [7]. Such customized EPS have been shown to be superior to generic stream processing solutions [8]. However, these solutions mostly assume homogeneity for the underlying RFID reader network and hence provide solutions for processing only in-order arrival of events from distributed RFID readers.

Out-of-order arrival of events[1], when not handled correctly, can result in significant issues as illustrated by the motivating example below. Let us consider a popular application for tracking books in a bookstore [6] where RFID tags are attached to each book and RFID readers are placed at strategic locations throughout the store, such as book shelves, checkout counters and the store exit. The path of the book from the book shelf to store exit can be tracked as it passes the different RFID readers, and the events generated from the RFID readers can be analyzed to detect theft. For example, if a book shelf and a store exit register the RFID tag for a book, but the RFID tag is not read at any of the checkout counters prior to the store exit, then a natural conclusion may be that the book is being shoplifted. Such a query can be expressed by the pattern query *SEQ(S, !C, E)* which aims to find sequences of types *SHELF-READING (S)* and *EXIT-READING (E)* with no events of type *COUNTER-READING (C)* between them. If events of type C (negative query components) arrive out-of-order, we cannot

[1]If an event instance never arrives at our system, our model assumes that it never actually happened. Event detection and transmission reliability in a network are not the focus of our paper. Instead please refer to [9].

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.95

ever output any results if we want to assure correctness of results. This holds true even if the query has an associated window. So no shoplifting will be detected. Also, operators cannot purge any event instances which may match with future out-of-order event instances. In the example above, no events of types *SHELF-READING(S)*, *COUNTER-READING(C)* and *EXIT-READING(E)* can be purged. This causes unbounded stateful operators which are impractical for processing long-running and infinite data streams. Customized mechanisms are needed for event sequence query evaluation to tackle these problems caused by out-of-order streams.

The only available method for dealing with out-of-order arrival of events, called *K-slack* [10], buffers the arriving data for *K* time units. A sort operator is applied on the *K-unit* buffered input as a pre-cursor to in-order processing of events. The biggest drawback of K-slack is rigidity of the *K* that cannot adapt to the variance in the network latencies that exists in a heterogenous RFID reader network. For example, one reasonable setting of K may be the maximum of the average latencies in the network. However, as the average latencies change, *K* may become either too large, thereby buffering unneeded data and introducing unnecessary inefficiencies and delays for the processing, or too small, thereby becoming inadequate for handling the out-of-order processing of the arriving events and resulting in inaccurate results.

To address the above shortcomings, we propose two strategies positioned on the two ends of the spectrum where out-of-order events are the norm on one end and the exception in the other. In contrast to K-slack type solutions [5], [11], our proposed solutions can process out-of-order tuples as they arrive without being forced to first sort them into a globally "correct" order. The conservative method designed for the scenario where out-of-order events are the norm exploits runtime streaming metadata in the form of partial order guarantee (*POG*) thereby permitting the use of unbounded stateful operators and maximally unblocking operators. Memory is effectively utilized to maintain potentially useful data. The aggressive solution designed to handle mostly in-order events outputs sequence results immediately without waiting for any potentially out-of-order events. For the unexpected scenario that out-of-order events do arise, a compensation technique is utilized to correct any erroneous results. This targets applications that require up-to-date results even at the risk of temporally imperfect results to assure delayed correctness. Contributions of this work include:

- Analysis of the issues when state-of-the-art event stream processing technology needs to handle out-of-order data.
- Classification of levels of correctness for out-of-order processing considering latency, output order, result correctness and result completeness.
- Two solution frameworks: the aggressive strategy when out of order event arrival is an exception and the conservative strategy when out of order is prevalent. The former outputs sequence results immediately without waiting for out-of-order events, but guarantees only delayed correctness. The later exploits partial order guarantees to produce permanently correct results.
- Persistent storage support for the proposed strategies to allow for customized access policies and to handle huge windows and spiky streams.
- An event processing prototype that includes the proposed strategies and the *K-slack* strategy. We demonstrate the relative scope of effectiveness of the proposed approaches compared to one another and also the state-of-art method *K-slack* to aid applications with selection of the most appropriate method.

In Section II we give background on event sequence processing. Problems caused by out-of-order data arrival are analyzed in Section III. In Section IV we define the correctness classification lattice. Sections V, VI and VII discuss the *K-slack*, conservative and aggressive strategies respectively. Section VIII introduced disk-based extensions. Our experimental analysis is given in Section IX, while related work is discussed in Section X. Section XI concludes this work.

II. Preliminaries

A. Event Streams and Pattern Queries

Event Instances and Types. An event instance is an instantaneous occurrence of interest at a point in time. It can be a primitive or a composite event [12]. We use lower-case letters (e.g., a, b, c) for event instances. Each event instance has two time-stamps, an occurrence and an arrival timestamp, both assigned from a discrete time domain. The occurrence timestamp $a.ts$ reflects the time when the event was generated while the arrival timestamp $a.ats$ reflects the time when the event was received by the system.

Similar event instances can be grouped into an *event type*. Event types are distinguished by event type names. We use capitalized letters (e.g., A, B, C) for event types. An event type *A* has an associated event schema that defines a set of attributes A_1, ..., A_n. We use $a \in A$ to indicate an instance a is of event type *A*.

Event Streams. The input to the query system is a potentially infinite event stream that contains all events of interest [6], [12]. The event stream is heterogeneous, being populated with event instances of different event types. For example, in the RFID-based retail scenario [6], all RFID readings are merged into a single stream sorted by their timestamps. The stream contains *SHELF-READING*, *COUNTER-READING* and *EXIT-READING* events.

Out-of-Order Event. Consider an event stream S: e_1, e_2, ..., e_n, where e_1.ats $<$ e_2.ats $<$ ... $<$ e_n.ats. For any two events e_i and e_j ($1 \leq i, j \leq n$) from S if $e_i.ts < e_j.ts$ and $e_i.ats < e_j.ats$, we say the stream is an *ordered event stream*. If however e_j.ts $<$ e_i.ts and e_j.ats $>$ e_i.ats, then e_j is flagged as an *out-of-order event*. Stream *S* in Figure 3(a) lists events in their arrival order, thus event c_9 received after d_{17} is an out-of-order event.

Pattern Queries. We focus on sequential pattern queries, a core feature for most event processing systems [5]. The theft query in Section I is an example of such a query. Pattern queries specify how individual events are filtered and multiple events are correlated via time-based and value-based constraints. For illustration purposes we utilize a query language syntax similar to SASE [6]. However our solution is general and compatible with existing algebraic-based event processing systems [5], [6].

In the EVENT clause, SEQ specifies a particular order in which the events of interest must occur. We say an event e_i is a positive (resp. negative) event if there is no '!' (resp. with

<Query>::=EVENT <event pattern>
[WHERE <qualification>]
[WITHIN <window>]
<event pattern>::=SEQ((E$_i$, (E$_i$|!E$_i$)$^+$)|((E$_i$|!E$_i$)$^+$, E$_i$)) (1≤ i ≤ n))
<window>::= Window length W

'!') symbol used before its respective event type in the SEQ construct. Positive events appear in the final query result while negative events do not. The WHERE clause corresponds to a boolean combination using logical connectives $\wedge$ and $\vee$ of predicates that use one of the six comparison operators (=, $\neq$, $>$, $<$, $\leq$, $\geq$). The WITHIN clause checks if the temporal difference between the first and last events within a result tuple is less than the specified window length W (time range). Our above shoplifting query is specified as:

```
EVENT   SEQ(SHELF s, !(CHECKOUT c), EXIT e)
WHERE   s.id = c.id  AND  c.id = e.id
WITHIN  5 hs
```

B. Event Stream Logical Algebra

A query expressed by the above language is translated into a query plan composed of the following operators: Window Sequence (*WinSeq*), Window Negation (*WinNeg*), and Selection (*Sel*) [6]. The *WinSeq* operator denoted *WinSeq* (E_1, E_2, ..., E_n, $window$) extracts all matches to the positive event pattern specified in the query and constructs positive event sequences. *WinSeq* also checks whether all matched event sequences occur within the specified sliding window. The *WinNeg* operator specified by *WinNeg*(!E_1, ($time\ constraint$); ...; !E_m, ($time\ constraint$)) checks whether no negative events as specified in the query exist within the indicated time constraint in a matched positive event sequence. The *Sel* operator, expressed as *Sel(P)*, where *P* denotes a set of predicates on event attributes, further filters event sequences by applying the predicates specified in the query. The qualification in the WHERE clause provides the parameters for the *Sel* operator.

Figure 1 shows an example algebra plan for the pattern query *Q*, with predicates in the *Sel* operator pushed down into the *WinSeq* and *WinNeg* operators as appropriate. [6] discusses several methods for placing *Sel* within the query plan, including pushing *Sel* inside sequence operators.

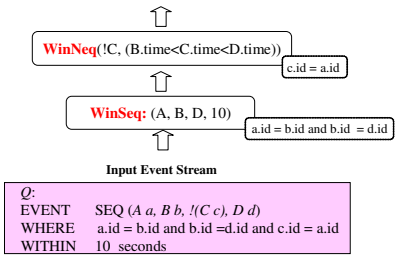

Fig. 1. One Possible Algebraic Query Plan

C. Event Stream Physical Algebra

Window Sequence Operator (WinSeq) *WinSeq* as the bottom-most operator employs a non-deterministic finite automaton (NFA) for pattern retrieval [6]. We push Window

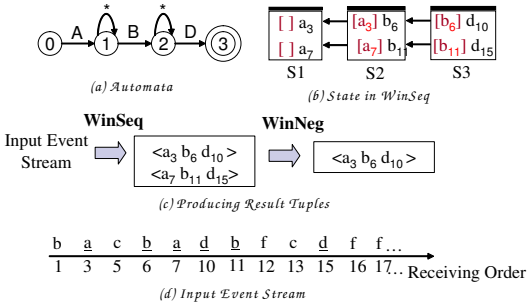

Fig. 2. Example Query Evaluation Steps

filtering into *WinSeq*. Let N denote the number of positive event types. Then the number of states in the NFA equals *N+1* (including the starting state). A data structure named *SeqState* associates a stack with each state of the NFA storing the events that trigger the NFA transition to this state (Figure 2(a)). For each instance e_i in the stack, an extra field named *PreEve* records the nearest instance in terms of time sequence in the stack of the previous state to facilitate sequence result construction. Figure 2(d) shows a partial input event stream. All retrieved events of type *A*, *B* and *D* are extracted by *WinSeq* and kept in *SeqState*. Figure 2(b) shows the stacks of *SeqState* after receiving the given portion of stream *S*. In each stack, its instances are naturally sorted from top to bottom in the order of their arrival timestamps e_i.ats. The most recent instance in stack *S1* of type *A* before b_{11} is a_7. The *PreEve* field of b_{11} is set to a_7, as shown in the parenthesis preceding b_{11}.

WinSeq has three core functionalities as stated below:

Insert. With the assumption that events come in order, each received positive event instance is simply appended to the end of the corresponding stack and its *PreEve* field is set to the last event in the previous stack. Each received negative event instance is passed to *WinNeg*.

Compute. When the newly inserted event e_n is an instance of the final stack then *WinSeq* computes sequence results. With *SeqState*, the construction is simply done by a depth first search in the DAG that is rooted at this instance e_n and contains all the virtual edges reachable from this root. Each root-to-leaf path in the DAG corresponds to one matched event sequence to be returned. This event sequence $<e_1, e_2, ..., e_n>$ for the query *SEQ*($E_1, E_2, ..., E_n$) guarantees that (1) e_i is an event instance of type E_i, $\forall$ i (1$\leqslant$ i $\leqslant$n) and (2) the sequence ordering constraint e_1.ts $\leqslant$ e_2.ts $\leqslant$...$\leqslant$ e_n.ts. After receiving the events in Figure 2(d), *WinSeq* outputs the two event sequences in Figure 2(c).

Purge. Purge of the *WinSeq* state removes all outdated events from *SeqState* based on window constraints. Any event instance e_i kept in *SeqState* can be safely purged from the top of its stack once an event e_k with ($e_k.ts - e_i.ts$) $>$ *W* is received by the query engine.

Window Negation Operator (WinNeg) A data structure similar to *SeqState*, called *NegState*, is utilized for the *WinNeg* operator. *NegState* associates a stack with each negative event type. *WinNeg* has the following three functionalities.

Insert. Events of the negative types are appended to the corresponding *NegState*, e.g., the events $c_i \in C$ for the above example are inserted into the *NegState* of type *C*.

Compute. For each intermediate event sequence sent from

WinSeq, *WinNeg* checks whether any negative instances corresponding to the query exist. The event sequence will not be output if any exist. In the above example, when d_{15} is received there are two *C* events in NegState (namely, c_5 and c_{13}. The second intermediate result sequence $<a_7, b_{11}, d_{15}>$ constructed by *WinSeq* will be removed by *WinNeg* because a negative *C* event (namely, c_{13}) exists between b_{11} and d_{15}.

Purge. Window-based purging proceeds as for *SeqState*.

III. PROBLEMS CAUSED BY OUT-OF-ORDER DATA ARRIVAL

A. Problems for WinSeq Operator

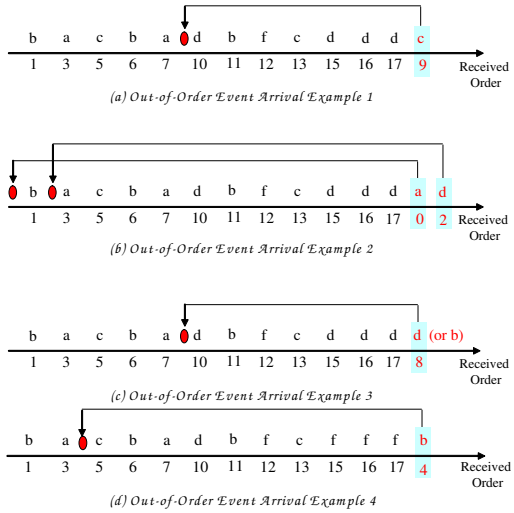

Fig. 3. Out-of-Order Event Arrival Example

Current event stream processing systems [6], [13] rely on purging of the *WinSeq* operator to efficiently and correctly handle in-order event arrivals. An event instance e_i is purged when it falls out of the window W, i.e., when a new event instance e_k with e_k.ts - e_i.ts $>$ W is received. This purging is considered "safe" when all events arrive in-order. However, with out-of-order event arrivals such a "safe" purge of events is no longer possible. Consider that an out-of-order event instance e_j (e_j.ts $<$ e_k.ts) arrives after e_k. In this scenario, if e_k is purged before the arrival of e_j, potential result sequences wherein e_j is matched with some event e_k are lost.

While this loss of results can be countered by not purging *WinSeq* state, in practice this is not feasible as it results in storing infinite state for the *WinSeq* operator.

Example 1: For the stream in Figure 3(c), suppose the out-of-order event d_8 arrives after d_{17} ($d_8.ats > d_{17}.ats$), d_8 should form a sequence output $<a_3, b_6, d_8>$ with a_3 and b_6. However *WinSeq* state purging would have already removed a_3 thus destroying the possibility for this result generation.

Observation 1: A purge of the *WinSeq* state (*SeqState*) is "unsafe" for out-of-order event arrivals resulting in loss of results. Not applying purge to *SeqState* results in unbounded memory usage for the *WinSeq* operator.

B. Problems for WinNeg Operator

With out-of-order data arrival, window-based purge of *NegState* is also not "safe", because it may cause the generation of wrong results. A negative event instance e_i will be purged once an event e_k with $(e_k.ts - e_i.ts) > W$ is received. When an out-of-order *positive* event instance e_j ($e_j.ts < e_k$.ts) arrives after the purge of a negative event instance e_i, this may cause the *WinSeq* operator to generate some incorrect sequence results that should have been filtered out by the negative instance e_i. Similarly, an out-of-order *negative* event instance e_i may be responsible for filtering out some sequence results generated by *WinSeq* previously. In short, this *negation state purge* is unsafe, because it may cause unqualified out-of-order event sequences to not be filtered out by *WinNeg*.

Example 2: For the stream in Figure 3(d), assume out-of-order event instance b_4 comes after f_{17}. Suppose *WinSeq* sends up the out-of-order sequence $<a_3, b_4, d_{10}>$ to *WinNeg*. *WinNeg* should determine that $<a_3, b_4, d_{10}>$ is not a qualified sequence because of the negative event c_5 between b_6 and d_8. However, if *NegState* purge would already have removed c_5, then this sequence would now wrongly be output.

Observation 2. We observe the dilemma that on the one hand purging is essential to assure that the state size of *NegState* does not grow unboundedly. On the other hand, any purge on *NegState* is unsafe for out-of-order input event streams because wrong sequence results may be generated.

Observation 3. *WinNeg* can never safely output any sequence results for out-of-order input streams, because future out-of-order negative events may render any earlier result incorrect. Hence, *WinNeg* is a *blocking operator* causing the queries to never produce any results.

IV. LEVELS OF CORRECTNESS

We define criteria of output "correctness" for event sequence processing.

Ordered output. The *ordered output* property holds if and only if for any sequence result $t = <e_1, e_2, ..., e_n>$ from the system, we can guarantee that for every future sequence result $t' = <e_1', e_2', ..., e_n'>$, $e_n.ts \leqslant e_n'.ts$. We refer to sequence results that don't satisfy the property as *out-of-order output*.

Immediate output. The *immediate* property holds if and only if every sequence result will be output as soon as it can be determined that no current negative event instance filters it out.

Permanently Valid. The property *permanently valid* holds if and only if at any given time point t_{cur}, all output result sequences from the system so far satisfy the query semantics given full knowledge of the complete input sequence. That is, for any sequence result $t = <e_1, e_2, ..., e_n>$, it should satisfy (1) the sequence constraint $e_1.ts \leq e_2.ts \leq e_3.ts ... \leq e_n.ts$; (2) the window constraint (if any) as $e_n.ts$ - $e_1.ts \leq W$; (3) the predicate constraints (if any) and (4) the restriction on the negation filtering (if there is a negative type E_{neg} between positive event type E_i and E_j then no current or future received event instance e_{neg} of type E_{neg} satisfies $e_i.ts \leq e_{neg}.ts \leq e_j.ts$).

Eventually Valid. We define eventually valid property to be weaker than *permanently valid*. At any time t_{cur}, all output results meet conditions (1) to (3) from above. Condition (4) is relaxed as follows: if in the query between event type E_i and E_j there is a negation pattern E_{neg} then (4.1') no e_{neg} of

type E_{neg} exists in the current *NegState* with $e_i.ts \leq e_{neg}.ts \leq e_j.ts$ and (4.2') if in the future e_{neg} of type E_{neg} with e_{neg}.ats $> t_{cur}$ satisfies $e_i.ts \leq e_{neg}.ts \leq e_j.ts$, then results involving e_i and e_j become invalid.

The *permanently* and *eventually valid* defined above are two different forms of *valid* result output.

Complete output. If at time t_{cur} a sequence result *t* = <e_1, e_2, ..., e_n> is known to satisfy the query semantics defined in (1) to (4) in the *permanently valid* category above or those defined in the *eventually valid* category then the sequence result *t* = <e_1, e_2, ..., e_n> will also be output at time t_{cur} by the system.

Based on this categorization, we now define several notions of output correctness. Some combination of these categories can never arise. For example, it is not possible that an execution strategy produces permanently correct un-ordered results immediately. The reason is that with out-of-order event arrivals, if sequence results are output immediately then they cannot be guaranteed to remain correct in the future. Similarly, it is not possible that output tuples produced are only eventually correct and at the same time are in order. The reason is that we cannot assure that sequences sent by some later compensation computation do not lead to out-of-order output. Also, it is not possible that out-of-order tuples can be output in order yet immediately. The reason is that out-of-order event arrivals can lead to out-of-order output. We now introduce four combinations as levels of output correctness that query execution can satisfy:

- **Full Correctness**: ordered, immediate output, permanently valid and complete output.
- **Delayed Correctness**: ordered, permanently valid and eventually complete output.
- **Delayed Unsorted Correctness**: unordered, permanently valid, and complete output.
- **Convergent Unsorted Correctness**: immediate output, eventually valid and complete output.

Although *full correctness* is a nice output property, it is too strong a requirement and unnecessary in most practical scenarios. In fact, if events come out-of-order, *full correctness* cannot be achieved and we must live with delayed correctness.

In some applications *delayed unsorted correctness* may be equally accepted as strict delayed but ordered correctness. Sequence results may correspond to independent activities in most scenarios and the ordering of different outputs is thus typically not important. For instance, if book1 or book2 was stolen first is not critical to a theft detection application. Sorting the sequence results will cause increased even possibly prohibitively large response time. *Delayed Unsorted Correctness* is thus a practical requirement. For example, in the RFID-based medicine transportation scenario, between the medicine cabinet and usage in the hospital, the medical tools cannot pass any area exposed to heat nor can they be near any unsanitary location. In this scenario, correctness is of utmost importance while some delay can be tolerated.

On the other hand, in applications where correctness is not as important as system response time, then the *convergence unsorted correctness* may be a more appropriate category. The detection of shoplifting of a high price RFID tagged jewelry would require a quick response instead of a guaranteed valid one. Actions can be taken to confront the suspected thief and in the worst case, an apology can be given later if a false alarm is confirmed. In the rest of the paper, we design a solution for each of the identified categories.

V. Naive Approach: K-slack

K-slack is a well-known approach for processing unordered data streams [10]. We now classify *K-slack approach* into the *delayed correctness* category. As described in the introduction, the *K-slack* assumption holds in situations when predictions about network delay can be reliably assessed. Large *K* as required to assure correction will add significant latency. We briefly review K-slack which can be applied for situations when the strict *K-slack* assumption indeed holds. Our slack factor is based on time units, which means the maximum out of orderness in event arrivals is guaranteed to be *K* time units. With *K* so defined, proper ordering can be achieved by buffering events in an input queue until they are at least *K* time units old before allowing them to be dequeued. We set up a clock value which equals the largest occurrence timestamp seen so far for the received events. A dequeue operation is blocked until the smallest occurrence timestamp *ts* of any event in the buffer is less than *c* - *K*, where *c* is the clock value.

The functionalities of *WinSeq* and *WinNeq* in the *K-slack* solution are the same as those in the ordered input case (Section II-C) because data from the input buffer would only be passed in sorted order to the actual query system.

VI. Conservative Query Evaluation

A. Overview of Partial Order Guarantee Model

We now propose a solution, called conservative query evaluation, for the category of *delayed unsorted correctness*. The general idea is to use meta-knowledge to safely purge *WinSeq* and *WinNeg* states and to unblock *WinNeg* (addressing the problems in Section III). Permanent valid is achieved because results are only reported when they are known to be final. Relative small memory consumption is achieved by employing purging as early as possible.

To safely purge data, we need meta-knowledge that gives us some guarantee about the nonoccurrence of future out-of-order data. A general method for meta-knowledge in streaming is to interleave dynamic constraints into the data streams, sometimes called punctuation [14].

Partial Order Guarantee Definition. Here we now propose special time-oriented metadata, which we call *Partial Order Guarantee (POG)*. *POGs* guarantee the future non-occurrence of a specified event type. *POG* has associated a special metadata schema POG = <*type*, *ts*, *ats*> where *type* is an event type E_i, *ts* is an occurrence timestamp and *ats* is an arrival timestamp. POG p_j indicates that no more event e_i of type p_j.*type* with an occurrence timestamp e_i.*ts* less than p_j.*ts* will come in the stream after p_j, i.e., (e_i.*ats* > p_j.*ats* implies e_i.*ts* > p_j.*ts*).

Many possibilities for generating *POGs* exist, ranging from source or sensor intelligence, knowledge of access order such as an index, to knowledge of stream or application semantics [15]. In fact, it is easy to see that due to the monotonicity of the time domain, such assertions about time stamps tend to be more realistic to establish compared to guarantees about the nonoccurrence of certain content values throughout the remainder of the possibly infinite stream. We note that network protocols can for instance facilitate generation of this

knowledge about timestamp occurrence. Note that the TCP/IP network protocol guarantees in-order arrival of packets from a single host. Further, TCP/IP's handshake will acknowledge that certain events have indeed been received by the receiver based upon which we then can safely release the next *POG* into the stream. Henceforth, we assume a logical operator, called punctuate operator [15], that embeds *POGs* placed at each stream source. Also the punctuate operator can be placed in the appropriate servers to synchronize the POGs of the same event type generated from each source, thus guaranteeing validity of POGs across different TCP/IP connections in a network.

Using *POGs* is a simple and extremely flexible mechanism. If network latency were to fluctuate over time, this can naturally be captured by adjusting the *POG* generation without requiring any change of the query engine. Also, the query engine design can be agnostic to particularities of the domain or the environment. While it is conceivable that *POGs* themselves can arrive out-of-order, a punctuate operator could conservatively determine when *POGs* are released into the stream based on acknowledged receival of the events in question. Hence, in practice, out-of-order *POG* may be delayed but would not arrive prematurely. Clearly, such delay or even complete loss of a *POG* would not cause any errors (such as incorrect purge of the operator state), rather it would in the worst case cause increased output latency. Fortunately, no wrong results will be generated because the *WinNeg* operator would simply keep blocking until the subsequent *POG* arrives.

B. POG-Based Solution for WinSeq

POGSeq State. We add an array called *POGSeq State* to store the *POGs* received so far with one array position for each positive event type in the query. For each event type, we store the largest timestamp which is sufficient due to our assumption of *POG* ordering (see Section VI-A).

Tuple Processing

Insert. In-order events are inserted as before. The simple append semantics is no longer applicable for the insertion of out-of-order positive event instances into the state. Instead out-of-order event $e_i \in E_i$ will be placed into the corresponding stack of type E_i in *SeqState* sorted by occurrence timestamp. The *PreEve* field of the event instance e_k in the adjacent stack with e_k.ts $> e_i$.ts will be adjusted to e_i if ($e_k.PreEve$).ts is less than e_i.ts.

Compute. In-order event insertion triggers computation as usual. The insertion of an out-of-order positive event e_i triggers an out-of-order sequence computation. This is done by a backward and forward depth first search in the DAG. The forward search is rooted at this instance e_i and contains all the virtual edges reachable from e_i. The backward search is rooted at event instances of the accepting state and contains paths leading to and thus containing the event e_i. One final root-to-leaf path containing the new e_i corresponds to one matched event sequence. If e_i belongs to the accepting (resp. starting) state, the computation is done by a backward (resp. forward) search only.

Purge. Tuple processing will not cause any state purging.

POGs Processing

Purge. The arrival of a POG p_k on a positive event type triggers the safe purge of the *WinSeq State*, as explained below.

Insert. If *WinSeq* receives a POG p_k on a positive event type, we update the corresponding *POGSeq* state POGSeq[i] := p_k.ts if p_k.ts is greater than the current POG time for p_k.type. If the positive event type is listed just before one negative event type in a query, we pass p_k to *WinNeg*. If *WinSeq* receives a POG p_k on a negative event type, we also pass p_k to *WinNeg*.

Definition 1: A positive event e_i is *purge-able* henceforth no valid sequence result $<e_1$, ..., e_i, ..., $e_n>$ involving e_i can be formed.

POG-Triggered Purge. Upon arrival of a POG p_k, we need to determine whether some event e_i with e_i.type $\neq p_k$.type can be purged by p_k. By Definition 1, we can purge e_i if it can't be combined with either current active events or potential out-of-order future events of type p_k.type to form valid sequence results.

Algorithm 1 Singleton-POG-Purge

Input: (1) Event $e_i \in E_i$ (2) $p_k \in POG$
Output: Boolean (indicating whether event e_i was purged by p_k

```
1  if (p_k.ts<e_i.ts) || (p_k.type==e_i.type)
2  then return false;
3  else
4    if (E_k = p_k.type listed after E_i in query Q)
5      if (e_i.ts is within [p_k.ts - W, p_k.ts])
6      then return false;
7      else
8        if (current events of type p_k.type exist
9          within [ei.ts, ei.ts + W] in WinSeq)
10         then return false;
11         else purge event e_i; return true; endif endif
12   else // E_k is listed before E_i in query Q
13     if (no events of p_k.type exist within [e_i.ts - W, e_i.ts] in WinSeq)
14     then purge event e_i ∈ E_i; return true;
15     else return false; endif endif
16 endif
```

Algorithm 1 depicts the purge logic for handling out-of-order events using POG semantics. In lines 1 and 2, we cannot purge e_i because an event instance e_k of p_k.type with e_k.ts $> p_k$.ts can still be combined with e_i to form results. In lines 4, 5 and 6, we cannot purge e_i if e_i.*ts* is within [p_k.*ts* - W, p_k.*ts*] for e_i could be composed with an event instance e_k of p_k.type with occurrence timestamp e_k.ts $> p_k$.ts and e_k.ats $> p_k$.ats. In lines 8, 9, 10, we cannot purge e_i for even though p_k can guarantee no out-of-order events of type p_k.type can be combined with e_i. Some current event instance e_k can still be combined with e_i. To understand Algorithm 1, let us look at the following example.

Example 3: Consider purging when evaluating sequence query SEQ(A, B, !C, D) within 7 mins on the data in Figure 3(b). Assume after receiving events a_0 and d_2 (both shaded), we receive a POG p_k = <A, 1> indicating that no more events of type A with timestamp less than or equal to 1 will occur. For there are no events of type A before b_1 in window W, we can safely purge b_1.

Optimized POG-Triggered Purge. By examining only one POG p_k at a time, Algorithm 1 can guarantee an event e_i can be purged successfully if no event instance e_k of type p_k.type (e_i. type $\neq p_k$.type) exists within window W. However, even though events of different POG types exist, they may not satisfy the sequence constraint as specified in one query. We need to make use of the knowledge provided by a set of $POGs$ as together they may prevent construction of sequence results.

In Algorithm 2 from line 1 to 7, we check whether e_i can form results with event instances of type listed before E_i in Query Q. We update the *checking* value once we find an instance of p_k.type. We need to continue the instance search after timestamp *checking* for the next type in the *POGSeq state*. The checking order guarantees the sequential ordering constraint among existing event instances of POG types. Similarly from line 8 to 15, the algorithm checks whether e_i can form results with event instances of type listed after E_i in Query Q. Example 4 illustrates this.

Example 4: Given the data in Figure 3(d), let's consider purging a_7 for query SEQ(B, A, B, D, F) within 10 mins. Assume after receiving b_4, we receive two $POGs$ (p_1 = <B,17>, p_2 = <D,17>). b_6 of type B exists before a_7. b_{11} of type B exists after a_7. However, no existing event instances of type D exist in the time interval [11, 7+10]. Due to p_2, we know no future events of type D will fall into [11, 7+10]. So a_7 is purge-able.

Algorithm 2 POG-Set-Purge

Query *Q*: "SEQ(E_1, E_2 ,...., E_n) within W";
Input: Event $e_i \in E_i$
Output: Boolean (whether e_i was purged by the existing POG Set.)

```
1 int checking = e_i.ts - W;
2 for (each POG p_k in POGSeq that p_k.type is before e_i.type in Q)
3  if (p_k.ts > e_i.ts)
4     if (no current event e_k of p_k.type in [checking, e_i.ts])
5     then purge event e_i ∈ E_i; return true;
6     else checking = min(e_k.ts); endif endif
7 endfor
8 checking = e_i.ts;
9 for (each POG p_k in POGSeq that p_k.type is after e_i.type in Q)
10  if (p_k.ts ≥ e_i.ts + W)
11     if (no event e_k of type p_k.type in [checking, e_i.ts + W])
12     then purge event e_i ∈ E_i; return true;
13     else checking = min(e_k.ts); endif endif
14 endfor
15 return false
```

C. POG-Based Solution for WinNeg

POGNeg State. An in-memory array called *POGNeg State* is used to store $POGs$ of negative event types sent to *WinNeg*. The length of $POGNeg$ corresponds to the number of negative event types in the query. For each negative event type, we only store one POG with its largest timestamp so far. POGNeg$[i]$:= p_k.ts if p_k.ts is greater than the current POG time for p_k.type.

Holding Set. A set named *holding set* is maintained in *WinNeg* to keep the candidate event sequences which cannot yet be safely output by *WinNeg*.

Tuple Processing Additional functionalities beyond *WinNeg* as introduced in Section II-C are:

Insert. If *WinNeg* receives output sequence results from *WinSeq*, it stores them in the holding set. If *WinNeg* receives a negative event, *WinNeg* stores it in the negative stack.

Compute. When *WinNeg* receives sequence results, after the computation as introduced in Section II-C, *WinNeg* will put candidate results in the *holding set*. When *WinNeg* receives an out-of-order negative event, the negative event will remove some candidate results from the holding set per the query semantics. No results are directly output in either case.

POGs Processing

Insert. Once *WinNeg* receives a POG p_k on a negative (resp. positive) event type, it updates the POGNeg$[i]$ = p_k.ts.

Compute. Let us assume the sequence query *SEQ(E_1, E_2, ..., E_i, !NE, E_j, ..., E_n)* where *NE* is a negation event type. When we receive a *POG* p_k = <*NE, ts*>, an event sequence "$e1$, $e2$ ..., e_i, e_j, ... e_n" maintained in *WinNeg* can be output from the holding set if $e_j.ts < p_k.ts$.

Now assume the negation type is at an end point of the query such as SEQ(E_1, E_2,..., E_n, !NE). Then any output sequence <e_1, e_2, e_3, ..., e_n> from *WinSeq* will be put into the holding set of *WinNeg* if no NE event exists in *NegState* with a time stamp within the range of [e_n.ts, e_1.ts + *W*]. When we receive a *POG* p_k = <NE, ts> which satisfies p_k.ts > $e_1.ts$ + *W*, this sequence can be safely output by *WinNeg*.

Example 5: Given query SEQ(A, B, !C, D) and the data in Figure 3(c), when d_{10} is seen, *WinSeq* produces <a_3, b_6, d_{10}> as output and sends it up to *WinNeg*. At this moment, the *NegState* of *WinNeg* holds the event instance c_5. c_5.ts is not in the range of [6,10]. However *WinNeg* cannot output this tuple because potential out-of-order events may still arrive later. Assume after receiving event d_{17}, we then receive POG p_i = <C,10>. So future out-of-order events of type C, if any, will never have a timestamp less than 10. *WinNeg* can thus safely output sequence result <a_3, b_6, d_{10}>.

Purging. For the negative events kept in the *WinNeg* state, Algorithms 1 can be utilized to safely purge *WinNeg*.

For illustration purposes, we discussed the processing of one negative event in the query. Algorithms can be naturally extended to also handle queries with more than one negation pattern.

VII. Aggressive Query Evaluation

A. Overview

We now propose the aggressive method to achieve *convergent unsorted correctness* category. The goal is to send out results with as small latency as possible based on the assumption that most data arrives in time and in order. In the case when out-of-order data arrival occurs, we provide a mechanism to correct the results that have already been erroneously output. Two requirements arise. One, traditionally streams are append-only [16], [17], [18], [19], meaning that data cannot be updated once it is placed on a stream. A traditional append-only event model is no longer adequate. So a new model must be designed. Two, to enable correction at any time, we need access to historical operator states until safe purging is possible. The upper bounds of *K-slack* could be used for periodic safe purging of the states of *WinSeq* and *WinNeg* operators when event instances are out of Window size + K. This ensures that data is kept so that any prior computation can be re-computed from its original input as long as still needed. Further, *WinSeq* and *WinNeg* operators must be equipped to produce and consume compensation tuples. Given that any new event affects a limited subset of the output sequence results, we minimize run-time overhead and message proliferation by generating only new results. That is, we generate delta revisions rather than regenerating entire results.

We extend the common append-only stream model to support the correction of prior released data on a stream. Two kinds of stream messages are used: **Insertion tuple <+, t>** is

induced by an out-of-order positive event, where "t" is a new sequence result. **Deletion tuple <-, t>** is induced by an out-of-order negative event, such that "t" consists of the previously processed sequence. Deletion tuples cancel sequence results produced before which are invalidated by the appearance of an out-of-order negative event. Applications can thus distinguish between the types of tuples they receive.

B. Compensation-Based Solution for WinSeq

Insert. Same as the POG-based *WinSeq* Insert function.

Compute. In-order event insertion triggers computation as usual. If a positive out-of-order event e_i is received, e_i will trigger the construction of sequence results in *WinSeq* that contain the positive event. The computation is the same as the Compute function introduced in Section VI-B. If a negative out-of-order event e_i is received, the negative event will trigger the construction of spurious sequence results in *WinSeq* that have the occurrence of the negative instance between the constituent positive instances as specified in a query. These spurious sequence results will be sent up to the *WinNeg* operator followed by the negative event e_i. See Algorithm 3 for details.

Example 6: The query is SEQ(A, !C, B) within 10 mins. For the stream in Figure 3(a), when an out-of-order negative event c_9 is received, new spurious sequence results $<a_3, b_{11}>$, $<a_7, b_{11}>$ are constructed in *WinSeq* for a_3.ts $< c_9$.ts $< b_{11}$.ts and a_7.ts$<c_9$.ts$<b_{11}$.ts and sent to *WinNeg*.

Purge. If some maximal arrival delay *K* is known, then any event instance e_i kept in *SeqState* is safely purged once an event e_k with (e_k.ts - e_i.ts) $>$ window *W + K* is received.

Algorithm 3 Out-of-order Processing in WinSeq

```
Query "EVENT SEQ(E1, E2, ..., Ei, !Ej, Ek, .., En)" within W
Input: Out-of-order Event et
Output: Results, Negative events

1 if (et.type==Ej)
2 then
3    WinSeq generates spurious results <e1, e2, ..., ei, ek, ..., en>
4    with ei.ts < et.ts < ek.ts and (en.ts - e1.ts ≤ W)
5    and sends them to WinNeg along with ei
7 else  //et.type ≠ Ej
8    <+, e1, e2, ..., et, ..., en> with (en.ts - e1.ts ≤ W)
9    is constructed by WinSeq and sent to WinNeg
10 endif
```

C. Compensation-Based Solution for WinNeg

Insert. When candidate results or negative instances are received, *WinNeg* will insert them as usual.

Compute. If the *WinNeg* operator receives spurious results from the *WinSeq* operator, *WinNeg* first checks whether these spurious results would have been invalidated by the negative event instances already in *WinNeg* before. If not, the *WinNeg* operator will send out these spurious results as compensation tuples of the deletion type.

Purge. Same as compensation-based *WinSeq* Purge.

Example 7: As in Example 6, $<a_3, b_{11}>$ and $<a_7, b_{11}>$ are sent to *WinNeg* as marked spurious results. (a_3, b_{11}) was filtered by c_5 in *WinNeg* for a_3.ts$<c_5$.ts$<b_{11}$.ts. So only $<a_7, b_{11}>$ is sent out as compensation tuple $<$-, a_7, $b_{11}>$.

Algorithm 4 Out-of-order Processing in WinNeg

```
Query "EVENT SEQ(E1, E2, ..., Ei, !Ej, Ek, .., En)" within W
Input: 1 Results sent from WinSeq; 2 Out-of-Order Negative Event et
Output: Compensation tuple

1 if marked spurious results are received from WinSeq
2 boolean output = true;
3   for each <e1, e2, ..., ei, ek, ..., en> sent from WinSeq
4     for each ej ∈ Ej stored in WinNeg
5       if(ei.ts < ej.ts < ek.ts)
6         then output = false; break; endif
7     endfor
8     if output == true
9     then <-, e1, e2, ..., ei, ek, ..., en> is output.
10    endif
11    output = true; endfor
12 endif
13 if results are regular (not marked spurious)
14 then
15 boolean output = true;
16   for each <e1, e2, ..., ei, ek, ..., en> or <+, e1, e2, ..., ei,
17        ek, ..., en> sent from WinSeq
18 Compute in WinNeg (Section II-C) endfor
27 endif
28 Insert et into the negative stack.
```

VIII. Disk-Based Extensions

Thus far we have assumed that sufficient memory was available. However, large window sizes or bursty event streams might cause memory resource shortage during query processing. In such rare cases, we would employ a disk spilling strategy, where a block of oldest memory-resident event instances is chosen as victim and flushed to disk when the memory utilization passes a set threshold. We store historical information at the operator level, that is the states of *WinSeq* and *WinNeg* are stored as frames indexed by time. To avoid context switching, we use two separate buffers. One stores newly incoming events, and the other is dedicated to load temporarily events back from disk for out-of-order handling.

Whenever an event instance e_i arrives out of order, and its event instances within W are stored in disk, then we first need to load the event window frame into *SeqState* and *NegState*. This incurs overhead due to extra I/O costs for bringing the needed slices of the historical event stream into the buffer.

There is a tradeoff between the aggressiveness with which this process is run, and the benefits obtained. To address the tradeoff, we design policies for mode selection. One criteria we consider is the likelihood that many results would be generated by this correction processing. Assuming uniformity of query match selectivities, we use the number of out-of-order events that fall into the same logical window (physical disk page) as indicator of expected result generation productivity. Further, we employ a task priority structure to record the yet to be handled events and the correspondingly required pages.

For each page that is required to be used, we maintain the out-of-order events yet to be processed. We also keep track of the expected execution time for each page. If the total number of required times for one page is greater than the activation threshold α or the expected execution time is greater then some threshold β, we load that page and trigger the execution of tuples in this batch.

TABLE I

PARAMETERS FOR EVENT GENERATION

term	meaning
$NumE$	Number of total events received so far
Num_{Ti}	Number of events of type Ti received so far
$NumT$	Number of event types
Num_{Tio}	Number of out-of-order events of type Ti received
$NumResult$	Number of permanent valid results
MD_{Ti}	Maximum arrival delay of event type Ti
λ_{TiO}	Out-of-order event percentage of event type Ti
λ_{op}	Total Out-of-order positive event percentage
λ_{on}	Total Out-of-order negative event percentage
λ_O	Total Out-of-order event percentage
λ_{POGs}	POGs percentage.

TABLE II

SYMBOLS FOR PERFORMANCE METRIC

term	meaning
$Seqi_{ts}$	The system time when ith sequence is outputted from our system.
$T_{Latency}$	The total sequence result latency
$T_{Execution}$	Total Execution time
α_p	Activation page threshold
α_t	Expected time threshold

IX. EXPERIMENTAL EVALUATION

A. Experimental Setup

We have implemented our proposed techniques in a prototype system using Java 1.4. Experiments are run on two Pentium 4 3.0Ghz machines with 512M RAM. One machine generates and sends the event stream to the second machine, i.e., the query engine. We develop an event generator that can be configured with parameters as listed in Table I. Each source sends out event instances of the same type. By setting the number of sources, we can specify *NumT*. We also set the maximum arrival delay MD_{T_i} and out-of-order percentage λ_{Tio} for each event source. λ_{op} (resp. λ_{on}) is determined by the total number of out-of-order positive (resp. negative) events over the total number of instances received by our system. *POGs* of each event type can be sent out after at most the maximum time units MD_{T_i} safely. λ_{POGs} corresponds to the total number of *POGs* over total number of event instances received by our system. In our experiments, the input event stream contains events of 10 different event types. The data distributes among event types from *A* to *J* evenly.

Table II defines terms used in the performance evaluation. The performance metric *average application latency* is the average time difference between the sequence output time and the maximum arrival time of event instances composed into the sequence result. It can be measured by $\Sigma(Seqi_{ts} - \max(e_i.ats))/NumResult$. In *WinSeq*, if the result is triggered by an in-order event, i.e., the last event type in the query pattern, then the maximum event arrival time $\max(e_i.ats)$ is the system time when that in-order event e_i arrived. If the result is triggered by an out-of-order event, $\max(e_i.ats)$ is the system time when the out-of-order event arrives. We keep track of e_i.ats for two cases: (1) for in-order events, we only maintain e_i.ats if e_i.type is the last type in a query pattern, because other in-order events will not trigger the generation of sequence results and (2) for out-of-order event instances, we keep all their e_i.ats because all out-of-order events will trigger the generation of sequence results. The sequence output time is the system time when *WinNeg* outputs the result.

B. Evaluating Aggressive Strategy

Varying Out-of-Order Event Percentage vs. Average Application Latency. In this first experiment, we examine the impact of out-of-order positive events on the performance of the aggressive strategy. The out-of-order positive event percentage λ_{op} is determined by $\Sigma(Num_{TiO})$*/NumE* with T_i a positive event type. The window size is 20 time units, and the maximum delay of out-of-order events is 10 time units. λ_{op} is varied from 10% to 50% by increasing Num_{TiO} of each positive event type T_i. We also change the length of the sequence queries from 3 to 7 (i.e., from *SEQ(A, B, !C, D)* to *SEQ(A, B, !C, D, E, F, G, H)*).

Figure 4 shows the results. The trend lines correspond to the different queries used in the experiment. Two observations can be made from these results. First, the average latency increases with the increase of the query length (Figure 4). Second, as expected the increase in the percentage of positive out-of-order events does not greatly impact the average application latency for different query lengths. More out-of-order positive events will only increase the event insertion time into the *WinSeq* state which is logarithmic in the number of tuples in the corresponding stack.

We also examine the impact of the out-of-order negative events on average latency of the aggressive strategy. We vary query length, and the out-of-order negative event percentage λ_{on} (measured by Num_{CO} /Num_C is varied from 0% to 50%). In Figure 5, we observe that average application latency increases as the percentage of out-of-order negative event instances increases. This is because more out-of-order negative events of a certain type will incur more re-computation for the generation of compensation tuples. The impact of λ_{on} on longer query lengths is greater than on shorter ones. One reason is that more compensation tuples are generated by *WinSeq*. Also the sequence construction time is longer for queries of longer lengths as compared to shorter ones.

Varying Out-of-Order Event Percentage vs. Compensation Tuple Number. We further conduct an experiment to investigate the effect of increasing the out-of-order negative events on the number of compensation tuples that are generated. We employ sequence queries of length from 3 to 7 (i.e., from *SEQ(A, B, !C, D)* to *SEQ(A, B, !C, D, E, F, G, H)*). The window size is 20 time units. The percentage of out-of-order events is varied for negative from 0% to 50% and for positive is set to 0%. In Figure 6, we observe that with larger out-of-order negative event percentage, the production of compensation tuples, being triggered by out-of-order negative events, will increase. That is out-of-order negative events will cause more compensation tuples to be generated. Also, queries of longer length will have more compensation tuples as compared to a shorter one for the same out-of-order negative event percentage.

Varying Out-of-Order Event Percentage vs. Execution Time. We also test the execution time for the aggressive strategy. Average execution time is measured by $T_{Execution}$*/NumE*,

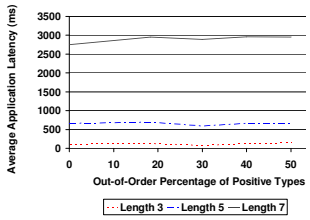

Fig. 4. Pos. Event Percent. vs. Average Latency

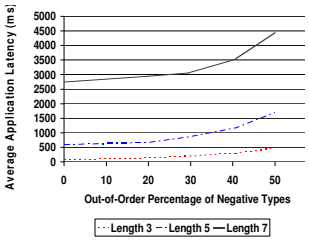

Fig. 5. Neg. Event Percent. vs. Average Latency

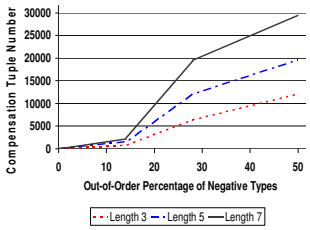

Fig. 6. Compensation Tuple Numbers

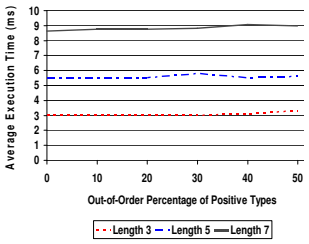

Fig. 7. Positive Event Percentage vs Average Execution Time

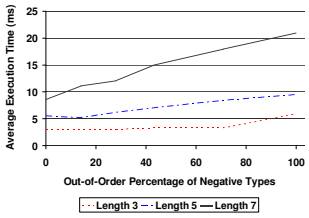

Fig. 8. Negative Event Percentage vs Average Execution Time

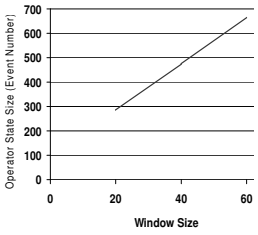

Fig. 9. Memory of Aggressive Strategy

with $T_{Execution}$ the summation of operator execution times. It does not include idle time. In Figures 7 and 8, the x-axis denotes the out-of-order positive and negative event percentage respectively. The y-axis is the average execution time. We observe that queries with longer length have longer average execution time because more sequence results are constructed with data of more types. Also the sequence construction time for a sequence result is also larger as compared with shorter ones. As we increase the out-of-order positive event percentage, the average execution time doesn't change much. But the average execution time increases as the out-of-order negative event percentage increases for more recomputing is needed for compensation tuple generation.

Varying Window vs. Operator State Sizes. Next, we evaluate the memory consumption, with operator state size measured by the average number of event instances stored. We change the window size from 20 to 60 time units. For the maximum delay of event instances is 10 time units, we can purge event instances if they are out of window size plus maximum arrival delay. Figure 9 shows that operator state size is linear in the window size.

C. Evaluating Conservative Strategy

We employ a sequence query of length 7 (i.e., *SEQ(A, B, !C, D, E, F, G, H)*) with a window of size 20 time units. The maximum delay of out-of-order events MD_{Ti} for each event type T_i is set to 10 time units and the out-of-order event percentage λ_O is 30%. *POGs* are generated on all event types, both positive and negative ones, in random order. *POGs* percentage is varied from 10%, 20%, 30% to 40%.

Varying POGs Percentage vs. Operator State Sizes. Figure 10 depicts our results concerning memory (operator state sizes) for different *POGs* percentages. Memory consumption is measured by the average number of event instances stored in our system. We observe that a larger *POGs* percentage results in larger savings in memory as *POGs* help the system to purge event instances in time.

Overhead Evaluation. We employ a sequence query of length 7 with a window of size 20 time units. We compare the average execution time of the conservative strategy with the basic approach (Non-POGs Applied) using an in-order event stream. The time difference is the overhead of the conservative strategy as shown in Figure 11.

Varying Out-of-Order Event Percentage vs. Average Application Latency. We also test the performance change of the conservative strategy as the out-of-order event percentage varies from 10% to 50%. We set *POGs* percentage to 20% as this keeps operator state and the overhead of *POGs* relative small. As shown in Figure 12, average application latency increases as out-of-order event percentage increases because *POGs* will be send out later and sequence results will be kept longer in the *WinNeq* operator. Smaller *POGs* arrival rate will increase the latency.

D. Evaluating K-Slack Strategy

We test behavior of *K-slack* as the out-of-order event percentage λ_O and query length vary. The window size is 20 time units. The maximum arrival delay is 10 time units. As shown in Figure 13, more out-of-order events will incur larger sorting costs in the buffer before events are dequeued. So the average application latency increases. Memory consumption for *K-slack* is close to the aggressive strategy. As shown in Figure 14, the average execution time of K-slack increases as the out-of-order percentage increases. The results confirm our complexity analysis that out-of-order percentage is proportional to application latency.

E. Comparison of Three Strategies

We now compare these methods concerning their average application latency. The maximum arrival delay MD_{Ti} is 10 time units. The *POGs* are uniformly generated on all event types. We conduct a sequence query of length 5. We change the percentage of out-of-order events from 10% to 50% and set *POGs* percentage to 20%.

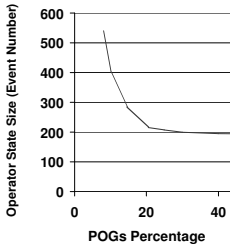

Fig. 10. Conservative Strategy

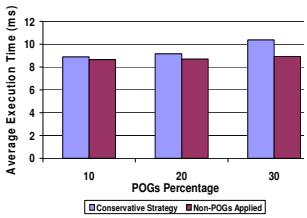

Fig. 11. Overhead of Conservative Strategy

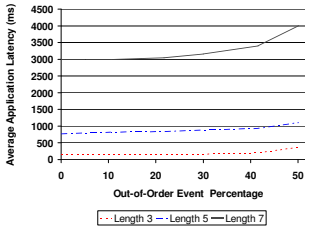

Fig. 12. Avg. Latency of Conservative Strategy

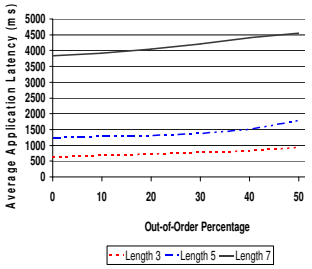

Fig. 13. Avg. Latency of k-slack Strategy

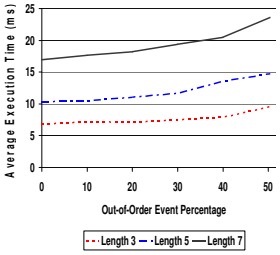

Fig. 14. Avg. Execution Time of k-slack

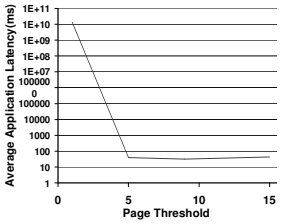

Fig. 15. Disk-Based Extensions with Changing Threshold

Fig. 16. Avg. Latency of all Methods

The three strategies are compared in Figure 16. As we run our system on event streams with different out-of-order event percentages, the *K-slack* strategy has larger latency because every event needs to wait for the maximum delay before processing. When the out-of-order percentage of negative types is relatively low, the aggressive solution wins over the conservative strategy. After a certain threshold (e.g., 40% in this experiment), the aggressive solution yields higher latency for it incurs extra processing time. That is when an out-of-order negative event (i.e., *C* in our example) arrives, the aggressive solution needs to recompute sequence results involving the negative event and send out compensation tuples. The average application latency increases as more compensation tuples need to be generated and it will delay the generation of sequence results. In contrast, the average application latency for the conservative method is primarily determined by *POGs* of the negative type.

F. Evaluating Disk-Based Extensions

In this experiment, the expected time threshold α_t is again set to 10 (Section VIII). Query length is 5, and out-of-order rate is 80%. Figures 15 reports the results for the threshold α_p not greater than 5 and greater than 5 respectively. These figures show that for the batch solution the average application latency decreases greatly until some point. The reason is that IO cost is saved with loading data in batch into our buffer. However, the production of sequence results is delayed due to first having to load data in a batch. So the average application latency begins to increase for $\alpha_p >= 9$ in this experiment.

G. Summary

The conservative and aggressive strategies have different scopes of applicability. When the negative event out-of-order percentage is relatively high, we prefer to use the conservative strategy for it can guarantee permanent correctness with less average sequence result latency. The total latency will be further reduced by greater *POGs* percentages. Compared to the conservative strategy, the aggressive strategy would incur higher latency by more re-computation of compensation tuples. When the negative event out-of-order percentage is relatively low, the aggressive strategy is preferred because applications can get acceptable results with less average sequence result latency. However, if *POGs* are not available (thus conservative method is inapplicable) and applications cannot tolerate temporarily wrong sequence results (making aggressive method unacceptable), then *K-slack* would need to be applied. In this case, for wrong results are intolerable, the K would have to be set extremely high, causing huge latency to avoid erroneous results.

X. RELATED WORK

Most stream query processing research has assumed complete ordering of input data [10], [14], [20]. Thus they tend to work with homogeneous streams (time-stamped relations), meaning each stream contains only tuples of the same type. The semantics of general stream processing which employs set-based SQL-like queries is not sensitive to the ordering of the data. While clearly ordering is core for the sequence matching queries we are targeting here.

There has been some initial work in investigating the out-of-order problem for generic (homogenous-input) stream systems, with the most common model being *K-slack* [10], [18]. *K-slack* assume the data may arrive out-of-order at most by some constant *K* time units (or *K* tuples). Using *K-slacks* for state purge has limitations in practical scenarios as real network latencies tend to have a long-tailed distribution [21]. This means for any *K* value, there exits a probability that the latency can go beyond the threshold in the future (causing erroneous results). Furthermore K-slack has the shortcoming that *WinSeq* state would need to keep events while considering only the worst case scenario (i.e., it must conservatively go with the largest network delay). Our conservative solution could easily

model such K-slack assumption, yet freeing the query system from having to hard-code such knowledge.

[7] proposes a spectrum of consistency levels and performance tradeoffs in response to out-of-order delivery. We borrow their basic ideas for our problem analysis, though their consistency levels are determined by the input stream blocking time in an alignment buffer and state size.

Borealis [22] extends Aurora in numerous ways, including revision processing. They introduce a data model to specify the deletion and replacement of previously delivered results. But their work is not designed for event systems, nor are any concrete algorithms shown for revision processing. They propose to store historical information in connection points. To design efficient customized query processing with out-of-order support, we instead store prior state information at the operator level to assure minimal information as required for compensation processing is maintained. The notion of negative tuples in [17] and revision tuples in Borealis [22] both correspond to models to communicate compensation. Though [17] does not deal with out of order data.

[11] proposes heartbeats to deal with uncoordinated streams. They focus on how heartbeats can be generated when sources themselves do not provide any. Heartbeats are a special kind of punctuation. The heartbeats generation methods proposed in [11] could be covered by our punctuate operator. But how heartbeats can be utilized in out-of-order event stream processing is not discussed.

[14], [15] exploit punctuations to purge join operator state. [20] leverages punctuations to unblock window aggregates in data streams. We propose partial order guarantee (*POG*) based on different namely *occurrence related punctuation* semantics for event stream processing.

Our concept of classification of correctness has some relationships with levels of correctness for warehouse view maintenance categories defined in [23].

Lastly, our work adopts the algebraic query architecture designed for handling sequence queries over event streams [5], [24], [25], [6]. These systems do not focus on the out-of-order data arrival problem.

XI. CONCLUSION AND FUTURE WORK

In this work, we address the problem of processing pattern queries on event streams with out-of-order data arrival. We analyze the problems state-of-the-art event processing technology experiences when faced with out-of-order data arrival including blocking, resource overflow, and incorrect result generation. We propose two complimentary solutions that cover alternative ends of the spectrum from norm to exception for out of orderness. Our experimental study demonstrates the relative scope of effectiveness of our proposed approaches, and also compares them against state-of-art *K-slack* based methods. Most current event processing systems either assume in order data arrivals or employ a simple yet inflexible mechanism (*K-slack*) which as our experiments confirm will induce high latency. Our work is complementary to existing event systems, thus they can employ our proposed conservative or aggressive solutions according to their targeted application preferences.

The amount of out-of-order data may change over time. Different negative event types in a query may experience different frequencies and also time delays of out-of-order data. Ongoing work is to design a hybrid solution positioned between the two ends of the spectrum, where our system seamlessly switches from one level of output correctness to another based on application requirements. We may control this tradeoff by selectively sending out possibly spurious results from a holding buffer without waiting for the punctuation on the negative event types based on observed levels of triggered corrections and estimations of the reliability of different event type behaviors.

Acknowledgements. This work was partly supported by National Science Foundation under grants IIS 0414567, SGER 0633930 and CRI 0551584. We thank Dr. Luping Ding for contributions to the initial idea for the project and Professor Murali Mani for many valuable comments. We are grateful to the CAPE team for creating the stream processing code base.

REFERENCES

[1] J. L. S. P. S. Sandoval-Reyes, "Mobile rfid reader with database wireless synchronization," in *CIE*, 2005, pp. 5–8.
[2] M. A. Hammad and et. al., "Scheduling for shared window joins over data streams." in *VLDB*, 2003, pp. 297–308.
[3] C. D. Cranor and et. al., "Gigascope: A stream database for network applications." in *SIGMOD Conference*, 2003, pp. 647–651.
[4] V. Raman and et. al., "Online dynamic reordering for interactive data processing." in *VLDB*, 1999, pp. 709–720.
[5] A. J. Demers, J. Gehrke, and et. al., "Cayuga: A general purpose event monitoring system." in *CIDR*, 2007, pp. 412–422.
[6] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams." in *SIGMOD Conference*, 2006, pp. 407–418.
[7] R. S. Barga and et. al., "Consistent streaming through time: A vision for event stream processing." in *CIDR*, 2007, pp. 363–374.
[8] S. Chandrasekaran and et. al., "Telegraphcq: Continuous dataflow processing for an uncertain world." in *CIDR*, 2003.
[9] D. J. Abadi and et. al., "Reed: Robust, efficient filtering and event detection in sensor networks," in *VLDB*, 2005, pp. 769–780.
[10] S. Babu and et. al., "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams." *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 545–580, 2004.
[11] U. Srivastava and J. Widom, "Flexible time management in data stream systems." in *PODS*, 2004, pp. 263–274.
[12] S. Chakravarthy and et. al., "Composite events for active databases: Semantics, contexts and detection." in *VLDB*, 1994, pp. 606–617.
[13] A. Ayad and et. al., "Static optimization of conjunctive queries with sliding windows over infinite streams." in *SIGMOD Conference*, 2004, pp. 419–430.
[14] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman, "Joining punctuated streams." in *EDBT*, 2004, pp. 587–604.
[15] P. A. Tucker and et. al., "Exploiting punctuation semantics in continuous data streams." *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 555–568, 2003.
[16] D. B. Terry and et. al., "Continuous queries over append-only databases." in *SIGMOD Conference*, 1992, pp. 321–330.
[17] L. Golab and M. T. Özsu, "Update-pattern-aware modeling and processing of continuous queries." in *SIGMOD Conference*, 2005, pp. 658–669.
[18] D. J. Abadi and et. al., "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, August 2003.
[19] A. Arasu and et. al., "Stream: The stanford stream data manager." *IEEE Data Engineering Bulletin*, vol. 26, no. 1, 2003.
[20] J. Li and et. al., "Semantics and evaluation techniques for window aggregates in data streams." in *SIGMOD Conference*, 2005, pp. 311–322.
[21] W. Willinger and et. al., "Self-similarity and Heavy Tails: Structural Modeling of Network Traffic," *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, 1998.
[22] E. Ryvkina and et. al., "Revision processing in a stream processing engine: A high-level design." in *ICDE*, 2006, p. 141.
[23] Y. Zhuge and et. al., "View maintenance in a warehousing environment," in *SIGMOD Conference, May 22-25, 1995*, 1995, pp. 316–327.
[24] P. Seshadri and et. al., "Sequence query processing." in *SIGMOD Conference*, 1994, pp. 430–441.
[25] M. K. Aguilera, , and et. al., "Matching events in a content-based subscription system." in *PODC*, 1999, pp. 53–61.