Complex Event Pattern Detection over Streams with Interval-Based Temporal Semantics

Ming Li
Silicon Valley Laboratory
IBM Corporation
San Jose CA, USA
mingli3@us.ibm.com

Murali Mani
Dept. of CSEP
University of Michigan-Flint
Flint MI, USA
mmani@umflint.edu

Elke A. Rundensteiner
Dept. of Computer Science
Worcester Polytechnic Institute
Worcester MA, USA
rundenst@cs.wpi.edu

Tao Lin
Research and Development
Amitive
Redwood City CA, USA
taolin2004@gmail.com

ABSTRACT

In this work, we study the event pattern matching mechanism over streams with interval-based temporal semantics. An expressive language to represent the required temporal patterns among streaming interval events is introduced and the corresponding temporal operator ISEQ is designed. For further improving the interval event processing performance, a punctuation-aware stream processing strategy is provided. Experimental studies illustrate that the proposed techniques bring significant performance improvement in both memory and CPU usage with little overhead.

Categories and Subject Descriptors

H.2.4 [**Database Manager**]: Query Processing

General Terms

Algorithms, Design

1. INTRODUCTION

Event stream processing (ESP) [3] [8] [23] [14] [7] [2] [13] [15] has become increasingly important in modern applications, ranging from supply chain management to real-time intrusion detection. Existing ESP systems have focused on detecting temporal patterns from instantaneous events, that is, events with no duration. Under such a model, an event instance can only be happening "*before*", "*after*" or "*at the same time as*" another event instance. However, such sequential patterns are inadequate to express the complex temporal relations in some real-world application domains such as medicine, finance, logistics and meteorology, where events usually have durations and the durations could play important roles.

For example, an ESP system can be monitoring the events generated by the warehouse of a supermarket. Based on the temperature values sent by the readers, temperature fluctuations as event instances captured using interval semantics are generated and sent to the ESP system. We assume three different types of intervals: HIGH, MEDIUM and LOW. Besides that, the duration of an item staying in the warehouse is extracted and sent to the ESP system as an interval event instance after the item leaves the warehouse, which is denoted as STAY. The pattern of a HIGH interval event contains a STAY interval event means that the duration of an item staying in the warehouse is with a high temperature the whole time. We can use such pattern to indicate that the quality of this item might not be good.

Event instances happen instantaneously at a time point are called events with point-based temporal semantics (*point events* in short) and event instances occurs over a time interval are called events with interval-based temporal semantics (*interval events* in short). For an interval event e, we use $e.ts$ and $e.te$ to denote its start time and end time timestamps, which are called the *endpoints* of e.

According to the classification scheme introduced by Allen [4], there are 13 temporal relations between any two interval events: "*before*", "*after*", "*during*", "*contain*", "*meet*", "*met by*", "*overlap*", "*overlapped by*", "*start*", "*started by*", "*finish*", "*finished by*" and "*equal*". *Table 1* shows the detail of all these 13 temporal relations. Some of them are mirror relations. For example, "*overlap*" and "*overlapped by*" are mirror relations since "a overlaps b" can be represented as "b is overlapped by a".

Relations between interval events can be expressed in terms of their endpoints, which is equivalent to the disjunction of relations given in Allen's classification above [22]. Given two interval events, if the order of all four endpoints of these two intervals are fixed, their temporal relation is one from the 13 relations given by [4]. While the order of endpoints are not fully fixed, relation between interval events can be even more flexible. Obviously, temporal relations among events in such interval-based scenario are more sophisticated than the sim-

Temporal Relation	**Temporal Algebra**
e before e'	$(e.\text{te}<e'.\text{ts})$
e after e'	$(e.\text{ts}>e'.\text{te})$
e during e'	$(e.\text{ts}>e'.\text{ts})\wedge(e.\text{te}<e'.\text{te})$
e contain e'	$(e.\text{ts}<e'.\text{ts})\wedge(e.\text{te}>e'.\text{te})$
e meet e'	$(e.\text{te}=e'.\text{ts})$
e met by e'	$(e.\text{ts}=e'.\text{te})$
e overlap e'	$(e.\text{ts}<e'.\text{ts})\wedge(e.\text{te}>e'.\text{ts})\wedge(e.\text{te}<e'.\text{te})$
e overlapped by e'	$(e.\text{ts}>e'.\text{ts})\wedge(e.\text{ts}<e'.\text{te})\wedge(e.\text{te}>e'.\text{te})$
e start e'	$(e.\text{ts}=e'.\text{ts})\wedge(e.\text{te}<e'.\text{te})$
e started by e'	$(e.\text{ts}=e'.\text{ts})\wedge(e.\text{te}>e'.\text{te})$
e finish e'	$(e.\text{ts}>e'.\text{ts})\wedge(e.\text{te}=e'.\text{te})$
e finished by e'	$(e.\text{ts}<e'.\text{ts})\wedge(e.\text{te}=e'.\text{te})$
e equal e'	$(e.\text{ts}=e'.\text{ts})\wedge(e.\text{te}=e'.\text{te})$

Table 1: Temporal Relations between Two Intervals

ple sequential relations in the point-based scenario. Thus, the query semantics and evaluation mechanisms used for detecting temporal patterns over streams with point events are not sufficient for pattern detection over interval streams.

Previous research on pattern detection over event streams mainly focused on extracting temporal patterns from point-based event data. For example, [23] proposes *sequence scan and construction* for implementing the SEQ operator. However it handles the "*before*" / "*after*" temporal relation only on point events. Even though in [3] [8] [7] the events are defined based on the interval model, only the "*before*" / "*after*" is supported. The data mining community studied discovering patterns over interval events [10] [19] [24]. [10] uses a hierarchical representation that extends Allen's interval algebra [4] for modeling complex event patterns over intervals. However, this representation is lossy as the exact relationships among the events cannot be fully recovered. [24] [19] devises a lossless representation to overcome the drawbacks of [10]. Based on their proposed representation, they design corresponding mining algorithms for pattern discovery over interval events. [19] also examines how the discovered temporal patterns can be utilized in classification to differentiate closely related classes thus building an interval-based classifier. However, these works mainly focus on pattern discovering algorithms instead of pattern detection algorithms. Besides that, they do not consider streaming input with window constraints.

Contributions. In this work, we study query processing over event streams with interval-based temporal semantics. The contributions include:

- We investigate an expressive language to represent the required temporal patterns among streaming interval events and design the corresponding temporal operator ISEQ.
- We study the physical implementation of ISEQ, focusing on its event buffering, result construction and state purge operations. Optimization utilizing the input order constraint of the interval events is provided and the ISEQ-incorporated query execution strategy is designed.
- For further improving the evaluation performance, we provide a mechanism to embed the Şinterval begin punctuationŤ (indicating the start of an upstream interval) to the event stream. Based on that, we provide the strategy for punctuation-aware interval event stream processing, which can greatly reduce the runtime memory and CPU footprint.
- We conduct experimental studies to demonstrate the efficiency of our proposed techniques in interval event stream handling.

Roadmap. The rest of the paper is organized as follows. *Section 2* proposes an evaluation mechanism for detecting temporal pattern over event streams with interval-based temporal semantics. Strategy of using punctuation for optimizing the interval stream processing framework is discussed in *Section 3*. Experimental results are presented in *Section 4*. The related work is given in *Section 5*, followed by conclusions in *Section 6*.

2. PATTERN DETECTION OVER INTERVAL EVENT STREAMS

2.1 Interval Event Query Model

An interval event e can be represented as two separated point events using its two endpoints, namely start endpoint and termination endpoint [22] [20]. We assume a data model in which an interval event is an atomic unit semantically. Thus an interval event is fully composed after it is completed. By then the event instance is ready to be sent to the ESP system. As discussed in *Section 1*, we use $e.ts$ and $e.te$ to denote the start time and end time timestamps of an interval event. As a simplified representation, we use a pair of numbers adjacent to e as $e_{t1|t2}$.

In the following discussion we assume that events' timestamps are globally ordered, reflecting the ordered semantics [15] of the physical events. In case of disordered event arrival, the mechanism introduced in [15] can be applied after some further adjustments and it will not affect the correctness of the basic approach introduced in this work. Each event is assigned a timestamp from a discrete ordered time domain. Such timestamps are assigned by a separate mechanism before events enter the event processing system and they reflect the true order of the occurrences of these events. For an ordered interval-based event stream, the event receiving order at the ESP system is the same as the order of the end time of the event instances.

Complex event pattern detection languages are studied in a number of existing works [3] [23] [7] [6]. In this work we adopt the query language defined in [3] [23] with supported set of event composite operators *SEQ*, *AND*, *OR* and *SELECT*, which specify a set of temporal and logical relations among events. For fully supporting event processing over interval streams, the query algebra and evaluation corresponding to detecting temporal relations among events need to be adjusted. The logical operators (i.e., AND / OR), which detect patterns with logical relations, apparently do not need to be adjusted for interval handling since they are independent from the temporal semantics [16] [17]. The pattern selection operator SELECT, which performs value-based predicate checking, needs special adjustment only if it is associated with negation patterns. However that can be simply avoided by pushdown of the SELECT operator [23]. Thus in this work, our main focus is on the event sequence operator SEQ.

A point event can be treated as an interval event with the same start time and end time timestamps. So the SEQ operator designed in [23] [3] for sequential pattern matching handles only the "*before*" / "*after*" temporal relation. Because of the transitive property of the "*before*" / "*after*" relation, this basic two-arguments operator can be extended to handle sequence with three or more event as *SEQ*($E1$, $E2$, $E3$, ...), which indicates that an $E1$ event is followed by an $E2$ and the $E2$ event is followed by an $E3$ event, and so on. For example, *SEQ*(A, B, C) detects a sequential event patterns $<a, b, c>$ where a is an event instance of type A, b is an event instance of type B, c is an event instance of type C, a *before* b and also b *before* c.

As we have pointed out in *Section 1*, besides the simple sequential relations, additional temporal relations can be defined between interval events since that intervals could have overlapping portion. We consider an *interval temporal relation* to be a relation among two or more interval events. Since the relations between interval events can be expressed in terms of relations between the event endpoints, we divide interval temporal relation into two different categories, namely, *closed endpoint relations* and *open endpoint relations*.

An interval temporal relation where the order of all endpoints of the events are fixed is called a *closed endpoint relation*, which falls into the categories given by [4]'s classification. We refer to these temporal relations as *Allen-based relations*. An interval temporal relation where the order of all endpoints of the events are not fully fixed is called an *open endpoint relation*. Real-world applications might have customized requirement on interval pattern detection where open endpoint relations are needed to be defined. For example, we can define temporal relation R as "intervals of type $E1$ starts before intervals of type $E2$". The temporal algebra of such pattern is as $E1.ts < E2.ts$, where temporal relation is only given on the start endpoints. We can see that such temporal relation is the disjunction of several closed endpoint relations. Again take the temporal relation R defined above as an example. It is equivalent to the disjunction of several closed endpoint relations as ($E1$ *before* $E2$) $\vee$ ($E1$ *meets* $E2$) $\vee$ ($E1$ *overlaps* $E2$) $\vee$ ($E1$ *finished by* $E2$) $\vee$ ($E1$ *contains* $E2$).

To express temporal relation between two intervals (referred to as *primitive temporal relation*), the simple SEQ operator becomes insufficient because it only considers "*before*" / "*after*" as temporal relations over point events. One approach to define a primitive temporal relation is simply using the 13 Allen-based relations and their disjunction using the syntax Rel[*list of Allen-based relations*]($E1$, $E2$), where *Rel* is a temporal operator defined by a list of Allen-based relations. For example, Rel[*overlap*](A, B) represents the *overlap* relation in Allen's model. Rel[*before*, *meet*, *overlap*, *finished by*, *contain*](A, B) represents an open endpoint temporal relation which is the disjunction of five different Allen-based relations.

While the expressiveness of such relation representation is no longer sufficient if it is extended to represent temporal relation among three or more intervals (referred to as *composite temporal relation*. as Rel[*list of Allen-based relations*]($E1$, $E2$, $E3$, ..., Em). One reason is that some temporal relation might not satisfy the transitive property, such as *overlap*. Take relation Rel[*overlap*](A, B ,C) as an example. Given three interval event instances a of type A, b of type B and c of type C, "a *overlap* b and b *overlap* c" cannot infer "a *overlap* c" because that *overlap* relation does not have the transitive property. This representation cannot express the pattern such as "A *overlaps* B, B *overlaps* C and A *overlaps* C". Another reason is that a composite relation might contain more than one temporal relations, such as a composite relation R defined as "A *contains* B and B *overlaps* C".

A *hierarchical representation* [10] is proposed to encode composite relations. Similarly, we can extend our previously defined operator syntax to represent a composite relation with multiple temporal relations as Rel_n(...Rel_2(Rel_1[*list of Allen-based relations*]($E1$, $E2$), $E3$), ..., Em). It can express composite relation such as "A *contains* B, which as a composite event, *overlaps* C". However, such representation is still not expressive enough as it lacks the ability to represent pair-wise relations among events. Thus, this approach still cannot express temporal relations such as "A *contains* B and B *overlaps* C".

We introduce the endpoint-based encoding mechanism to represent temporal relations among interval events, following the work in [16] [17], where an endpoint sequence representation for intervals is studied. The basic idea is to express a relation using the conjunction of *temporal restriction*, which restricts the temporal relation to $<$, $<=$, $=$, $>$ and $>=$ between two interval endpoints. Such conjunction representation is called a *temporal restriction list* (*TList* in short). TList is with the syntax as "TList ::= TList$\wedge$TList | $I_1{}^* < I_2{}^*$ | $I_1{}^* <= I_2{}^*$" | $I_1{}^* = I_2{}^*$", where $I_1{}^*$ and $I_2{}^*$ define two event endpoints.

Example 1. Consider temporal relations *R1*: "A starts earlier than B, B starts earlier than C" and *R2*: "A overlaps B, B overlaps C". They are represented through TLists as "($A.ts < B.ts$) $\wedge$ ($B.ts < C.ts$)" and "($A.ts < B.ts$) $\wedge$ ($B.ts < A.te$) $\wedge$ ($A.te < B.te$) $\wedge$ ($B.ts < C.ts$) $\wedge$ ($C.ts < B.te$) $\wedge$ ($B.te < C.te$)" respectively.

As discussed earlier, primitive temporal relations between two intervals can be expressed using the 13 Allen-based relations and their disjunction. Thus the same expressibility of a TList can be achieved by conjunctions of such Allen-based representation between pair-wise intervals among the given composite pattern. For example, the temporal relation *R1* given in *Example 1* above can be expressed through Allen-based relations as "((A *before* B) $\vee$ (A *meets* B) $\vee$ (A *overlaps* B) $\vee$ (A *finished by* B) $\vee$ (A *contains* B)) $\wedge$ ((B *before* C) $\vee$ (B *meets* C) $\vee$ (B *overlaps* C) $\vee$ (B *finished by* C) $\vee$ (B *contains* C))". However, the endpoint-based approach utilized by TList has the advantage of simplicity in use and it is closer to business rules of the real-world applications [16] [17].

Please note that if there exists any conflicts in a TList, such as ($ep1 > ep2$) $\wedge$ ($ep1 < ep2$) where *ep1* and *ep2* are two event endpoints, the TList becomes invalid. We assume a validating process thus all the TLists in this work are considered valid. Also, a TList representation can be simplified using the transitive property of the $<$, $<=$, $=$, $>$ and $>=$ relations. For example, TList for *R1* in *Example 1* can also be represented as "$A.ts < B.ts < C.ts$".

Using the TList, the EVENT clause of our query language for intervals is with the syntax as "EVENT *ISEQ* [TList]($E1$, $E2$, $E3$, ..., Em; W)", where ISEQ is the temporal operator with the following semantics:

$$\begin{aligned} &ISEQ[TList](E1, E2, ..., Em; W)[H] \ = \\ &\quad \{< e1\ e2\ ...\ em > \mid (TList(e1, e2, ..., em)) \wedge \\ &\quad (< e1\ e2\ ...\ em > \in E1[H] \times E2[H] ... \times Em[H]) \wedge \\ &\quad (max(ei.te)_{i \in \{1,...,m\}} - min(ej.ts)_{j \in \{1,...,m\}} < W)\}. \end{aligned} \tag{1}$$

In the ISEQ operator given above, $\{E1, E2, ..., Em\}$ is the set of event types defined in ISEQ and the TList defines the endpoint relation among the instances. An occurrence number will be attached to distinguish multiple occurrences of the same event type. In most event processing scenarios, it is assumed that the input to the system is a potentially infinite stream which contains all events that might be of interest [23]. Such real-time input is referred to as an *event trace*, which is denoted as H above. For an event trace H and an event type E, $E[H]$ denotes the set of all the event instances of E in H. Given E and E' defined in ISEQ, maximum four different temporal restrictions could be defined: *(1)* $E.ts\ Rel_1\ E'.ts$, *(2)* $E.ts\ Rel_1\ E'.te$, *(3)* $E.te\ Rel_1\ E'.ts$ and *(4)* $E.te\ Rel_1\ E'.te$. Rel_1 to Rel_4 are among possible point-based temporal relations $<$, $<=$, $=$, $>$ and $>=$ ('$>$' is the mirror relation of '$<$' and '$>=$' is the mirror relation of '$<=$'). For any given E in ISEQ, $E.ts <= E.te$ is always a required temporal restriction in the TList. We adopt the reasoning framework on the endpoint-based temporal representation studied in [16] [17]. It introduces an algorithm with exponential complexity which can be used to infer the temporal relation between two endpoints based on a given set of temporal restrictions.

In a traditional point-based event query algebra, the window constraint specification is expressed as the time window parameter, used for restricting the duration length among events in the temporal pattern. In [23], the window expression gives the time window argument W, which specifies the maximum time duration between the occurrences of the first and last events in the event temporal pattern. We follow the operator pushdown approach in [23] to handle the window-based filtering in ISEQ, which uses the window size W to control the maximum span of the result composite, defined as $\max(ei.end)_{i \in \{1,2,...,m\}}$ - $\min(ej.start)_{j \in \{1,2,...,m\}}$.

2.2 ISEQ Operator

The physical implementation of ISEQ has three core operations listed below:

Event Buffering. A newly received event instance is kept in the operator state of ISEQ if it is necessary. Given a newly received interval event e of type E which is among the set of expected events $\{E1, E2, ..., Em\}$, e needs to be buffered into a stack structure referred to as the *instance stack* if and only if it is possible to form result tuples using e together with some other received interval or future coming intervals. So, if E is with a given or inferred temporal restriction as $E.te > E'.te$ and no event instance of E' is currently buffered, the condition that requires an instance to be buffered is not satisfied thus e can be discarded directly (referred to as *on-the-fly dropping*). For other cases, e is added to the corresponding stack for buffering unless its interval length is larger than the window size.

Result Construction. The result construction is performed on the fly triggered by newly arrived tuples to ISEQ. Given a newly received and buffered interval event e of type E among expected event types, new results could possibly be constructed if and only if e might be contained by a result composite event consisting of currently received instances. So, if E is not with a given or inferred temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <= ep$, where ep is another interval endpoint, e could then possibly contribute in forming new result sequences consisting of the current buffered intervals. So the result construction condition is satisfied thus the construction process triggered by e can be called. The process uses a multi-join algorithm based on the attribute constraints on the interval endpoints defined by TList. In the join process, the values of event endpoints (both the start and termination endpoint) are used if the endpoints are associated with some temporal restrictions or with the window constraint.

Operator State Purge. Window constraints can be utilized in ISEQ to avoid unnecessary event buffering. It provides opportunity to purge events from the ISEQ operator dynamically when the event has fallen out of the sliding window. The latter is important in stream processing where runtime data structures need to be pruned to avoid memory depletion. Memory footprint is reduced due to such pruning. In addition, if the checking overhead is kept to be small, CPU footprint can also be reduced because of the saving on buffering-related operators and result construction Furthermore, similar to pushing down the window constraint into SEQ operator in [23], if the purge at ISEQ is conducted on a timely fashion, the checking on window constraint could be skipped thus the corresponding computation for window-based filtering is avoided in the result construction phase. Given a buffered interval event e of type E among expected event types, e can be safely purged from the buffer if and only if it is no longer contributing in forming new result sequences. So, if the termination endpoint associated with E is in given or inferred "$>$" temporal restrictions with all the endpoints in the pattern except itself, the purge condition is satisfied and the event instance e can be purged from the buffer once the result construction process triggered by e (if any) is completely finished. A window-based purge named *cascading purge* could be performed: if E is with a given or inferred temporal restriction as $E.te > E'.te$ and the stack for E' events is empty, all the E events can be safely removed. The process can go on following the chain of such temporal restrictions on the interval termination endpoints. While a fine-grained duration constraint [16] [17] defined in ISEQ, it can be utilized to further avoid unnecessary event buffering. The basic idea is checking the window constraint dynamically while a new interval instance e of type E is received. For a buffered interval ei of Ei with a duration restriction as $Ej.te - Ei.ts(te) <$ W, if $e.te$ - $ei.ts(te) >$ W, ei can be purged from the operator state of ISEQ. The correctness of this window-based purging mechanism is shown as follows. By the arrival of e, we can know any future interval e' satisfies $e.te < e'.te$. Thus, any future Ej instance ej will satisfy $ej.te - ei.ts(te) >$ W. So, e is guaranteed to no longer contribute in forming further results.

An optimization can be brought into this process. Remember that we assume the input interval stream is ordered and the event receiving order at the ESP system is the same

Algorithm 1 Basic ISEQ Operations

```
1: Procedure: $ISEQOperation$
2: Input:
3: (1) event Query EVENT $ISEQ[TList](E1, E2, ..., Em; W)$,
4: (2) newly received event $e$ (under event type $E$)
5: Output:
6: matched result sequences triggered by the input event instance
7:
8: compute the inferred temporal restrictions
9: form the DAG $G$ representing the temporal restrictions
10: compute the indexing scheme
11: if CLOCK updates then
12:     perform window-based purge
13:     perform corresponding cascading purge
14: end if
15: checkState = true
16: if $E$ is among $E1, E2, ..., Em$ then
17:     if $e.te - e.ts < W$ then
18:         if temporal restriction $E.te > E'.te$ exists then
19:             if no event instance of $E'$ is currently buffered then
20:                 checkState = false
21:             end if
22:         end if
23:         if CheckState then
24:             buffer $e$ into the corresponding AIS stack if indexing is
                applied on $E$
25:             if $E$ is not with any temporal restriction as $E.te < ep$,
                $E.te = ep$ or $E.te <= ep$ in $G$, where $ep$ is a vertex in
                $G$ and $ep \neq E.te$ then
26:                 produce event sequences containing $e$ (if any)
27:             end if
28:             if $G$ covers all the endpoints in the pattern and $E$ is
                with a temporal restriction as $E.te > ep$ for any $ep \in G$
                and $ep \neq E.te$ then
                    purge $e$
29:             end if
30:         end if
31:     end if
32: end if
```

as the order of the end time of the event instances. Such order semantics of the input intervals can be utilized to reduce the join computation in the result construction of ISEQ. This is similar to the idea of using a runtime stack nondeterministic finite automaton (NFA) for pattern retrieval on point events [23]. The optimization is for avoiding the multi-join on the longest path of termination endpoints linked through temporal restrictions. Let N denote the length of the path. Then the number of states in the NFA equals N+1 (including the starting state). The data structure *Active Instance Stacks* (AIS) associates a stack with each state of the NFA storing the events that trigger the NFA transition to this state. For each instance in the stack, an extra field *most Recent Instance in the Previous stack* (RIP) records the nearest instance in terms of time sequence in the stack of the previous state to facilitate sequence result construction. When the newly inserted event is an instance of the final stack then AIS computes sequence results. With the AIS states, the construction is simply done by a depth first search in the AIS stacks that is rooted at this instance and contains all the virtual edges reachable from this root. Each root-to-leaf path in the AIS stacks corresponds to the whole or a portion of a matched event sequence, which will be constructed through the rest of the multi-join process. With such AIS data structure, a more sophisticated cascading purge named *cascading AIS purge* could be performed: once an event instance is purged from the AIS stack, events whose RIP field pointing to this event can also be purged.

Algorithm 1 depicts the key ISEQ operations described above. Upon the arrival of a new interval event, buffering decision is made and possible result sequences are produced at the earliest moment. Window-based and cascading purge are performed triggering by the CLOCK updates *Line 11*. The CLOCK value equals to the largest end time timestamp seen from the received intervals. The given and inferred temporal restrictions are managed as a DAG structure [12], with the edges marked as either ">", ">=" or "=". Corresponding construction for applying the AIS data structure is given in *Line 10* and *24*. In addition to that, specific AIS-incorporated computation (*Line 13* and *26*) are plugged in for utilizing the indexing structure.

Example 2. Consider event pattern query $Q = ISEQ[A.ts < B.te < C.te < D.te](A, B, C, D)$ and interval event trace H = "$b_{3|6}$, $d_{6|10}$, $b_{9|11}$, $c_{4|12}$, $a_{7|14}$, $d_{9|15}$, $a_{8|16}$" (shown in *Figure 1*). Interval instance $d_{6|10}$ will be discarded upon arrival through the on-the-fly dropping since no C events are currently buffered and between C and D there is a temporal restriction defined as $C.te < D.te$. While $d_{9|15}$ arrives, the result construction is triggered to produce a result sequence $<a_{7|14}\ b_{9|11}\ c_{4|12}\ d_{9|15}>$. While $a_{8|16}$ arrives, the construction process is triggered again, producing another result sequence $<a_{8|16}\ b_{9|11}\ c_{4|12}\ d_{9|15}>$. Assuming the window size W equals to 30 and we further receive interval $e_{20|35}$, $b_{3|6}$ and $c_{4|12}$ can then be safely purged from the operator state.

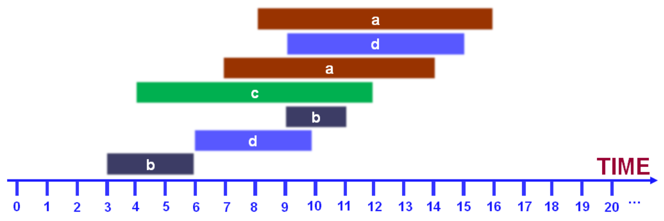

Figure 1: Interval Event Input Example

2.3 Query Execution Strategy

Algorithm 2 sketches the query execution strategy for a long running process of interval event pattern detection. The monitoring process is stopped when the event trace is terminated. Corresponding CPU and buffer resources are released on the fly. During the monitoring process, each received event triggers data buffering, result construction and operator state purge following *Algorithm 1* given in *Section 2.2*.

Algorithm 2 Execution Strategy

```
1: Procedure: $IntervalProcessingExecutionStrategy$
2: Input: real-time evolving interval sequence trace $seq$ as "$e1$, $e2$,
   $e3$ ..." by the order of their termination endpoint, with the End
   of Stream (EOS) message arriving at the very end if input termination is indicated
3: Output: matched result sequences
4:
5: var $e \leftarrow pull(seq)$ /* fetching instance from sequence queue */
6: while $e \neq$ EOS do
7:     process $e$:
8:     perform necessary data buffering and state purge, construct
       new results if possible based on $Algorithm\ 1$
9:     $e \leftarrow pull(seq)$
10: end while
11: terminate the pattern monitor for the current event trace
```

3. TOWARDS EFFICIENT INTERVAL PROCESSING

In many ESP applications, interval events are actually extracted from the raw primitive point events (such as the RFID sensor readings) by business intelligence (referred to as BI) middlewares [9] [18] and then passed to the downstream ESP systems. Consider the previous example given in *Section 1* where an ESP system is used to monitor the events generated by warehouses of a supermarket. Based on the temperature values sent by the temperature readers, temperature fluctuations, *HIGH*, *MEDIUM* and *LOW*, as the interval events are generated and sent to the ESP system. In real-world applications, such temperature fluctuation intervals are actually extracted by the middleware systems which receives the actual readings from the temperature sensors. Assume the *HIGH* temperature is above 100F, the *MEDIUM* temperature is within the range of (50F, 100] and the *LOW* temperature is 50F or lower. We also assume the sensor reads temperature every two seconds and reports the following reading: 01:00:00PM - 55F; 01:00:02PM - 70F; 01:00:04PM - 95F; 01:00:06PM - 80F; 01:00:08PM - 110F; 01:00:10PM - 120F; 01:00:12PM - 90F; 01:00:14PM - 60F; 01:00:16PM - 30F... The interval streams generated will be: "*MEDIUM* 01:00:00PM - 01:00:08PM, *HIGH* 01:00:08PM - 01:00:12PM, *MEDIUM* 01:00:12PM - 01:00:16PM ...". Assume we are having another two interval events, *WET* and *DRY*, to represent the humidity of the environment generated under a similar sensor layer as the one used for the temperature readings. By such context, a practical event query can be looking for the pattern of *HIGH* overlaps *DRY*. Such corner changes which trigger new intervals are called *critical state changes*. These critical state changes (as the begin and end of an interval) are captured by the BI middlewares and then composed into interval events and passed to the downstream ESP systems once the interval is fully formed. Thus, under the above application structure, the information of the "interval start" is actually known to the BI middlewares at real-time. A mechanism to improve the efficiency of interval stream processing is to embed the *interval begin punctuations* defined below into the input interval event streams.

Interval Begin Punctuation (IBP). IBP indicates the initialization of an interval instance. At the moment an interval event e starts, its corresponding IBP will be created and sent. It has associated a metadata schema as ibp_e = <*e.id*, *e.ts*>, where *e.id* is the ID value of e, assigned automatically by the EPS. The ID value is unique among the events in the stream. Given an IBP p, its timestamp $p.t$ equals to $e.ts$.

In the discussion in *Section 2.1*, a data model in which an interval event is an atomic unit semantically is assumed. Thus an interval event is fully composed after it is completed and then it is sent to the ESP system. Applying IBPs does not require the change of this model. However, the IBP information can be used for effective interval event processing. The interval event sender (i.e., the BI middlewares as shown in the warehouse example) should have the mechanism to encode an unique ID to the interval events. The ESP system receives interval event streams mixed together with IBPs. Remember that we assume order for the input interval stream. Under such model which interleaves IBPs with interval event instances, the order of receiving events and IBPs at the ESP system is the same as the order of their end time timestamps. Note that since IBPs are point-based data, the time stamp of an IBP equals to its end time timestamp.

An IBP-aware interval event processing approach can be utilized to reduce the runtime memory and CPU footprint for temporal pattern detection over interval event streams. The key operations of an IBP-incorporated ISEQ operator is given as below.

Event Buffering. The event buffering conditions in the basic ISEQ stays. However, the IBP information is also hold in the AIS for the events being indexed. We will have further discussion on this in the result construction. With the IBP information being available, additional on-the-fly event dropping becomes possible, as follows. Given a newly received IBP of E interval e, which is among the set of expected events $\{E1, E2, ..., Em\}$, if $E.ts$ is among the indexed start endpoints (referred to as the IBP of E being indexed) and *(1)* the AIS stack pointed by the AIS stack of ibp_e is empty, or *(2)* E is with a given or inferred temporal restriction as $E.ts > E'.te$ and no events of E' is currently buffered, the received IBP can be dropped without buffering. Given a newly received interval instance e of type E which is among the set of expected events $\{E1, E2, ..., Em\}$, if *(1)* the IBP of E is required to be indexed and no IBP entry corresponding to e is currently buffered, or *(2)* E is with a given or inferred temporal restriction as $E.te > E'.ts$, the IBP of E' is required to be indexed and no IBPs of E' is currently buffered, or *(3)* E is with a given or inferred temporal restriction as $E.te > E'.te$ and no events of E' is currently buffered, the condition that requires e to be buffered is not satisfied thus e can be discarded directly without buffering.

Result Construction. As discussed earlier, the result construction is performed on the fly triggered by newly arrived tuples to ISEQ. Given a newly received and buffered interval event e of type E among expected event types, new results could possibly be constructed if and only if e might be contained by a result composite event consisting of currently received instances. The conditions for result construction triggering in the basic ISEQ stays for the IBP-incorporated ISEQ. So, if E is not with a given or inferred temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <= ep$, where ep is another interval endpoint, e could then possibly contribute in forming new result sequences consisting of the current buffered intervals. So the result construction condition is satisfied thus the construction process triggered by e can be called. For the part without AIS indexing, the process uses a multi-join algorithm based on the attribute constraints on the interval endpoints defined by TList is applied to construct possible composite events. In the join process, the value of event endpoints (both the start and termination endpoints) are used if the endpoint is associated with some temporal restriction or with the window constraint. The AIS stack is brought into the multi-join process for the indexed temporal restrictions to avoid the joins on a path of event endpoints (both the start and termination endpoints) linked through temporal restrictions using not only the interval termination but also the IBPs. This is different than the AIS-based approach in the basic ISEQ, where the IBPs are not available. The path with the most join avoidance will be selected, which is the longest path in the DAG formed by the temporal restriction, and it is not counted as one join if an edge is formed by one single event type. For event types with only indexed termination endpoints, the AIS structure remains the same as the basic ISEQ operator. For event types with only indexed IBPs, a corresponding AIS stack at first holds the IBPs and later is filled with the corresponding full instances. Similar to SEQ, the RIP pointers can be applied to the stacks consisting of the IBP entries. If the path includes both the start and termination endpoints of an event type, two different AIS stacks will be applied and they both link to a shared structure (referred to as the *full edge stack*) holding the event instance. The construction on the indexed path remains as a simple depth first search in the AIS stacks that is rooted at this instance and contains all the virtual edges reachable from this root. Each root-to-leaf path in the AIS stacks corresponds to the whole or a portion of a matched sequence constructed through the multi-join process.

Operator State Purge. The conditions for operator state purging in the basic ISEQ stays for the IBP-incorporated ISEQ. So, if the termination endpoint associated with E is in given or inferred ">" temporal restrictions with all the endpoints in the pattern except the ones from E itself, the purge condition is satisfied and the event instance e can be purged from the buffer once the result construction process triggered by e (if any) is completely finished. However, more purging opportunities become possible with the IBP being available: the window-based purge and the corresponding cascading purge can be simply extended to cover the IBPs kept in the AIS stacks. The benefits of doing so is that it can lead to further on-the-fly dropping since there could be fewer IBPs kept in the indexes after the purge.

Algorithm 3 depicts the corresponding operations given above. We can see that upon the arrival of a new interval event and an event IBP, buffering decision is made and possible result sequences are produced at the earliest moment. Similar to the basic ISEQ, upon the arrival of new interval events, corresponding construction and operator state purge are triggered to performed. The query execution strategy based on ISEQ given in *Algorithm 2* stays the same for the IBP-incorporated ISEQ.

Example 3. Again consider the scenario given in *Example 2*. Interval event $b_{3|6}$ can be discarded without buffering, since we can guarantee that no future arrival of A could have a start time smaller than $b_{3|6}$'s end time by the fact that no IBPs of A is met before the arrival of $b_{3|6}$. Similarly, interval $b_{9|11}$ is required to be buffered, indicated by the IBP of $a_{7|14}$.

As mentioned earlier for the IBP-based solution, in many ESP applications, event intervals are actually extracted from the raw primitive point-based events (such as the RFID sensor readings) by BI middlewares and then passed to the downstream ESP systems. Following such application structure, the low level physical devices (i.e., the sensor network) with enough computing power would actually be able to capture these critical state changes. Such mechanism of pushing down the computation of interval event abstraction to the low level sensor network can greatly improve the efficiency and scalability for ESP applications with intense computing ability on the physical level devices. This is because

Algorithm 3 IBP-Incorporated ISEQ Operations

```
1: Procedure: PunctuationAwareISEQOperation
2: Input:
3: (1) event Query EVENT ISEQ[TList](E1, E2, ..., Em; W),
4: (2) newly received event IBP $ibp_e$ / event instance $e$ (under event type $E$)
5: Output: matched result sequences triggered by the input event instance
6:
7: same as Line 8 to 14 in Algorithm 1
8: if $E$ is among $E1$, $E2$, ..., $Em$ then
9:   on the arrival of $ibp_e$:
10:    checkState = true
11:    if the IBP of $E$ is required to be indexed then
12:      if the AIS stack pointed by the AIS stack of $ibp_e$ is empty then
13:        checkState = false
14:      end if
15:      if $E$ is with a given or inferred temporal restriction as $E.ts > E'.te$ && checkState && no events of $E'$ is currently buffered then
16:        checkState=false
17:      end if
18:    end if
19:    if checkState then
20:      buffer $ibp_e$ into the corresponding AIS stack by the append semantics
21:    end if
22:    on the arrival of $e$ instance:
23:    checkState = true
24:    startFlag, endFlag = false
25:    if $e.te$ - $e.ts$ < W then
26:      if the IBP of $E$ is required to be indexed then
27:        startFlag = true
28:        if no IBP entry corresponding to $e$ is currently buffered then
29:          checkState = false
30:        end if
31:      end if
32:      if the full instance of $E$ is required to be indexed then
33:        endFlag = true
34:      end if
35:      if checkState && $E$ is with a given or inferred temporal restriction as $E.te > E'.ts$ && the IBP of $E'$ is indexed and no IBP of $E'$ is currently buffered then
36:        checkState = false
37:      end if
38:      if checkState && $E$ is with a given or inferred temporal restriction as $E.te > E'.te$ && no event instance of $E'$ is currently buffered then
39:        checkState = false
40:      end if
41:      if checkState then
42:        if startFlag && !endFlag then
43:          insert $e$ into the corresponding AIS entry based on the event ID
44:        else
45:          if !startFlag && endFlag then
46:            buffer $e$ into the corresponding AIS stack by the append semantics
47:          else
48:            if startFlag && endFlag then
49:              buffer $e$ into the full edge stack and buffer $e$'s reference into the corresponding AIS stack by the append semantics, update existing AIS
50:            else
51:              buffer $e$ into the corresponding instance stack
52:            end if
53:          end if
54:        end if
55:        if $E$ is not with any temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <= ep$ in $G$, where $ep$ is a vertex in $G$ and $ep \neq E.te$ then
56:          produce event sequences containing $e$ (if any)
57:        end if
58:        if $G$ covers all the endpoints in the pattern and $E$ is with a temporal restriction as $E.te > ep$ for any $ep \in G$ and $ep \neq E.te$ then
           purge $e$
59:        end if
60:      end if
61:    end if
62: end if
```

that the computation of interval event abstraction happens much closer to the information source thus the cost of data transportation is avoided.

4. PERFORMANCE EVALUATION

4.1 System Implementation

The system architecture for incorporating the proposed interval event handling into an ESP system structure is shown in *Figure 2*. The ESP system receives events through an *Input Adapter*, which connects to different kinds of data sources, such as system transaction datalogs, supply chain RFID readings, stock market data and e-commerce online transaction data. The ESP connects to two different output sockets, one is the *Result Monitor*, which consists within the *ESP Console*, the other is the *Output Adapter*, which relays output sequences to downstream receivers, such as different operational applications, spreadsheets, BI tools and BI dashboards. The ESP console also includes the *Query Register* for defining customized pattern monitor requirements. The *Plan Generator* parses and translates a given event query into an execution plan. The *Execution Engine*, which constructs results on the fly, is the key component of the ESP system. The definition and implementation of the query operators are contained by the *Operator Containers*, which includes the *Libraries* of the *Logical and Physical Operators*. The proposed ISEQ operator is incorporated into the corresponding operator library containers. While the input is point events (seen as interval events each with the same start time and end time timestamps) and the AIS indexing is applied, the ISEQ operator behaves in the same way as a regular sequence operator. Thus, it can be treated as an extended SEQ operator.

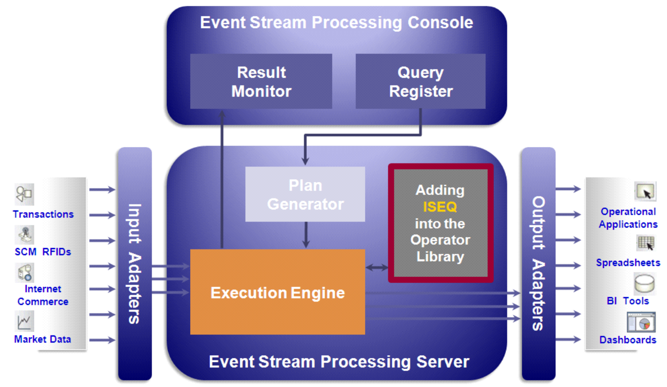

Figure 2: System Architecture

4.2 Experimental Setting

Experiments are run on two Pentium 4 3.0GHz machines, both with 1.98G of RAM. One machine sends the event stream to the second machine. From *Section 4.3* to *4.6* we are going to study the performance of the proposed interval event stream processing techniques on a 2G generated data input, which contains 20 different event types with uniform distribution.

Name	Description
100t-0s	100% termination endpoint
90t-10s	90% terminationendpoint and 10% start endpoint
80t-20s	80% termination endpoint and 20% start endpoint
70t-30s	70% termination endpoint and 30% start endpoint
60t-40s	60% termination endpoint and 40% start endpoint
50t-50s	50% termination endpoint and 50% start endpoint
40t-60s	40% termination endpoint and 60% start endpoint
30t-70s	30% termination endpoint and 70% start endpoint
20t-80s	20% termination endpoint and 80% start endpoint
10t-90s	10% termination endpoint and 90% start endpoint
0t-100s	100% start endpoint indexing

Table 2: Indexing % in Different Query Types

Totally four sets of experiments are run to test the effects of various factors: *(1)* the indexing percentage that controls the indexable endpoints and endpoints types (either start or termination); *(2)* the query length that controls the number of interval patterns in the ISEQ operator; *(3)* the average interval length that controls the average span of the interval events with the normal distribution and *(4)* the event density that controls the number of events within one sliding window with the normal distribution. The applied queries are with the template as "EVENT *ISEQ*[TList](A, B, ...; W)", where the TList defines the endpoint temporal restrictions among the event patterns. Performances of *(1)* the basic ISEQ without AIS indexing (referred to as *naive ISEQ*) approach, *(2)* the basic ISEQ with AIS indexing (referred to as *basic indexing*) approach and *(3)* the IBP-incorporated ISEQ (referred to as *IBP-incorporated*) approach are measured respectively. Experimental results are given in *Section 4.3* to *4.6* below.

4.3 Varying Query Types

This set of experiments varies the percentage of indexable endpoints as well as the indexable endpoints types in the given query. The indexable endpoints will contribute to the AIS construction for the basic ISEQ with AIS indexing approach and the IBP-incorporated ISEQ approach. Ten different combinations are covered by the experiments, which is shown in *Table 2*. The query length is fixed as 10. The average interval length is fixed as W/10 (W is the sliding window size, which is fixed as 30 seconds for all queries) with the event density as 200 events per window. Results are shown in *Figure 3* and *4*. The property of the input event data such as the average interval length and event density greatly affects the performance, which will be studied later in *Section 4.5* and *4.6*.

Memory Consumption (*Figure 3*). X axis here shows the ten groups of queries categorized by indexing scheme discussed earlier (*Table 2*) and Y axis shows the accumulative memory consumption for each query. With the cascading AIS purge, the basic indexing approach and the IBP-incorporated approach both have less memory footprint than the naive ISEQ approach except the case with no termination endpoint indexing for the basic indexing approach. However it only shows a slight gain (less than 5% for the case with the most gain) under the given setting. With a smaller window, which can be achieved by increasing the average interval length or decreasing the event density, more memory footprint can be avoided. This will be further discussed in *Section 4.5* and *4.6*. Addition to that, for the basic indexing approach, the gain on memory consumption is affected by the percentage of indexable termination endpoints in the query.

CPU Performance (*Figure 4*). X axis still shows the ten different indexing scheme and Y axis shows the execution time for each query. We can see that the IBP-incorporated approach in all cases outperform the naive ISEQ approach. This is because that it has indexing support for all the query categories due to the IBP utilization. In most cases the basic indexing approach outperforms the naive ISEQ approach: with a higher percentage of the termination indexing, more CPU computation could be avoided in terms of result sequences construction using the costly multi-join algorithm. For example, in the best case (i.e., the query with 100% indexable termination endpoint patterns), execution with the basic indexing approach reduce the execution time of the plan with naive ISEQ by 60%. However, while the percentage of indexable termination endpoints is not high in the given query, the basic indexing approach has poor performance because the overheads on index construction and maintenance. The overhead ranges from 3% to 12% in the query categories of *20t-80s*, *10t-90s* and *0t-100s*. The overhead increases while decreasing the portion of indexable termination endpoints in the query. We can also observe that the basic indexing approach does not perform as well as the IBP-incorporated approach. This is due to the cost avoidance using the IBP information in the IBP-incorporated approach is not applicable for the basic indexing approach.

4.4 Varying Query Length

This set of experiments studies how varying the relative query length affects the interval stream processing cost. The query length is varied from 2 to 18. For example, among them a sequence query with length 6 (i.e., $ISEQ[A.ts < B.ts < C.ts < D.te < E.te < F.te](A, B, C, D, E, F)$) is run. The *50t-50s* indexing profile is applied to all the queries in this set of experiments. The average interval length is fixed as W/10 with the event density as 200 events per window, which stays the same as *Section 4.3*. Experimental results are shown in *Figure 5* and *6* and analysis is given as follows.

Memory Consumption (*Figure 5*). X axis here represents the query length and Y axis shows the accumulative memory consumption for each query. We can see that the ratio of memory consumption saving (the slight saving on memory footprint discussed earlier in *Section 4.3*) stays relatively steady for the index-applied approaches while the query length increases, since the interval events among different types are with uniform distribution.

CPU Performance (*Figure 6*). X axis still represents the query length and Y axis shows the execution time for each query. A query with a longer length requires much more CPU resources for the result construction than the naive ISEQ approach. Thus we can see that the ratio of CPU gain increases sharply for the index-applied approaches while the query length increases. Similar observation can be found in the comparison between the two index-applied approaches. The ratio of the IBP-incorporated approach's CPU gain over the basic indexing approach increases steadily while the query length increases, from 45% to 66%.

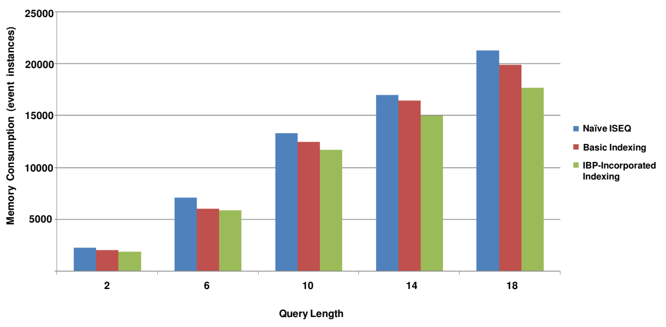

Figure 5: Varying Query Length (I)

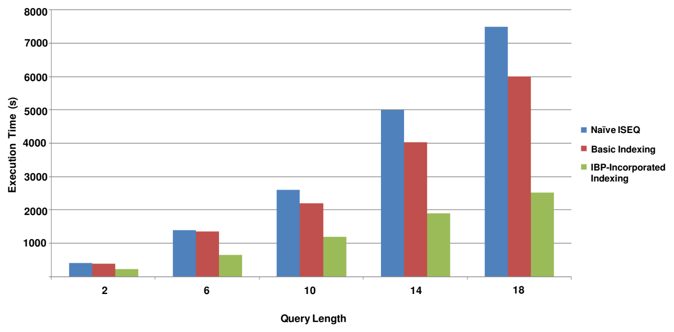

Figure 6: Varying Query Length (II)

4.5 Varying Average Interval Length

Since the input event stream is infinite, consistent performance over time can only be achieved by actively maintaining the data structures incrementally based on the given window constraint of the query [2]. Thus the size and density of the interval events both affect the cost of buffer consumption and the result construction since they both affect the amounts of active instances kept in the operator state. We next study the effect of interval size by varying it from W/100 to W/5. Similar to the earlier settings, the *50t-50s* indexing profile is applied to all the queries in this set of experiments. The event density is set to 200 events per window and the query length is set to 10. Experimental results are shown in *Figure 7* and *8* and analysis is given as follows.

Memory Consumption (*Figure 7*). X axis here represents the interval length and Y axis shows the accumulative memory consumption for each query. We can see that with larger intervals (thus relatively smaller sliding window size in terms of holding how many complete interval events), more memory footprint can be avoided for the IBP-incorporated approach. The ratio of the memory consumption gain scales with the average interval length. This is because that more intervals can be discarded directly through the on-the-fly dropping and more cascading AIS purge can be applied while intervals become easier to fall out of the sliding window. Similar observation can be found while comparing the basic indexing approach and the naive ISEQ approach.

CPU Performance (*Figure 8*). X axis still represents the interval length and Y axis shows the execution time for each

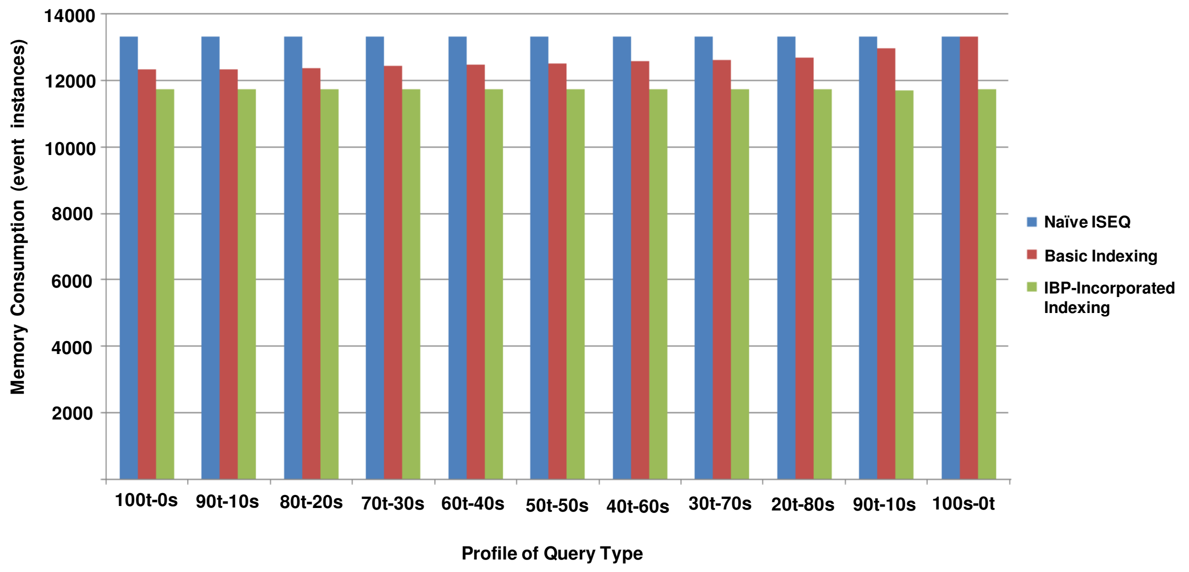

Figure 3: Varying Query Types (I)

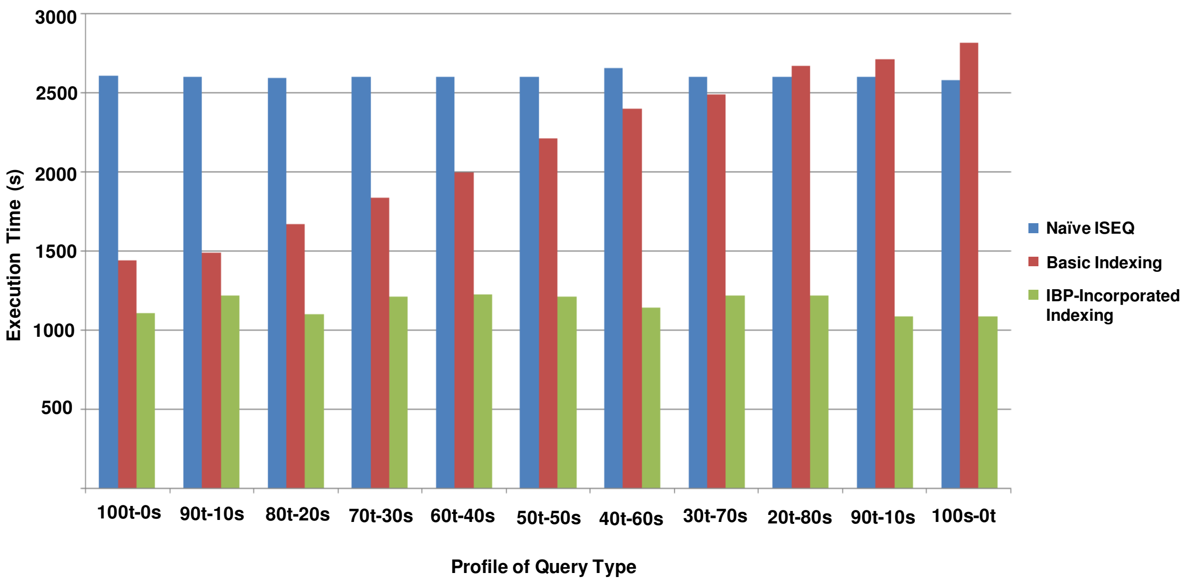

Figure 4: Varying Query Types (II)

query. Similar to the observation on the memory consumption, we can see that with larger intervals, more CPU cost can be avoided for both index-applied approaches, with a gain ratio in proportion to the interval length.

4.6 Varying Event Density

As the discussion in *Section 4.5*, the size and density of the interval events both affect the cost of buffer consumption and the result construction. In this set of experiments we study the effect of event density by varying it from 50 events per window to 800 events per window. Note that for intervals we consider the event center (the middle point of the interval) as its representation. Similar to the earlier settings, the *50t-50s* indexing profile is applied to all the queries in this set of experiments. The average interval length is given as W/20 and the query length is set to 10. Experimental results are shown in *Figure 9* and *10* and analysis is given as follows.

Memory Consumption (*Figure 9*). X axis here represents the interval length and Y axis shows the accumulative memory consumption for each query. We can see that with more sparse input (thus relatively smaller sliding window size in terms of covering how many interval event centers), more memory footprint can be avoided for the IBP-incorporated approach. The ratio of the memory consump-

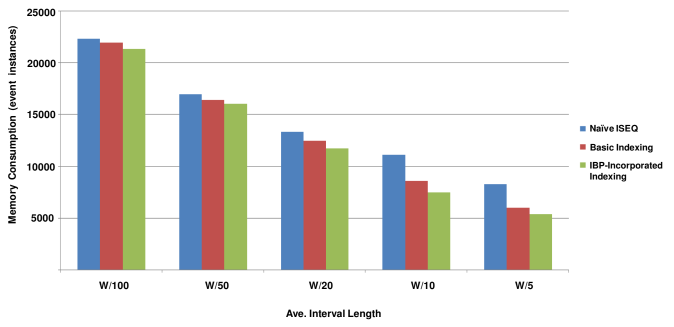

Figure 7: Varying Average Interval Length (I)

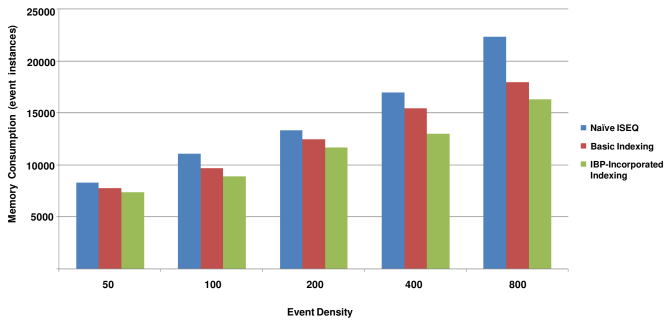

Figure 9: Varying Event Density (I)

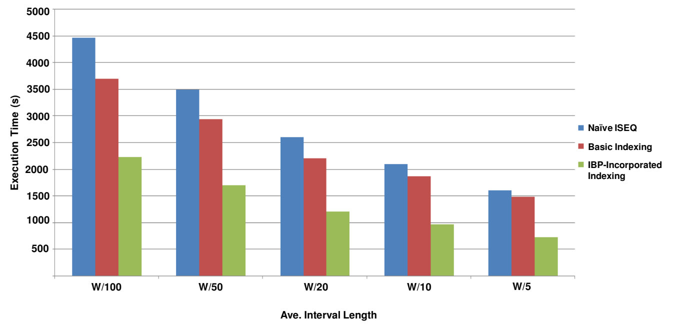

Figure 8: Varying Average Interval Length (II)

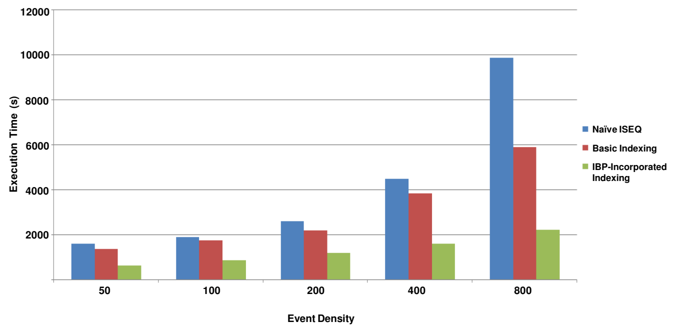

Figure 10: Varying Event Density (II)

tion gain is in inverse proportion to the event density. This is because that with a more sparser input data set, less data will be hold by the operator since the state purge. Similar observation can be found while comparing the basic indexing approach and the naive ISEQ approach.

CPU Performance (*Figure 10*). X axis still represents the interval length and Y axis shows the execution time for each query. Similar to the observation on the memory consumption, we can see that with more sparse input, more CPU cost can be avoided for both index-applied approaches. However, the ratio is no longer just in inverse proportion to the event density when the input becomes very dense. We can see that the CPU cost increases sharply for the naive ISEQ approach comparing with the index-applied approaches while the query density jumps from 200 to 400 and from 400 to 800. Similar observation can be found for the comparison between the two index-applied approaches. This is because that with larger operator state, the result construction for the patterns without indexing becomes more and more inefficient.

4.7 Conclusions of the Experimental Studies

Above experimental results reveal that the proposed interval event stream processing framework is practical in three senses: *(1)* interval streams are handled correctly by the proposed framework with expected query results; *(2)* the index-applied approaches outperform the naive ISEQ approach in most cases and *(3)* the IBP-incorporated outperforms the approach with the basic indexing.

5. RELATED WORK

Event-specific ESP technology, which has an event-specific system design and evaluation mechanism, is shown to be superior to generic stream processing solutions [1] [5] [11] [21] because it is being specifically designed for handling pattern queries over streaming event. In [23], the authors propose an expressive yet easy-to-understand language to support pattern queries on such sequential streams and propose customized algebra operators for the efficient processing of such pattern queries with sliding windows. [3] uses a plan-based technique to perform streaming complex event detection across distributed sources. These researches on event pattern detection over event streams mainly focused on extracting temporal patterns from point-based event data [23]. Even though in [3] [8] [7] the events are defined based on the interval model. However, only the "*before*" / "*after*" temporal relation is supported, which simplifies the interval-based temporal model to the point-based temporal model by overlooking the patterns where events as intervals can overlap with each other. [25] studied sequence pattern detection for point events with imprecise timestamps, where an event could occur somewhere within a time interval with uniform distribution. Instead our model considers interval events occurring over the entire time range. The data mining community studied discovering patterns over interval events [10] [19] [24]. [10] uses a hierarchical representation that extends Allen's interval algebra [4] for modeling complex event patterns over intervals. However, this representation is lossy as the exact relationships among the events cannot be fully recovered. [24] [19] devise a lossless representation to overcome the drawbacks of [10]. Based on their proposed representation, they propose corresponding mining algorithms for pattern discovering over interval events. [24] proposes the TPrefixSpan algorithm to mine the new temporal patterns from interval events. The completeness and accuracy of the results are also proven. Their experimental results show that the efficiency and scalability of the TPrefixSpan algorithm are satisfactory. An efficient algorithm called IEMiner is designed by [19] to discover frequent temporal patterns from interval events. The algorithm employs two optimization techniques to reduce the search space and remove unpromising candidates. [19] also examines how the discovered temporal patterns can be utilized in classification to differentiate closely related classes thus building an interval-based classifier called IEClassifier. Even though our endpoint representation is also lossless as in these works, we cannot adapt their algorithms because they mainly focus on pattern discovering algorithms instead of pattern detection algorithms. Besides that, they do not consider streaming input with window constraints.

6. CONCLUSIONS

Existing ESP systems have focused on detecting temporal patterns from instantaneous events, that is, events with no duration. However, such sequential patterns are inadequate to express the complex temporal relations in domains such as medical, multimedia, meteorology and finance where the durations of events could play an important role. Due to the differences between the temporal patterns over interval events and point events, the query semantics and evaluation mechanisms used for pattern detection over point events is not sufficient for pattern detection over interval events. An expressive language to represent the required temporal patterns among streaming interval events and corresponding evaluation mechanism for such event temporal queries is needed. In this work, we provide a framework to support interval event stream processing: *(1)* we introduce an expressive language to represent the required temporal patterns among streaming interval events; *(2)* we design the corresponding temporal operator ISEQ; *(3)* for further improving the event processing performance, we provide a mechanism to embed the "interval begin punctuation" into the interval stream and study the corresponding punctuation-aware interval processing; *(4)* we conduct experimental studies to validate our proposed approach.

7. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[3] M. Akdere, U. Cetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *PVLDB*, 1(1):66–77, 2008.

[4] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[5] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[6] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *DEBS*, pages 50–61, 2010.

[7] 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.

[8] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. In *ICDE*, pages 676–685, 2008.

[9] D. M. Eyers, L. Vargas, J. Singh, K. Moody, and J. Bacon. Relational database support for event-based middleware functionality. In *DEBS*, pages 160–171, 2010.

[10] P. Kam and A. W. Fu. Discovering temporal patterns for interval-based events. In *DaWaK*, pages 317–326, 2000.

[11] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

[12] D. Kozen. Automata and computability. In *W.H.Freeman and Company, New York*, 2003.

[13] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin. Constraint-aware complex event pattern detection over streams. In *DASFAA*, pages 199–215, 2010.

[14] M. Li, M. Mani, E. A. Rundensteiner, D. Wang, and T. Lin. Interval event stream processing. In *DEBS*, 2009.

[15] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.

[16] B. Nebel and H.-J. Burckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. In *AAAI*, pages 356–361, 1994.

[17] B. Nebel and H.-J. Burckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *J. ACM*, 42(1):43–66, 1995.

[18] A. Paschke and P. Vincent. A reference architecture for event processing. In *DEBS*, 2009.

[19] D. Patel, W. Hsu, and M. Lee. Mining relationships among interval-based events for classification. In *SIGMOD*, pages 393–404, 2008.

[20] G. Rosu and S. Bensalem. Allen linear (interval) temporal logic - translation to ltl and monitor synthesis. In *CAV*, pages 263–277, 2006.

[21] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[22] D. Toman. Point vs. interval-based query languages for temporal databases. In *PODS*, pages 58–67, 1996.

[23] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[24] S. Wu and Y. Chen. Mining nonambiguous temporal patterns for interval-based events. *IEEE Trans. Knowl. Data Eng.*, 19(6):742–758, 2007.

[25] H. Zhang, Y. Diao, and N. Immerman. Recognizing patterns in streams with imprecise timestamps. *PVLDB*, 3(1):244–255, 2010.